# Intergiciels pour la répartition
# JMS : Java Message Service

Patrice Torguet
torguet@irit.fr
Université Paul Sabatier

# Plan du cours

- Introduction aux MOM
- Introduction à JMS
- Quelques mots sur JNDI
- Connexions et Sessions
- Destinations, producteurs et consommateurs
- Les Messages
- Mécanisme de Requête/Réponse
- Exemple complet

# Introduction aux MOM

- MOM : Message Oriented Middleware
- Intergiciels orientés Messages
- But :
  - Création de messages complexes (pas uniquement des tableaux d'octets)
  - Emission/Réception de messages
  - Eventuellement : stockage de messages

# Introduction aux MOM

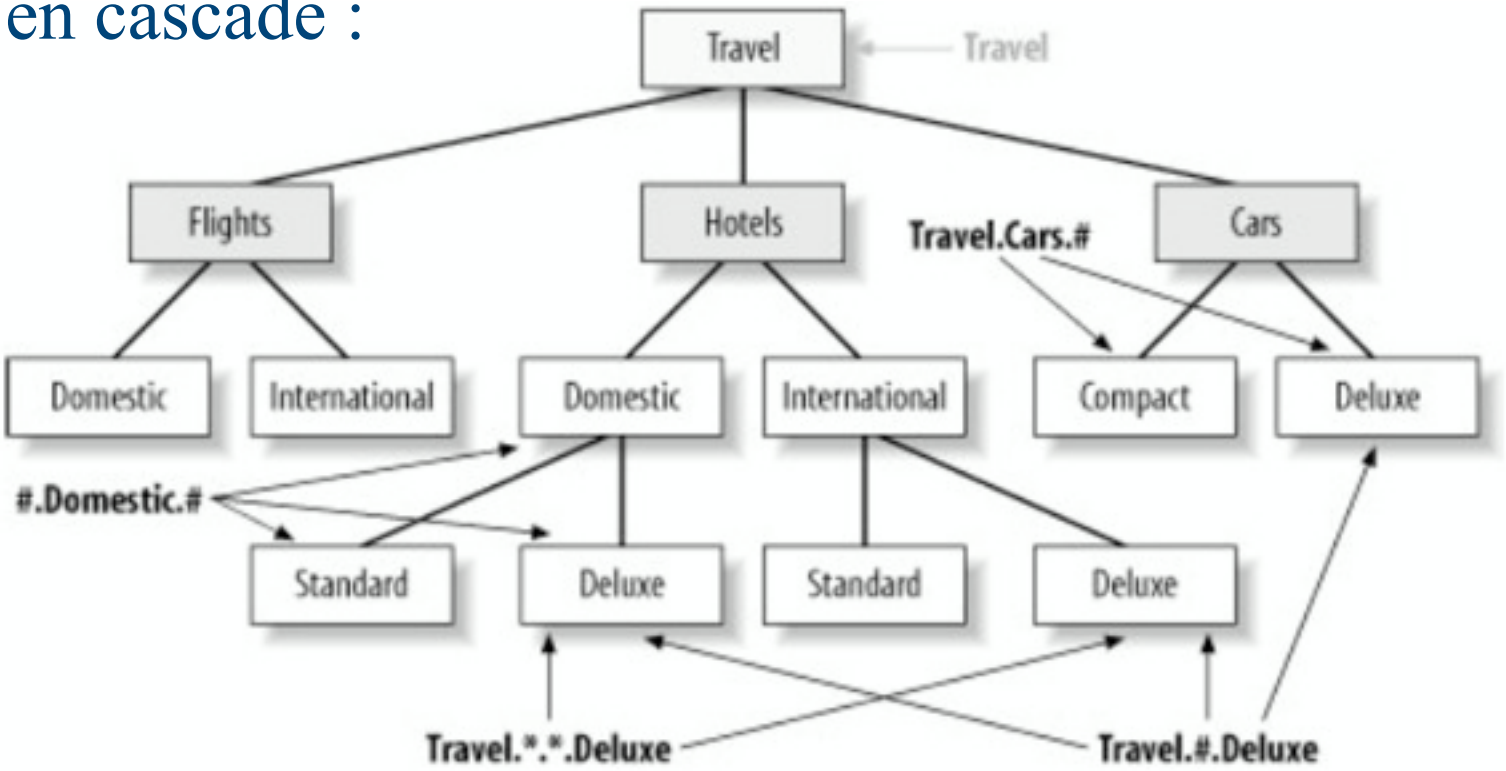
- Message Passing et Message Queuing
- Les messages peuvent soit :
  - être envoyés directement d'une application à l'autre. C'est ce qu'on appelle du passage de message. MPI est un exemple de MOM utilisant le Message Passing.
  - être stockés dans des files de message avant d'être récupérés ultérieurement.

