Intégration de systèmes répartis

- Problématiques
  - construction de systèmes selon les organisations
    - hétérogénéité des technologies, des formats, des protocoles
    - rarement interopérables
  - comment, lorsqu'un événement se produit dans un système, déclencher un traitement dans un système étranger ?
  - comment assurer la consistance et la propagation des données entre plusieurs systèmes
- Evolution historique des solutions
  - batch
  - client-serveur : de données, distribué, à objets distribués
  - services

Limites du client-serveur

- La communication est de 1 vers 1 ou n vers 1
- Les interlocuteurs sont désignés explicitement
- L'organisation de l'application est peu dynamique
  - facilité d'évolution, adjonction et retrait d'entités participantes
- Le schéma de base est synchrone

Limites d'un échange synchrone

- Pour invoquer des services en synchrone, il faut :
  - que le service soit en état de marche, à l'instant où il est invoqué
  - qu'il soit joignable par le réseau
- Si non, on renonce et l'application doit être prévue pour traiter l'échec de l'appel
- Si conditions réunies : combien de temps attendre la réponse ?
- Synchrone : dépendance très forte entre deux applications
  - ok si les deux applications sont sur la même plateforme, avec temps de réponse garantis
- Asynchrone, pas d'attente d'une réponse => faible dépendance (couplage lâche) => plus grande flexibilité dans les architectures

Middleware

- Logiciel qui permet à différentes applications d'échanger et d'interopérer, généralement lorsqu'elles tournent sur des serveurs différents interconnectés par un réseau
- Il offre ses services aux applications, mais les échanges induits s'appuient sur toute une pile de protocoles réseau
  - davantage qu'un simple protocole d'appel de services
    - RPC, RMI ou SOAP ne sont pas considérés comme des middlewares
- Deux grandes familles
  - ceux permettent l'invocation synchrone de méthodes, les request brokers (CORBA ou DCOM)
  - les middlewares d'échange asynchrones, qui sont principalement à base de messages, les MOM

Services d'un middleware

- L'identification et la localisation des applications, à un niveau au dessus des adresses réseau et des noms de serveurs, et l'acheminement des échanges à ce niveau
- Dans certains cas, la conversion de formats de représentation des données entre les applications, permettant à des applications d'environnements et langages différents d'échanger de manière transparente
- Dans certains cas, des fonctions de sécurité, de répartition de charge ou de gestion du secours

Qu'est-ce qu'un MOM ?

- Middleware Orienté Messages : implémentation du principe d'échanges asynchrones
  - standard : la spécification JMS nombreuses implémentations
- Les applications communiquent en échangeant des messages
- Message : l'objet véhiculé par le MOM
  - rien n'est imposé au message (taille, ou format des données) : conventions entre les 2 applications
  - le MOM, ne s'intéresse pas au contenu du message, ne fait que le transmettre, et le remet au destinataire sans changement

Enterprise Application Integration

- Dans un MOM, les applications doivent parler le même langage
  - un EAI au contraire prend en charge les traductions entre représentations différentes
- Englobe les fonctionnalités du MOM, et ajoute des possibilités facilitant l'intégration des applications au niveau des données transférées
  - réalise transformations sur les messages, afin d'adapter les formats de données émetteur/destinataire
- Middleware qui a comme principales fonctions :
  - l'interconnexion des systèmes hétérogènes
  - la gestion de la transformation des messages
  - la gestion du routage des messages

Enterprise Service Bus

- Concept plus ambitieux : socle uniforme d'une architecture SOA globale
- L'EAI peut prendre en charge des transformations de formats entre applications
- L'ESB généralise le concept
  - une application interfacée à l'ESB peut interopérer par son intermédiaire avec toute autre application interfacée à l'ESB
  - la connexion à l'ESB n'est pas exclusivement à base de messages, elle supporte une grande diversité de modes d'échange et de protocoles

Event-Driven Architecture

- Architecture pilotée par les événements, alternative à l'approche SOA
- Idée : tout traitement est exécuté en réaction à un événement et tout traitement est générateur d'événements
  - tout est événement, tout est réaction à des événements
- L'approche SOA est malgré tout par essence une approche synchrone (même si elle peut se décliner dans une logique asynchrone)
- La réaction à un événement est par essence asynchrone, même si elle impose des exigences de rapidité
- Les MOM sont le support naturel d'une approche EDA

Principes

- L'émetteur et le récepteur utilisent une "destination"
  - plusieurs émetteurs et récepteurs sur la même destination
- Persistance du message (reçu ou non reçu)
  - reprise après panne
- Mode de synchronisation :
  - communication asynchrone ; émission non bloquante
  - réception bloquante (attente d'arrivée d'un message, ou retour d'erreur)
- Format du message libre

Format des messages

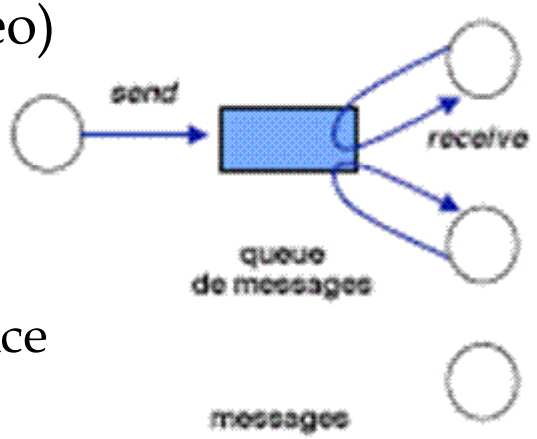
- Entête
  - information permettant l'identification et l'acheminement du message
  - identificateur unique, destination, priorité, durée de vie, etc.
- Attributs
  - couples (nom, valeur) utilisables par le système ou l'application pour sélectionner les messages (opération de filtrage)
- Données définies par l'application
  - Texte
  - Données structurées (XML)
  - Binaire
  - Objets (sérialisés)

Modes de désignation

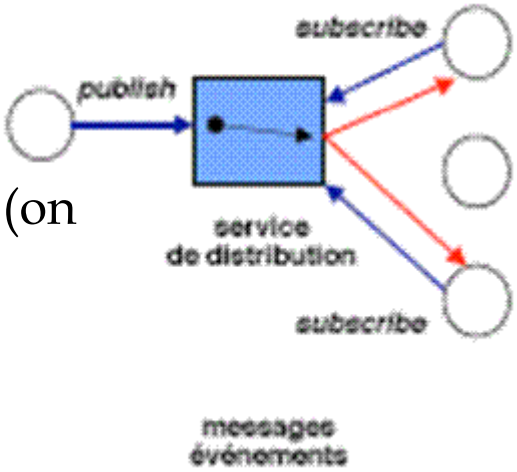
- Désignation indirecte
  - les entités communiquent via un objet intermédiaire : destination
    - destination : structure de données réceptacle de messages
    - exemple : Queue (file) de messages
- Désignation de groupe
  - groupe = ensemble de consommateurs identifiés par un nom unique
    - gestion dynamique du groupe : arrivée/départ de membres
    - différentes politiques de service dans le groupe : 1/N, N/N
- Désignation anonyme
  - désignation associative : les destinataires d'un message sont identifiés par des propriétés (attributs du message)
  - Publish/Subscribe (Publication / Abonnement)

Modèles de messageries

- Mode pull : point à point (ex: consultation météo)
  - l'émetteur livre une information
  - réception explicite des messages (retrait)
    - le récepteur va chercher l'information
    - il n'est pas nécessairement bloqué en l'absence de message



- Mode push : publish/suscribe (ex: Publicité, spam)
  - l'émetteur diffuse une information
  - réception des messages de manière implicite
    - abonnement préalable du destinataire au service (on peut filtrer les messages par leurs attributs)
    - lors du dépôt d'un message, chaque destinataire est informé
    - modèle "événement-réaction"