# Introduction aux MOM

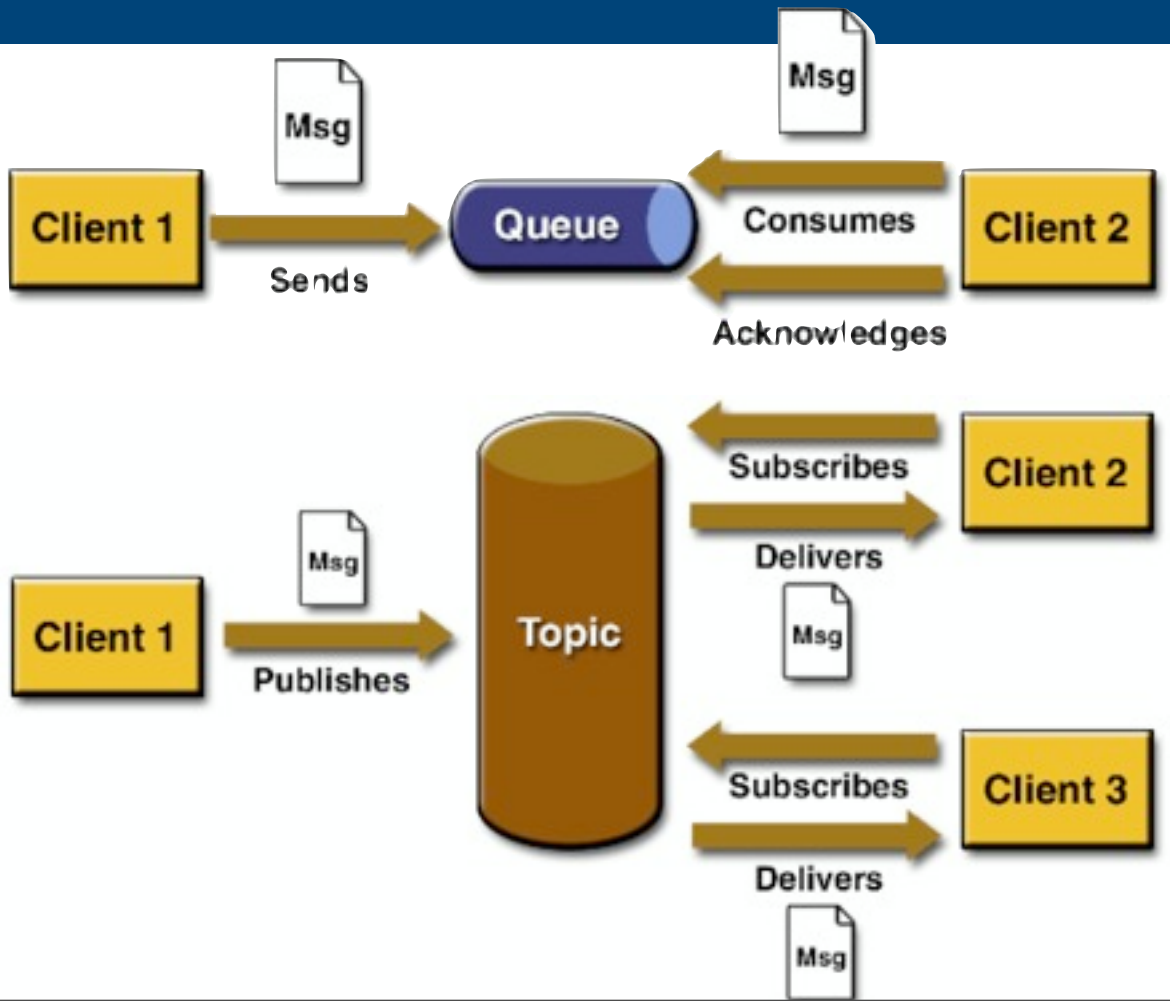
- La plupart des MOM gérant le Message Queueing permettent 2 concepts :
- Les files de messages : qui stockent les messages produits jusqu'à leur consommation.
- Les sujets de discussion : qui permettent une communication un vers plusieurs. Des applications s'abonnent à un sujet et lorsqu'un message est envoyé sur le sujet, tous les abonnés le reçoivent.

# Introduction aux MOM

- Les sujets de discussion peuvent parfois fonctionner en cascade :



# Introduction aux MOM



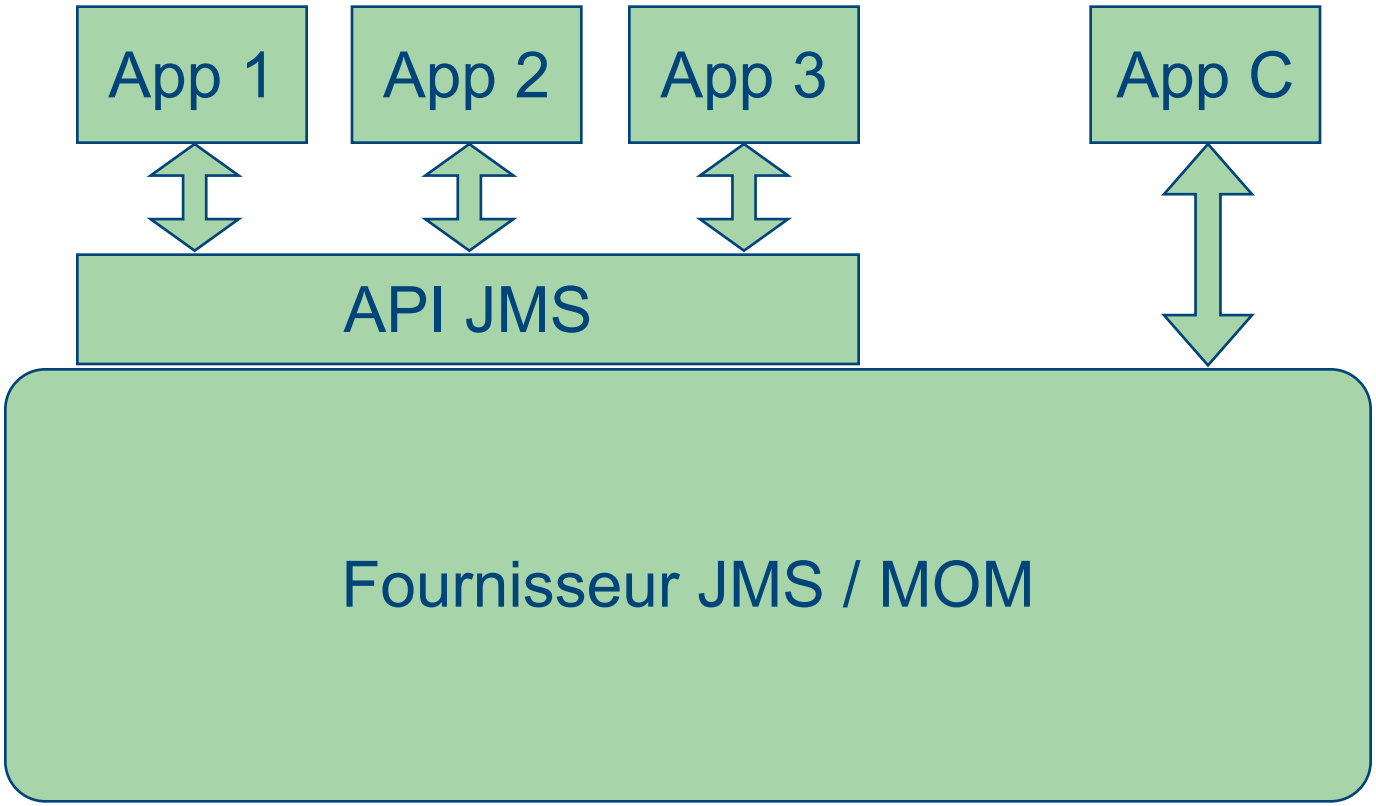
# Introduction aux MOM

- Avantages des MOM par MQ
  - Le plus gros avantage est l'asynchronisme des communications. Une application peut produire des messages alors que le récepteur n'est pas présent. Ceci est très utile pour gérer le nomadisme.
  - Pour les sujets on dispose du concept d'abonnement persistant (une file stocke les messages publiés pour un abonné qui n'est pas présent).
  - Autre avantage : sûreté de fonctionnement. Si le récepteur s'arrête à cause d'une erreur, les messages non traités complètement seront disponibles après le redémarrage.

# Introduction à JMS

- JMS est une API Java qui permet de travailler avec des MOM fonctionnant en mode Message Queuing
- Pour travailler avec JMS il faut :
  - l'API JMS qui permet d'écrire les applications qui s'échangent les message : package javax.jms disponible dans jms-1.1.jar
  - Un fournisseur (provider) JMS qui gère l'administration des files de message, le stockage des messages... Il peut soit être implémenté en Java ou être un front-end JMS pour une application existante écrite dans un autre langage

# Introduction à JMS



# Introduction à JMS

- Ce que définit JMS
  - La création de messages structurés (entêtes, propriétés, corps)
  - La connexion a un fournisseur JMS nommé via JNDI
  - La récupération d'une file ou d'un sujet nommés via JNDI
  - La production d'un message dans une file ou un sujet
  - La consommation d'un message depuis une file ou un sujet

# Introduction à JMS

- Ce qui n'est pas défini par le standard
  - L'administration (création, destruction, vidage...) des files
  - Comment sont stockées les files
  - La répartition d'un fournisseur JMS
  - L'interopérabilité entre plusieurs fournisseurs JMS (cf. MOMA : MOM Association)
  - ...

# Introduction à JMS

- Hiérarchie de classes/interfaces
  - JMS définit deux ensembles de classes/interfaces pour les files et pour les sujets (exemple QueueConnection et TopicConnection)
  - pour permettre une utilisation générique JMS définit un ensemble supplémentaire de classes génériques (exemple Connection)
  - Ici nous présenterons essentiellement cet ensemble générique.

# Introduction à JMS

- MOMs compatibles JMS :
  - ActiveMQ de Apache
  - OpenJMS de The OpenJMS Group
  - JBoss Messaging de JBoss
  - JORAM de Objectweb
  - Weblogic de BEA
  - Oracle AQ
  - SAP NetWeaver WebAS Java JMS de SAP AG
  - SonicMQ de Progress Software
  - Sun Java System Message Queue

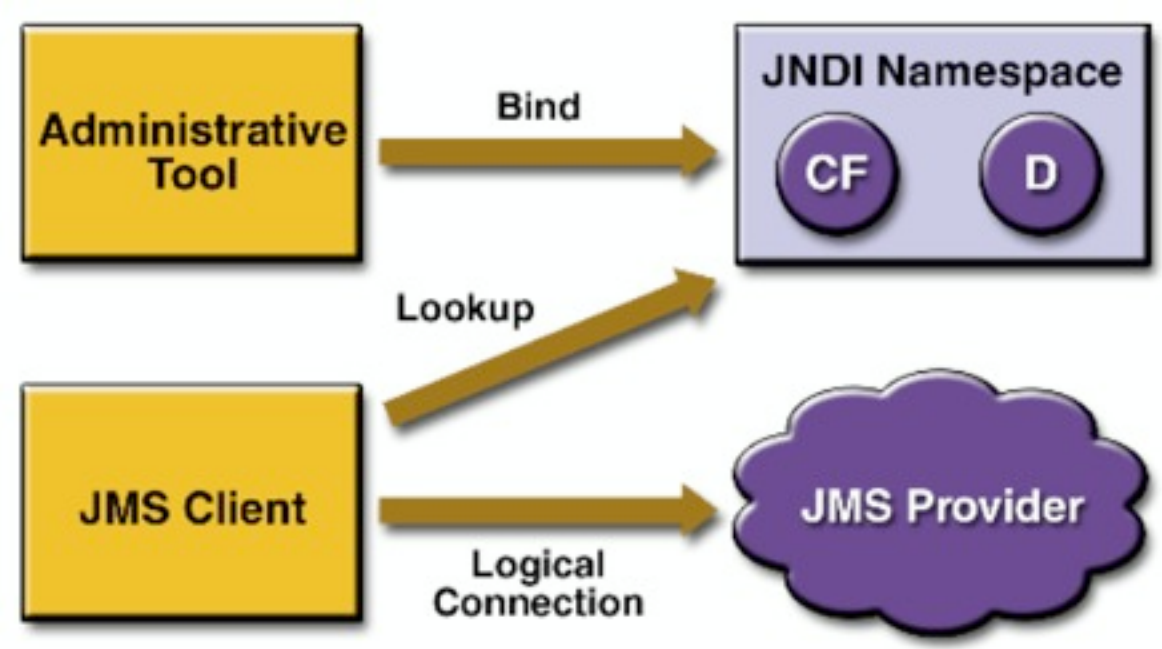
# Introduction à JMS

- MOMs compatibles JMS (suite) :
  - TIBCO Software
  - webMethods Broker Server de webMethods
  - WebSphere Application Server d'IBM
  - WebSphere MQ d'IBM (appelé précédemment MQSeries)

# Quelques mots sur JNDI

- JNDI : Java Name and Directory Interface
- API pour gérer des annuaires de données ou d'objets
- Les annuaires peuvent être gérés par des fichiers ou par des systèmes d'annuaires répartis comme LDAP
- JNDI offre essentiellement deux méthodes :
  - Bind : qui permet de donner un nom à une ressource
  - Lookup : qui permet de trouver la ressource à partir du nom
- JNDI est utilisé par RMI, JMS et Java EE

# Quelques mots sur JNDI



# Connexions et Sessions

- JMS utilise deux concepts pour gérer la connexion entre un client et le fournisseur :
- Les connexions qui peuvent gérer la concurrence via des threads. Elles sont créées à partir d'une usine à connexion (ConnectionFactory) récupérée via JNDI. Lors de leur création on peut préciser un login/mot de passe.
- Les sessions qui sont mono-thread. Elles sont créées à partir d'une connexion. Elles peuvent gérer des transactions recouvrables (commit/rollback)