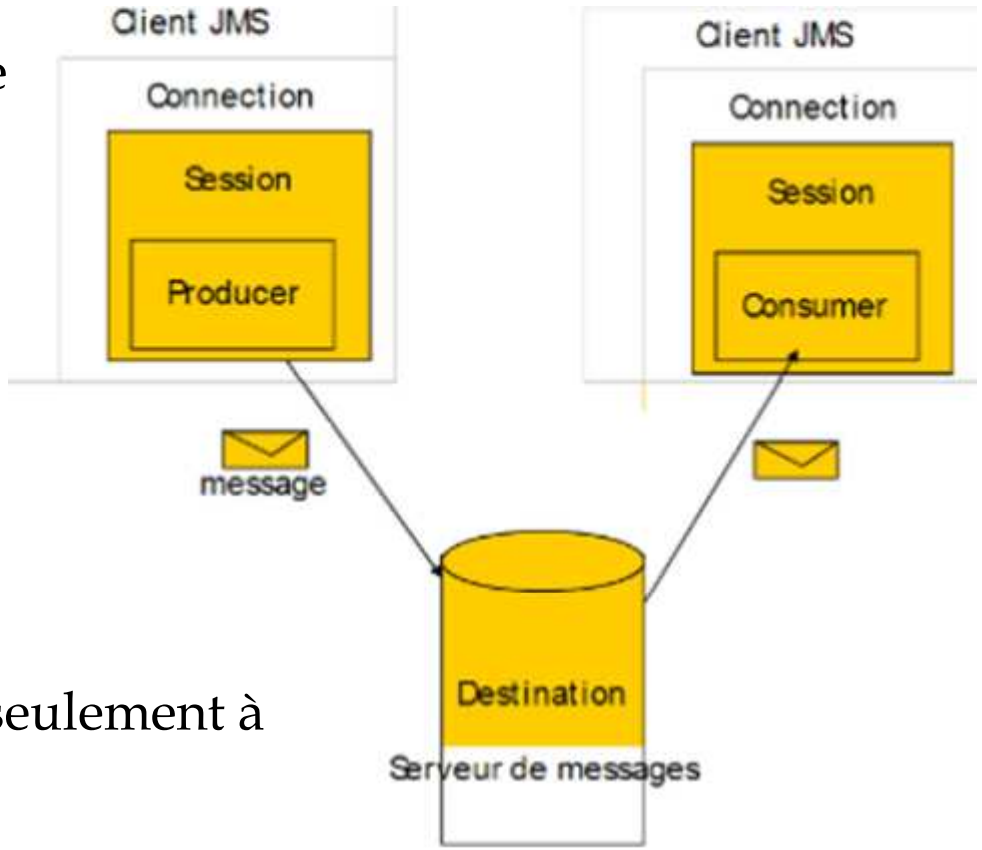


Mise en œuvre

- JMS (Java Message Service)
  - interface Java standard pour communication par message
  - ne définit pas d'implémentation
- Event Service de CORBA
  - l'un des Object Services définis par l'OMG
  - ensemble de classes permettant de réaliser un service d'événements
  - supporte les modes push et pull
- IBM Websphere MQ, Novell, Oracle, ActiveMQ, OpenMQ
  - des implémentations spécifiques ont été réalisées
  - supporte point à point et publish/subscribe
- Jini (Sun puis Apache)
  - service de coordination d'objets partagés (Java)
  - utilisé pour des environnements dynamiquement variables (découverte de ressources)

Architecture JMS

- Plusieurs émetteurs et plusieurs récepteurs sur une même destination
- Une connexion est liée à un serveur de message
- Une session existe à l'intérieur d'une connexion
- Il peut y avoir plusieurs sessions par connexion
- La session gère le processus global de transmission
- Le consommateur/producteur existe seulement à l'intérieur d'une session
- Le consommateur/producteur connaît la destination
- Le message n'existe qu'à l'intérieur d'une session



Types de destinations

- Queue pour le point à point
  - chaque message n'a qu'un consommateur
  - pas de couplage temporel fort
  - le consommateur envoie un reçu
  - le message est détruit à la réception du reçu
  - il peut y avoir plusieurs consommateurs en attente
- Topic pour le publish/subscribe
  - similaire à un modèle d'événements
  - un message peut avoir plusieurs consommateurs par abonnement
  - abonnement et activité du consommateur requise
  - tous les abonnés reçoivent le message publié

Interfaces de base

- Package javax.jms
- Les interfaces
  - ConnectionFactory
  - Connection
  - Destination
  - Session
  - MessageProducer
  - MessageConsumer
- Exception JMSException

Etapes du mode Point-à-Point

Emetteur | Destinataire

Trouver la queue : utilisation de JNDI
Créer une connexion et une session
Démarrer la session

Instancier un objet

pour envoi sur la queue

Envoyer un message sur la queue

Instancier un objet

pour réception sur la queue

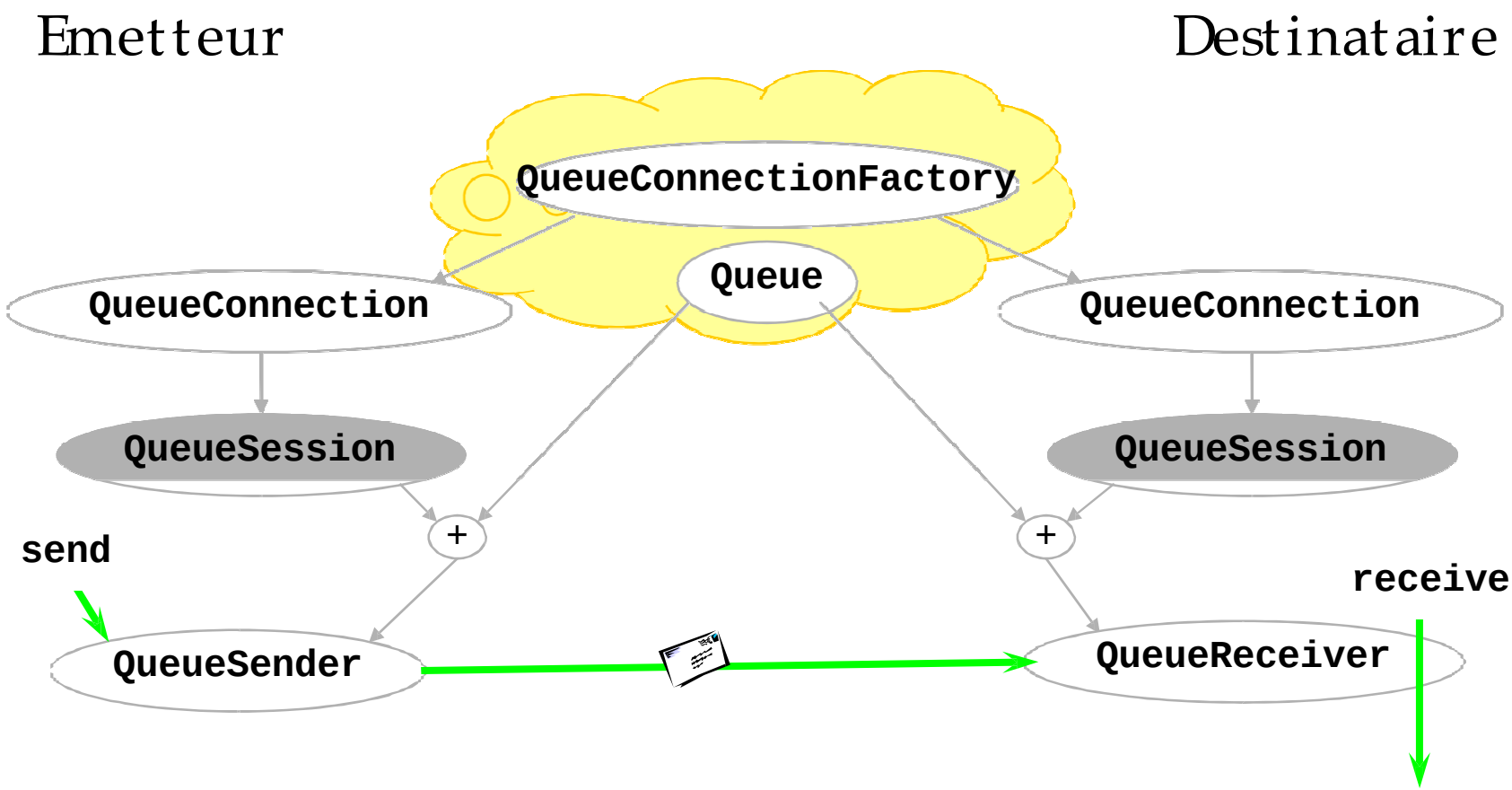
Demander la réception d'un message

Clore la connexion

JNDI

- Java Naming and Directory Interface : interface commune pour d'autres services existants : DNS, CORBA, RMI, …
  - fournit des fonctionnalités pour nommer et répertorier des informations
  - cache tous les détails d'implémentation des différents services, permet à plusieurs services de cohabiter, toutes les informations semblent être fédérées sous un seul service de nommage
- Package javax.naming
- Classe InitialContext point de départ de toute recherche
  - la méthode lookup de cette classe retourne un Object, son paramètre est une chaîne de caractères : le nom de l'élément recherché
  - il faut ensuite convertir le type de cet objet pour qu'il correspond réellement à l'interface recherchée
  - ne pas oublier de gérer les exceptions qui peuvent être levées

Interfaces du mode PTP



Mise en œuvre du mode PtP

Emetteur

```
InitialContext messaging = new InitialContext();
QueueConnectionFactory connectionFactory = (QueueConnectionFactory)
        messaging.lookup("…");
Queue queue = (Queue) messaging.lookup("…");
QueueConnection connection = connectionFactory.createQueueConnection();
QueueSession session =
connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
connection.start();
```

```
QueueSender sender =session.createSender(queue);

TextMessage msg = session.createTextMessage();
msg.setText("…");
sender.send(msg);
```