# Connexions et Sessions

- Code :

```
Context contexte = null;
Connection connexion = null;
String factoryName = "ConnectionFactory";
Session session = null;
// note ce code peut générer des NamingException et JMSException
// création du contexte JNDI.
contexte = new InitialContext();

// récupération de la ConnectionFactory
ConnectionFactory cf = (ConnectionFactory) contexte.lookup(factoryName);

// création de la connexion
connexion = cf.createConnection();
// avec login : connexion = cf.createConnection("login", "pass");

// création de la session sans transaction et avec des acquittements automatiques
session = connexion.createSession(false, Session.AUTO_ACKNOWLEDGE);
// avec transaction : session = connexion.createSession(true, 0);
...
```

# Destinations, producteurs et consommateurs

- Les destinations représentent les files/sujets que l'on utilise pour produire/consommer des messages. Elles sont récupérées via JNDI.
- Une session peut créer des producteurs (MessageProducer) et des consommateurs de message (MessageConsumer)
- Pour pouvoir recevoir il faut démarrer la connexion : connexion.start(); !!!

# Destinations, producteurs et consommateurs

- Les consommateurs peuvent être synchrones (bloquants avec ou sans expiration) ou asynchrones (callback)
- Pour les callbacks on utilise l'interface MessageListener
  - On doit définir la méthode void onMessage(Message m) qui va traiter les messages dès qu'ils sont disponibles
  - Attention : cette méthode doit récupérer toutes les exceptions (y compris les RuntimeException) !!!
  - Puis on doit référencer le listener auprès du consommateur : conso.setMessageListener(ml);

# Producteur Anonyme

- Pour éviter d'avoir un producteur pour chaque destination on peut créer un producteur anonyme :

```
MessageProducer anon_prod = session.createProducer(null);
```

- Pour l'utiliser on utilisera une version de la méthode send qui prends une destination en paramètre :

```
anon_prod.send(dest, message);
```

# Destinations, producteurs et consommateurs

- Code :

```
  ...
MessageProducer mp = null;
MessageConsumer mc = null;

// récupération de la Destination
Destination dest = (Destination) contexte.lookup("FileProduction");
// création du producteur
mp = session.createProducer(dest);

// récupération de la Destination
Destination dest = (Destination) contexte.lookup("FileConsommation);
// création du consommateur
mc = session.createConsumer(dest);

// démarre la connexion (il faut le faire pour pouvoir recevoir)
connexion.start();
...
```

# Destinations, producteurs et consommateurs

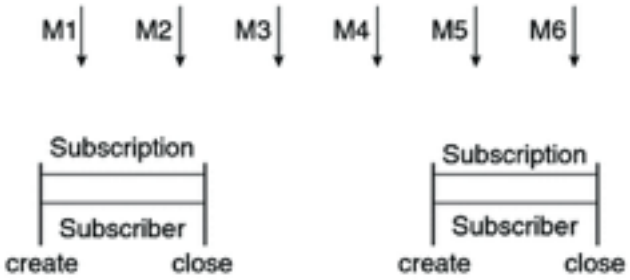
- Spécifique aux files :
  - On peut créer à la place d'un MessageConsumer un QueueBrowser pour examiner les messages sans les prélever

# Destinations, producteurs et consommateurs

- Spécifique aux sujets :
  - On peut créer un abonné persistant en utilisant la classe TopicSubscriber (à la place de MessageConsumer).
  - On devra préciser un nom qui sera utilisé pour retrouver les messages stockés pendant que le programme de réception n'est pas disponible
  - D'autre part il faut que le client est le même identifiant (géré via l'appli d'administration du fournisseur)
  - Pour détruire l'abonné persistant il faudra utiliser la méthode unsuscribe de Session

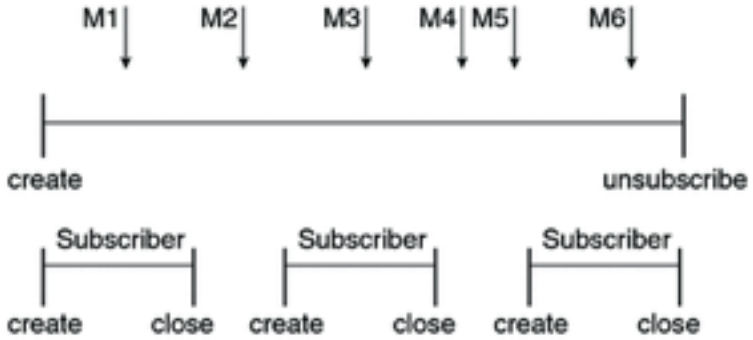
# Destinations, producteurs et consommateurs

- Sans Abonné persistant :