Destinataire

```
QueueReceiver receiver =
        session.createReceiver(queue, selector);

TextMessage msg = (TextMessage) receiver.receive();
```

Etapes du mode Pub/Sub

Emetteur

Destinataire

Trouver la queue

Créer une connexion et une session

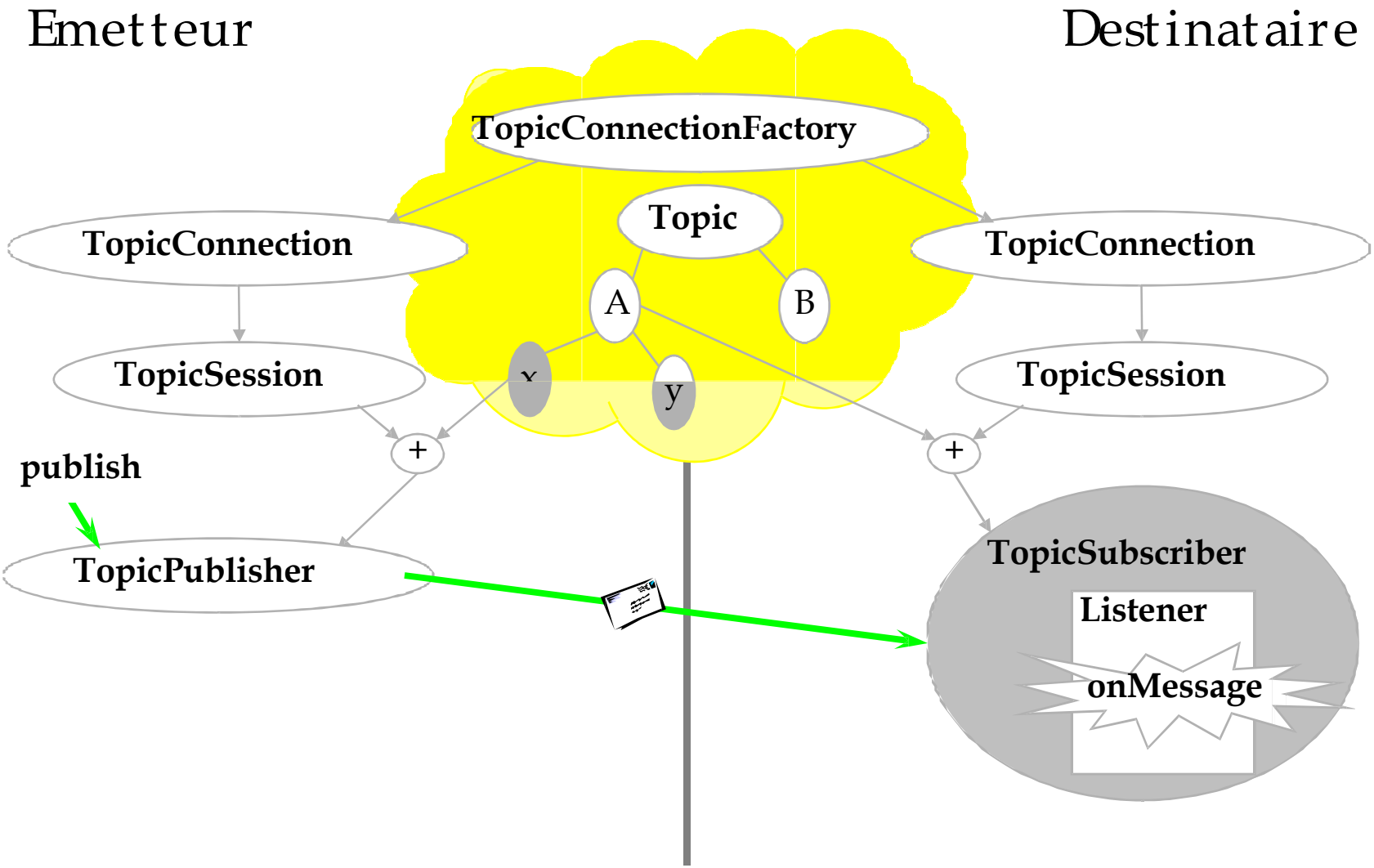
Créer un sujet

Publier un message

Préparer un message à écouter

Se mettre à l'écoute d'un sujet

Interfaces du mode Pub/Sub



Mise en œuvre du mode Pub/Sub

Emetteur

```
TopicConnectionFactory connectionFactory = (TopicConnectionFactory) messaging.lookup("…");
Topic topic = (Topic) messaging.lookup("…");
TopicConnection connection = connectionFactory.createTopicConnection();
TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();

TopicPublisher publisher =
        session.createPublisher(topic);

publisher.publish(msg);
```