M3 et M4 ne sont pas reçus

- Avec Abonné persistant




Tous sont reçus

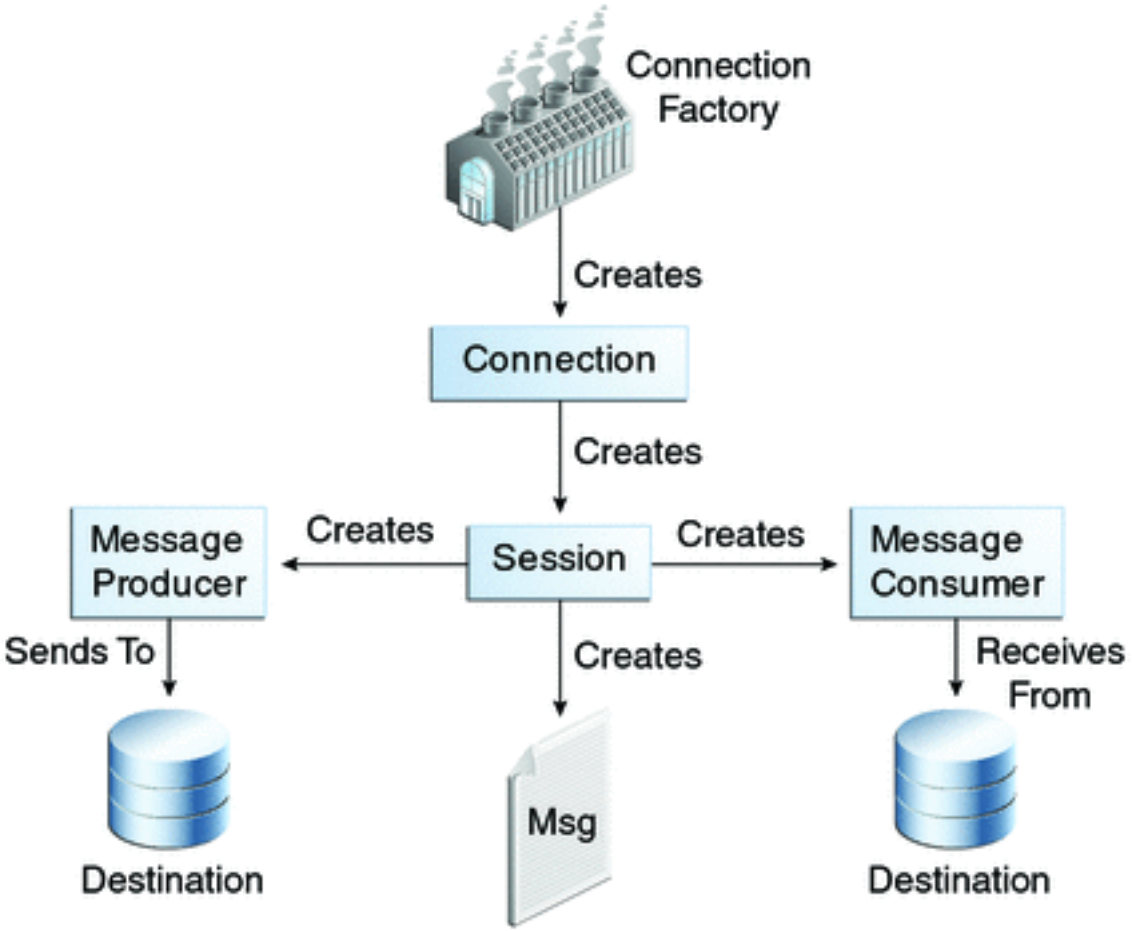
# Destinations, producteurs et consommateurs

- Code :

```
// création du browser
QueueBrowser browser = session.createBrowser(queue);
// démarrage de la connexion
connection.start();
// consultation des messages présents
Enumeration messages = browser.getEnumeration();
while (messages.hasMoreElements()) {
    Message message = (Message) messages.nextElement();
}
```

```
// création de l'abonné persistant (
TopicSubscriber subscriber = session.createDurableSubscriber(topic, "nomAboPersist");
// démarre la connexion. Si l'abonné existait déjà on va recevoir les messages
// en attente dès ce moment
connection.start();
while(true) {
     Message message = subscriber.receive();
}
```

# Schéma récapitulatif



# Usines à Connexions, Destinations et Java EE

- Avec un serveur d'entreprise Java EE on utilise en général l'injection de ressource pour les CF et les destinations ainsi :

```
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(lookup = "jms/Queue")
private static Queue queue;
@Resource(lookup = "jms/Topic")
private static Topic topic;
```

# Les Messages

- Les Messages comportent :
  - Des entêtes standard ;
  - Des propriétés applicatives nommées (optionnelles) ;
  - Un corps dépendant du type de message.
- Il y a 5 sous-types de messages :
  - texte (TextMessage)
  - associations noms-données (MapMessage)
  - flot de données (StreamMessage)
  - flot de données binaires (BytesMessage)
  - objet (ObjectMessage)

# Les Messages

- Les entêtes standard
  - JMSDestination : référence vers la file / le sujet utilisé pour transférer le message
  - JMSReplyTo : référence vers une autre file/sujet que l'on doit utiliser pour répondre au message
  - JMSType : type applicatif du message (String)
  - ... : d'autres entêtes existent mais sont peu utilisées et pas gérées par tous les fournisseurs
- Pour chaque entête on dispose d'une méthode set et d'une méthode get (exemple : setJMSReplyTo)

# Les Messages

- Les propriétés nommées
- Il s'agit de données de types de base (int, float...) ou String qui sont repérées par un nom.
- Pour chaque type on a une méthode set et une méthode get
- exemple :
  - void setIntProperty(String, int)
  - int getIntProperty(String)

# Les Messages

- Il y a un mécanisme de conversion automatique suivant le tableau :

| | boolean | byte | short | int | long | float | double | String |
|---|---|---|---|---|---|---|---|---|
| boolean | X | | | | | | | X |
| byte | | X | X | X | X | | | X |
| short | | | X | X | X | | | X |
| int | | | | X | X | | | X |
| long | | | | | X | | | X |
| float | | | | | | X | X | X |
| double | | | | | | | X | X |
| String | X | X | X | X | X | X | X | X |

# Les Messages

- Lors de la création des consommateurs on peut préciser un filtre utilisant les entêtes et propriétés
- La syntaxe reprend celle de SQL92
- Exemples :
  - “JMSType=’car’ AND color=’red’ AND year < 2008” (où color et year sont des propriétés de type String et int)
  - “JMSType IN (‘car’, ‘motorbike’, ‘van’)”
  - “age BETWEEN 15 AND 19” (age prop de type int)
  - “age >= 15 AND age <= 19”

# Les Messages

- Code :

```
// création d'un consommateur avec sélecteur
mc = session.createConsumer(dest, "JMSType IN ('un', 'deux', 'trois')");
// on ne recevra que des messages d'un de ces 3 types
Message m = mc.receive();
```

# Les Messages

- Les messages textes (TextMessage)
  - Ils permettent de transporter un texte quelconque
  - Utile pour l'interopérabilité : transporter du texte XML
  - méthodes : setText / getText
- Les messages transportants des objets (ObjectMessage)
  - Ils permettent de transporter un objet sérialisé
  - méthodes : setObject / getObject
  - spécifique à Java (interopérablité impossible avec un autre langage)