Destinataire

```
void onMessage(Message msg)
        throws JMSException {
// gérer le message
…
}
TopicSubscriber subscriber =
        session.createSubscriber(topic);
Subscriber.setMessageListener(listener);
```

Rôle des objets impliqués

- Une connectionFactory est un objet utilisé par un client JMS pour créer la connexion au fournisseur de service
- Une connexion encapsule une connexion virtuelle au fournisseur de service JMS
  - cela pourrait être par exemple un socket TCP/IP ouverte entre un client et le provider. On s'en sert pour créer un ou plusieurs sessions
- Une session est un contexte unique dans lequel sont créés producteurs ou consommateurs de messages
- Une destination est un objet utilisé par un client JMS pour spécifier où stocker les messages produits ou où trouver la source des messages qu'il consomme
  - en mode PTP une destination est nommée queue
  - en mode pub/sub une destination est nommée topic

Les différents types de messages

- TextMessage : chaîne de caractères
- MapMessage : un ensemble de paires nom-valeur dont les noms sont des String et les valeurs de types primitifs
- BytesMessage : un flot d'octets sans interprétation particulière
- StreamMessage : un flot de valeurs de type primitif Java alimenté et relu séquentiellement
- ObjectMessage : un objet sérialisable en Java (dont la classe implémente Serializable

Filtrer les messages

- On peut associer à un message un certain nombre de propriétés qui pourront être utilisées pour faire des sélections de message à la réception
- Pour associer une propriété à un message, méthode (du message) setStringProperty
- Pour ne recevoir que certains messages, on crée le consommateur en utilisant un second paramètre pour la méthode createReceiver : un sélecteur de message
- Un sélecteur de message est une String qui contient une expression
  - syntaxe basée sur un sous-ensemble des expressions conditionnelles de SQL92
  - exemple : sujet = 'Sports' OR sujet= 'Actu'

Accusé de réception

- Tant qu'on n'a pas accusé réception d'un message, il n'est pas considéré comme totalement consommé
- L'émission de l'accusé de réception peut être sous la responsabilité du fournisseur JMS ou du client, suivant la configuration
- Le 2nd paramètre de la méthode create...Session peut valoir
  - Session.AUTO_ACKNOWLEDGE
    - l'accusé de réception est automatique quand le message est consommé
  - Session.CLIENT_ACKNOWLEDGE
    - on utilise alors acknowledge() de la classe Message
  - Session.DUPS_OK_ACKNOWLEDGE
- Dans la mesure où l'émetteur et le récepteur sont découplés, l'accusé de réception intervient uniquement dans la communication entre un client et le fournisseur JMS

La réponse à l'émetteur

- Si l'émetteur attend une réponse du récepteur, il pourra créer une destination temporaire sur laquelle le récepteur pourra émettre un message
  - cette destination ne dure que le temps de la connexion qui l'a créée
  - on utilise une des méthodes de Session createTemporaryQueue ou createTemporaryTopic
  - pour que le récepteur connaisse la destination temporaire, on configure le message à l'émission avec setJMSReplyTo
- Le récepteur
  - crée un Sender et ne l'associe pas à une destination (null)
  - utilise la méthode send avec un premier paramètre : la destination temporaire qu'il obtient avec getJMSReplyTo sur le message
  - pour éviter une confusion entre 2 messages du même émetteur, celui-ci peut affecter à son message un numéro (setJMSCorrelationID) que le récepteur peut éventuellement utilisé dans son message de réponse

```
replyMsg.setJMSCorrelationID(msg.getJMSCorrelationID());
```

La gestion des transactions

- On peut regrouper plusieurs opérations dans une unité de travail : une transaction
  - si une des opérations échoue, la transaction est annulée (rollback) sinon elle est validée (commit)
- Dans un client, on peut utiliser les transactions locales pour grouper des émissions et des réceptions de messages
  - il s'agit d'une transaction gérant des interactions uniquement entre un client et un provider JSM, jamais 2 clients JMS
  - le message expédié n'est pas mis dans la destination de manière définitive tant que la transaction n'est pas validée
- On utilise le premier paramètre de create...Session que l'on positionne à true si on veut démarrer une transaction
- On utilise les méthodes commit et rollback de session pour terminer les transactions