# Les Messages

- Les messages associatifs (MapMessage)
  - Ils transportent des données de type de base, String ou byte[] (tableaux d'octets non interprétés) associées à des noms (idem propriétés)
  - méthodes : void setInt(String, int) / int getInt(String) ...
- Les messages de type flot (StreamMessage)
  - Ils transportent des données (types de base, String, byte []) placées les unes à la suite des autres
  - méthodes : void writeInt(int) / int readInt()
- Ces 2 types utilisent des conversions similaires au propriétés

# Les Messages

- Les messages de type flot binaire (BytesMessage)
  - Ils transportent des données (types de base, String, byte []) placées les unes à la suite des autres
  - méthodes : void writeInt(int) / int readInt()
  - différence avec précédent : pas de conversion
  - peu utilisé car non interopérable

# Les Messages

- Méthodes utilitaires :
  - clearBody() sert à effacer toutes les données d'un message
  - getPropertyNames() renvoie une énumération des noms des propriétés contenues dans un message
- Remarque : un message reçu est en lecture seule

# Les Messages

- Code (côté production) :

```
// création de l'ObjectMessage et affectation de l'objet à envoyer
ObjectMessage message = session.createObjectMessage();
message.setJMSType("type"); // header
message.setIntProperty("i", 10); // prop1
message.setStringProperty("ch", "bonjour"); // prop2
message.setObject(obj); // corps : obj est sérialisable
// envoie l'objet
mp.send(message);
// autres exemples
TextMessage tmess = session.createTextMessage("Bonjour");
MapMessage mmess = session.createMapMessage();
mmess.setBoolean("bool", true);
mmess.setFloat("PI", 3.1459f);
mmess.setString("ch", "test");
StreamMessage smess = session.createStreamMessage();
smess.writeInt(1);
smess.writeString("test");
mp.send(tmess); mp.send(mmess); mp.send(smess);
```

# Les Messages

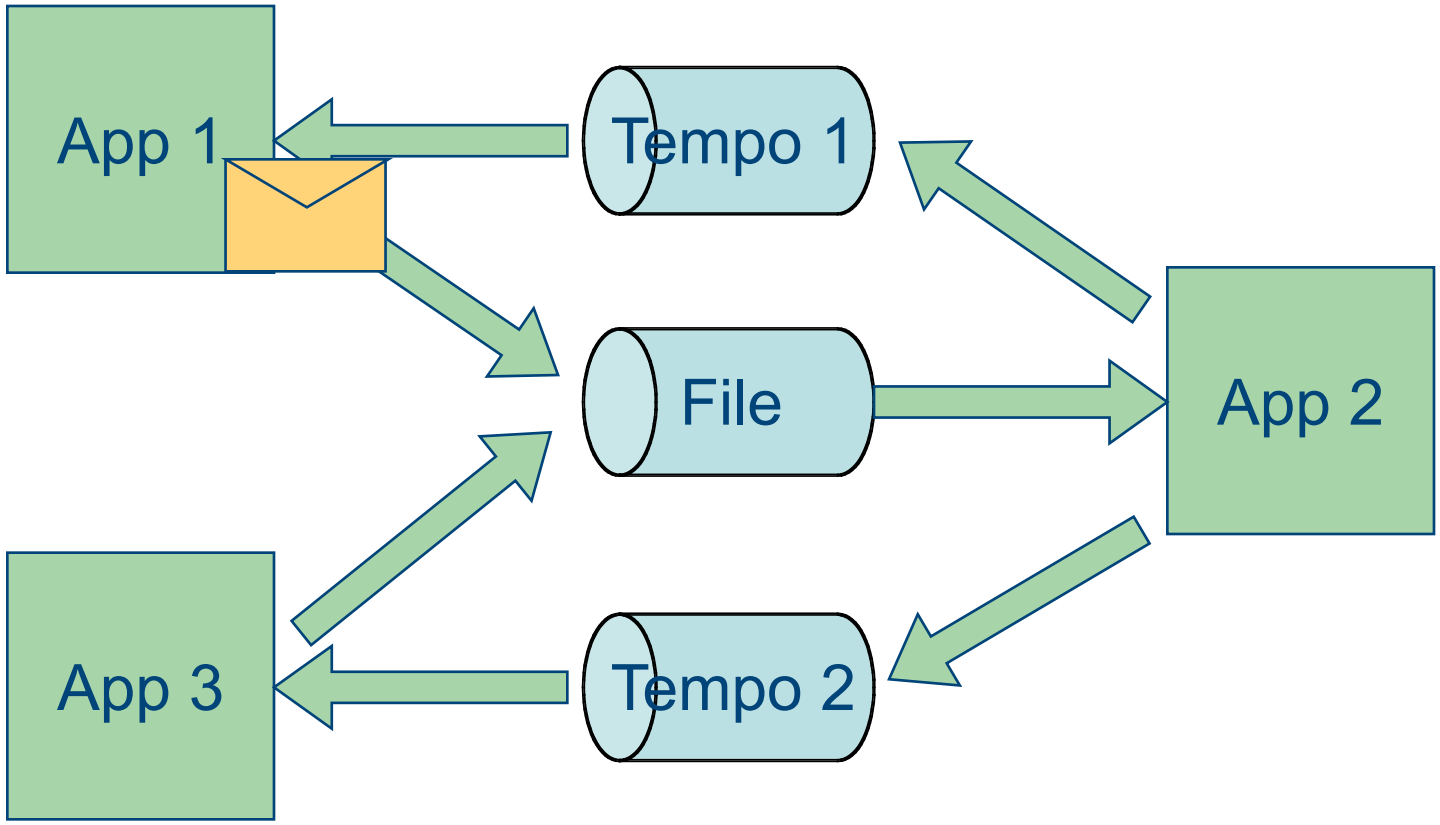
- Code (côté consommation) :

```
// récupération d'un message
Message m = mc.receive();
// on s'assure qu'il est du bon type
if (m instanceof ObjectMessage) {
   ObjectMessage om = (ObjectMessage)m;
   // récupération de l'objet transporté
   Object o = om.getObject();
   // entêtes et propriétés
   String type = om.getJMSType();
   int i = om.getIntProperty("i");
   String ch = om.getStringProperty("ch");
}
// autres exemples
TextMessage tmess = (TextMessage) mc.receive(100); // attente pendant 100 ms
if (mmess != null) System.out.println(tmess.getText());
MapMessage mmess = (MapMessage) mc.receiveNoWait(); // pas d'attente
if (mmess != null) {
      boolean b = mmess.getBoolean("bool");
      String ch = mmess.getString("ch");
      float f = mmess.getFloat("PI");
}
```