Les messages prioritaires

- On peut demander à ce qu'un message soit distribué en urgence
- Deux possibilités
  - soit en utilisant la méthode setPriority du MessageProducer producer. setPriority(6);
  - soit en utilisant la forme longue de la méthode send producer.send(message, DeliveryMode.PERSISTENT, 6, 10000);
  - si la valeur est 0, le message n'expire jamais
- Il existe 10 niveaux de priorité de 0 (la plus basse) à 9 (la plus haute)

Réception asynchrone

- Lorsqu'une queue reçoit un message, le provider déclenche la méthode onMessage des objets qui sont à l'écoute
- Cette méthode est définie dans une interface MessageListener
  - elle doit être déclarée public
  - elle ne doit pas être déclarée final ou static
  - elle n'a qu'un seul argument, un objet de type javax.jms.Message
  - elle ne retourne pas de résultat
- Elle contient le traitement qui sera associé à la réception du message
- Tout objet qui instancie une classe implémentant MessageListener sera interrompu dans son traitement si sa méthode onMessage est invoquée

Abonné durable

- Les messages persistants sont des messages qui seront conservés même si le provider JMS ne fonctionne plus
- La méthode Session.createConsumer crée un souscripteur non durable càd qui ne recevra les messages qui s'il est actif
- La méthode Session.createDurableSubscriber permet de créer un souscripteur durable
  - tout message persistant sera délivré lorsque l'abonné sera actif
  - il faut préciser un identifiant unique pour la connexion

    connection.setClientID("unNom");
  - il faut préciser un nom de souscripteur unique à sa création

  MessageConsumer mc=sessi.createDurableSubscriber(leTopic, "unNom");
- Pour supprimer une souscription durable, on utilise la méthode unsubscribe de Session

  session.unsubscribe(" unNom ");

Expiration de message

- Par défaut un message n'expire jamais
- Si on veut le rendre obsolète au bout d'un certain temps, on peut lui affecter un temps d'expiration (en millisecondes)
- Deux possibilités
  - soit en utilisant la méthode setTimeToLive du MessageProducer
    producer.setTimeToLive(60000);
  - soit en utilisant la forme longue de la méthode send
    producer.send(message, DeliveryMode.PERSISTENT, 3, 10000);
  - si la valeur est 0, le message n'expire jamais

Les serveurs d'application Java EE

- Ce sont des logiciels qui proposent un contexte d'exécution pour des applications web
- Permettent au développeur de se concentrer sur la logique métier et de se dégager des problématiques standard
- Prennent en charge la recherche d'un service à exécuter, son cycle de vie, la gestion de la charge, la gestion de la sécurité, la gestion des transactions, ...
  - ils sont totalement configurables, et utilisent sur des API appropriées
- On y déploie des composants qui sont conformes à une spécification (Servlet, EJB, Services WEB, ...)
  - on les configure en utilisant des annotations dans le code
- Les clients peuvent être des programmes autonomes, des applets ou d'autres composants

Principes d'un EJB

- Composant qui est développé en respectant une spécification (JavaEE)
  - encapsule une logique métier
  - est déployé dans le serveur d'application
  - pet être composé de un ou plusieurs objets
- S'exécute dans un conteneur, qui est configuré, et qui a la charge du cycle de vie, de l'activation, des accès concurrents, de l'accès aux ressources ...
- Plusieurs types d'EJB
  - Session : pour effectuer un traitement, une session de travail
    - on y accède via une interface
  - Entity : pour gérer des données persistantes
    - le conteneur est configuré pour être associé à une base de données
  - MessageDriven : pour gérer des messages

Les Message Driven Bean

- On intègre JMS dans les serveurs d'application
- C'est un EJB qui va recevoir des messages
  - il peut consommer des messages depuis une queue ou un topic
    - selon que l'on souhaite que le message ne soit consommé qu'une fois ou par plusieurs consommateurs
- On n'y accède jamais à partir d'un client
  - un MDB est découplé du producteur de message
  - c'est le conteneur qui est le consommateur de messages
- Il est sans état transactionnel mais, à la différence d'un EJB Session Stateless, il n'a pas d'interface
- Il fonctionne en mode asynchrone
- Il n'a pas de valeur de retour