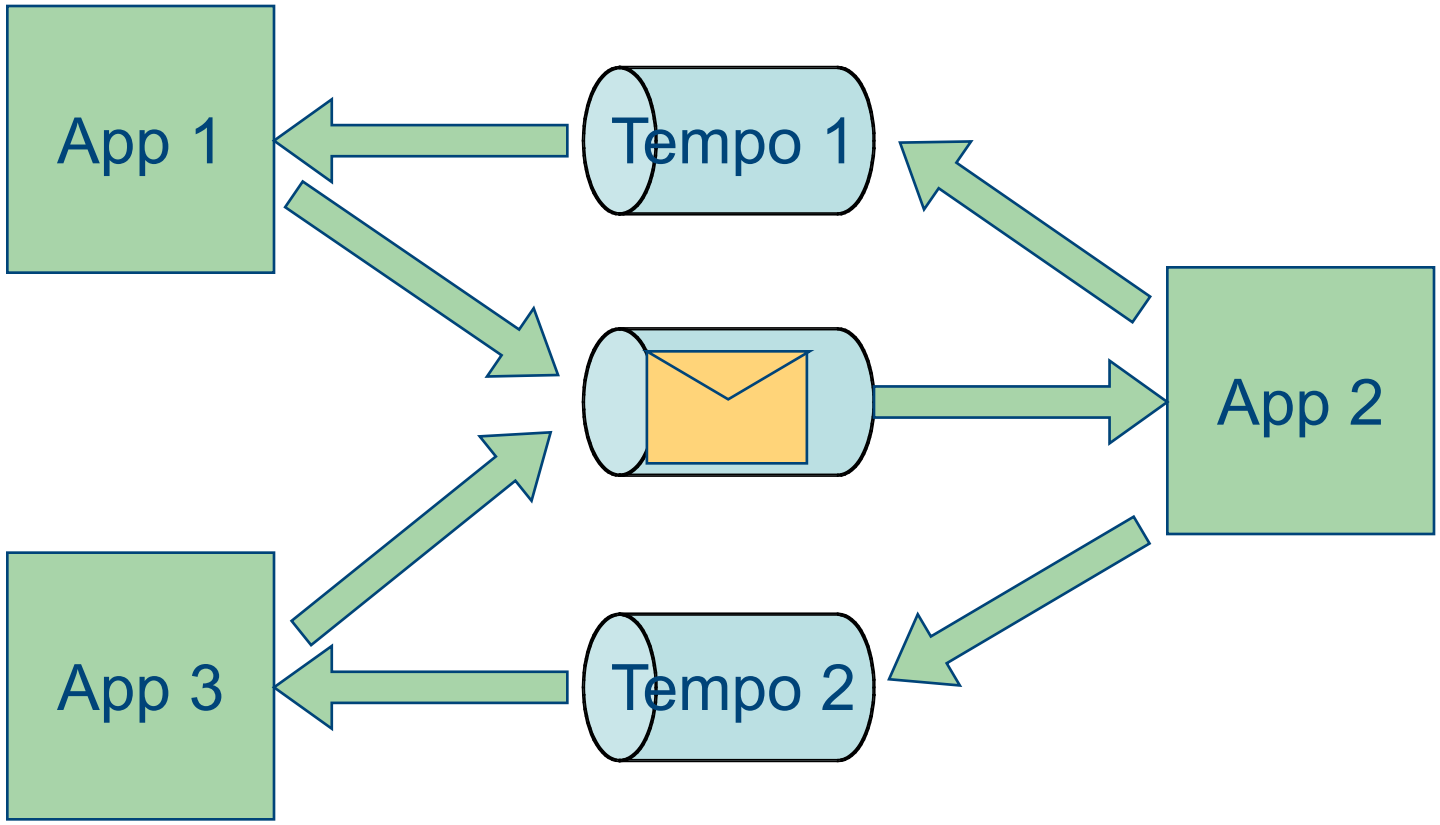
# Mécanisme de Requête/Réponse

- Outils :
  - Entête JMSReplyTo
  - Files temporaires (créées avec la méthode createTemporaryQueue de Session)
- Fonctionnement :
  - L'application 1, crée une file temporaire, crée un message requête, y indique la file temporaire en JMSreplyTo et produit le message ;
  - L'application 2, récupère le message, le traite, crée la réponse, utilise la session pour créer un producteur sur la file temporaire (getJMSReplyTo) et utilise ce producteur pour envoyer la réponse.