Implémentation d'un MDB

- Il implémente l'interface javax.ejb.MessageDrivenBean
  - on utilisera une annotation @MessageDriven avant la déclaration de la classe
  - l'attibut mappedName de cette annotation permet d'associer à la destination d'où sont consommés les messages
    - le MDB et son conteneur seront configurés par injection de dépendance
  - exemple :@MessageDriven("jms/QPremierEssai")
- Il est en attente jusqu'à arrivée d'un message
  - il implémente donc l'interface javax.jms.MessageListener
- Il ne possède qu'une seule méthode onMessage()
  - cette méthode a pour paramètre un message JMS
  - le type n'est pas vérifié à la compilation
  - pour connaître le type du message à l'exécution, on utilise instanceof

MDB - Exemple

```
@MessageDriven(mappedName="jms/QPremierEssai")
public class MonMessageDBean implements MessageListener {
        //@Resource private MessageDrivenContext mdc;
        public void onMessage(Message msg) {
                TextMessage tmsg = null;
                try {
                        if ( message instanceof TextMessage ){
                                tmsg = (TextMessage) msg;
                                System.out.println ("Message reçu : " + tmsg.getText( ));
                        }  else System.out.println ("Message pas de type texte");
                } catch (JMSException e) {
                        e.printStackTrace( );
        //              mdc.setRollbackOnly( );
}       }       }
```

Configuration d'un MDB

- La configuration de la source de messages se fait en utilisant l'attribut mappedName de l'annotation MessageDriven
- On peut choisir de filtrer les messages suivant certaines propriétés, l'émission d'accusé de réception, de configurer les différentes ressources (en particulier le contexte d'exécution), le fait que le MDB soit durable ou pas, ...
- On utilise l'attribut activationConfig de l'annotation MessageDriven qui est un tableau de propriétés ActivationConfigProperty
- Exemple :

```
activationConfig = { @ActivationConfigProperty(
                        propertyName = "messageSelector",
                        propertyValue = "dest = 'MDB9'")}
```

La gestion des erreurs

- Les MDB n'ont pas de valeur de retour
- Rien n'est prévu pour renvoyer un message à l'expéditeur, il faut tout programmer
  - le MDB crée un message de réponse et envoie ce message dans un destination qui a été précisée dans le champ JMSReplyTo du message initial
- Les MDB sont découplés du client donc ne peuvent pas lui renvoyer d'exception
  - ils peuvent renvoyer une exception au container qui agira en fonction : le message n'est pas considéré comme consommé ...
- En cas de problème suite à la réception du message, on peut avoir à annuler le travail en cours

La gestion des transactions

- Série d'opérations qui apparaissent comme une unique opération, qui réussit ou qui échoue
- Le bean est automatiquement enrôlé (enrolled) dans une transaction
- Plusieurs types de transaction : Bean-Managed, Container-Managed; pas de transaction
- Si on décide de gérer les transactions dans le bean, on n'a pas de moyen de conserver le message dans la queue de destination si un problème arrive
- Sinon on peut accéder au contexte d'exécution pour éventuellement marquer la transaction pour annulation par le conteneur en utilisant la méthode setRollBackonly

Les messages empoisonnés

- Lorsque le MDB échoue, le message n'est pas acquitté, il reste dans la file d'attente
- A cause des transactions un message peut ne jamais être consommé
- On peut alors :
  - traiter le problème dans le bean et ne pas lever d'exception
  - faire gérer les transactions par le bean et non pas par le container,
  - configurer le serveur pour gérer une queue de messages "dead message queue" et le paramétrer pour un nombre maximal de tentatives d'envoi

Les traitements associés

- En général le MDB est uniquement le récepteur du message, mais ce n'est pas le composant qui effectuera l'action qui accompagne la réception du message
- Il devient alors client d'un composant sur le serveur d'application qui exécutera le traitement
- Pour une application WEB, l'infrastructure est du type
  - une JSP ou servlet qui émet un message
  - MDB qui consomme le message et qui démarre un
  - EJB Session qui effectue le traitement
- Le MDB et son EJB Session associé sont souvent packagés dans la même archive web (war) et déployés ensemble sur le serveur

Services Web

- L'émetteur peut être aussi une application complètement indépendante, écrite en Java ou pas
- On propose alors avec le MDB un service web qui permettra de transmettre un message reçu
- Le service web reçoit sa demande en SOAP (XML sur HTTP) et renvoie sa réponse en SOAP
- Les services proposés par le service Web sont décrits en WSDL que le client doit utiliser pour générer les artefacts nécessaires à la communication sur le web

Exemple de service Web

```
import javax.jws.*;

@WebService(targetNamespace="http://localhost:8080")
public class WSBonjour
{
        @WebMethod
        public String salut(String s)
        {
                return "Salut "+s;
        }
}
```