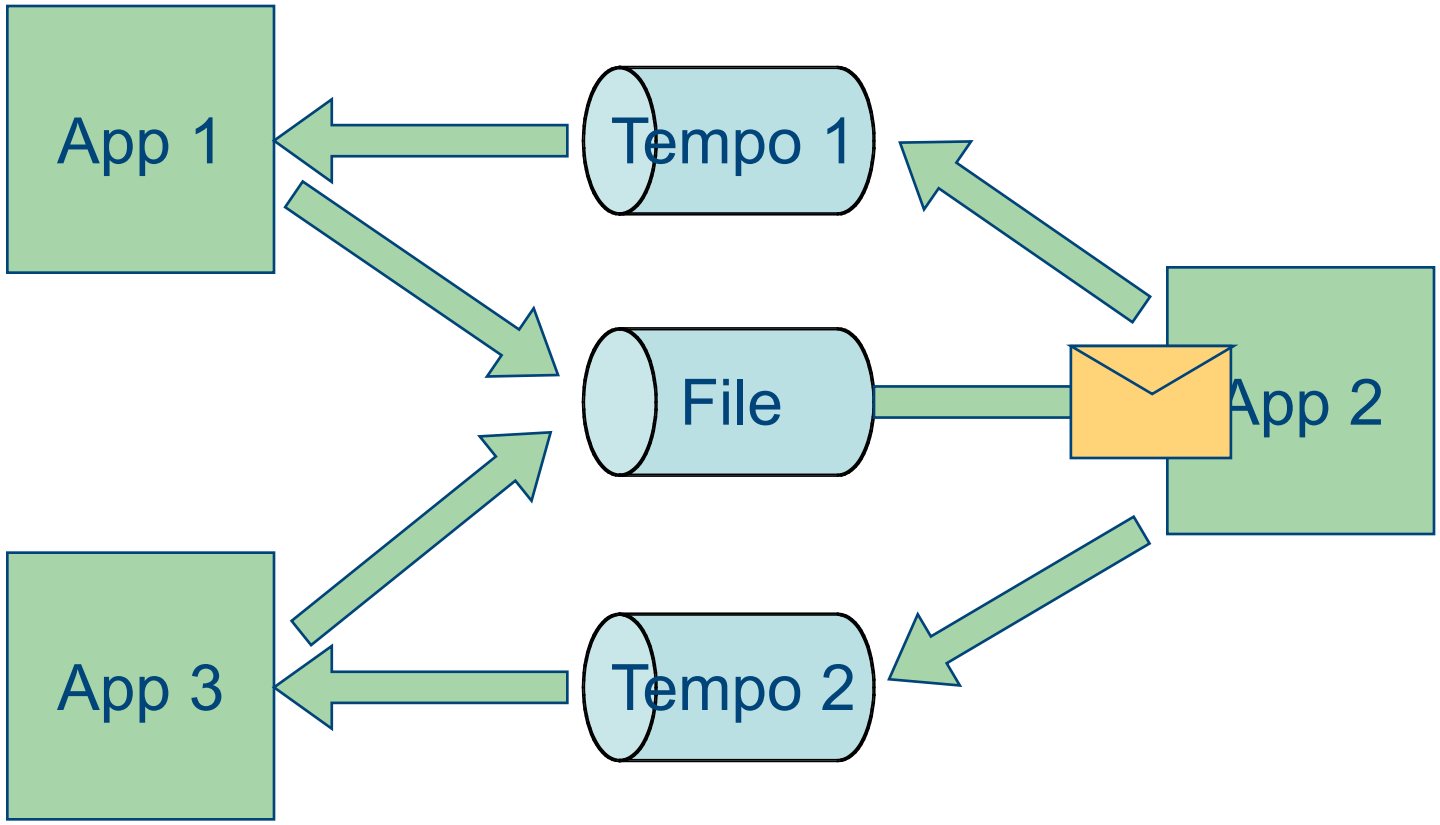
# Mécanisme de Requête/Réponse



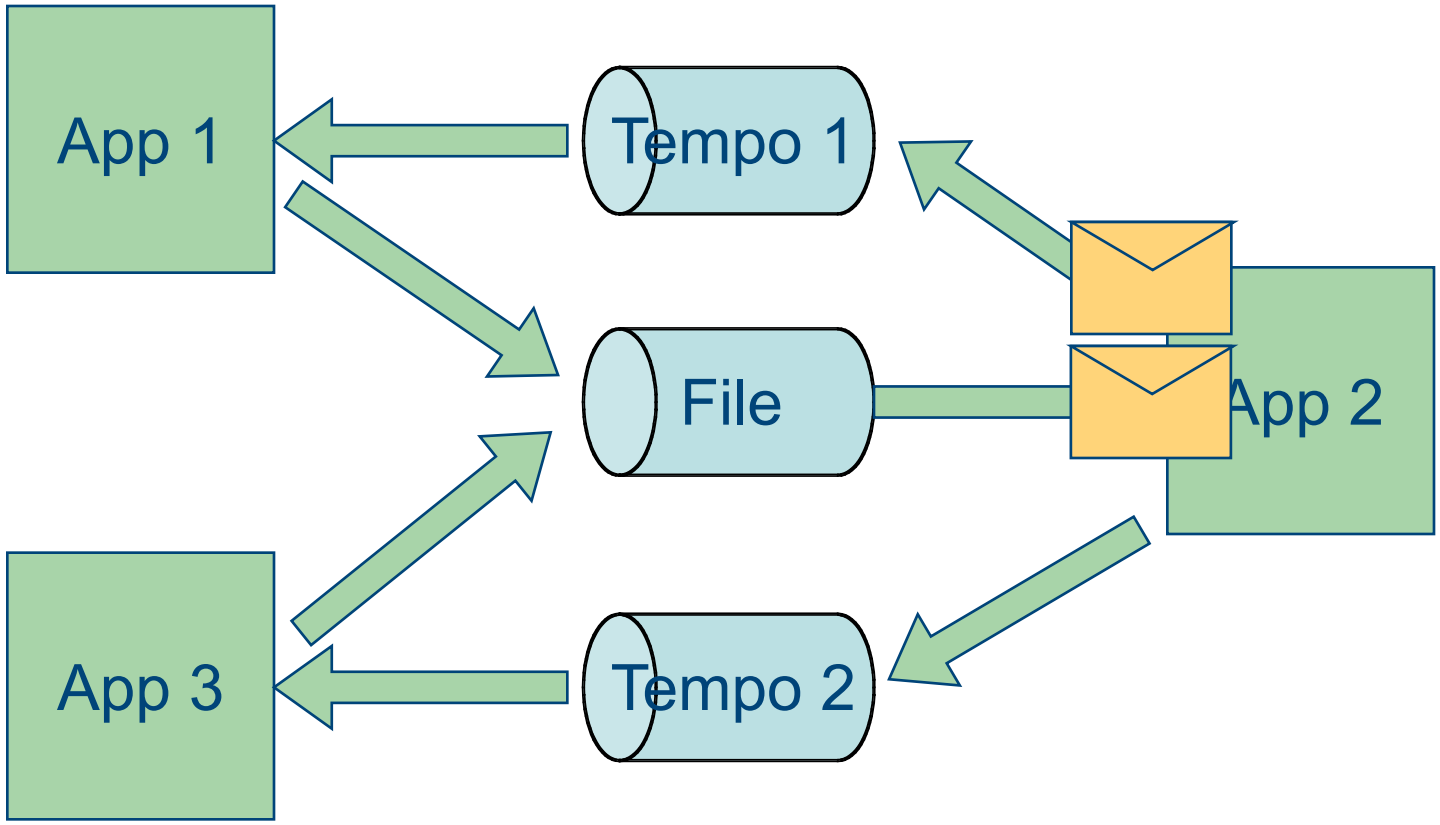
# Mécanisme de Requête/Réponse



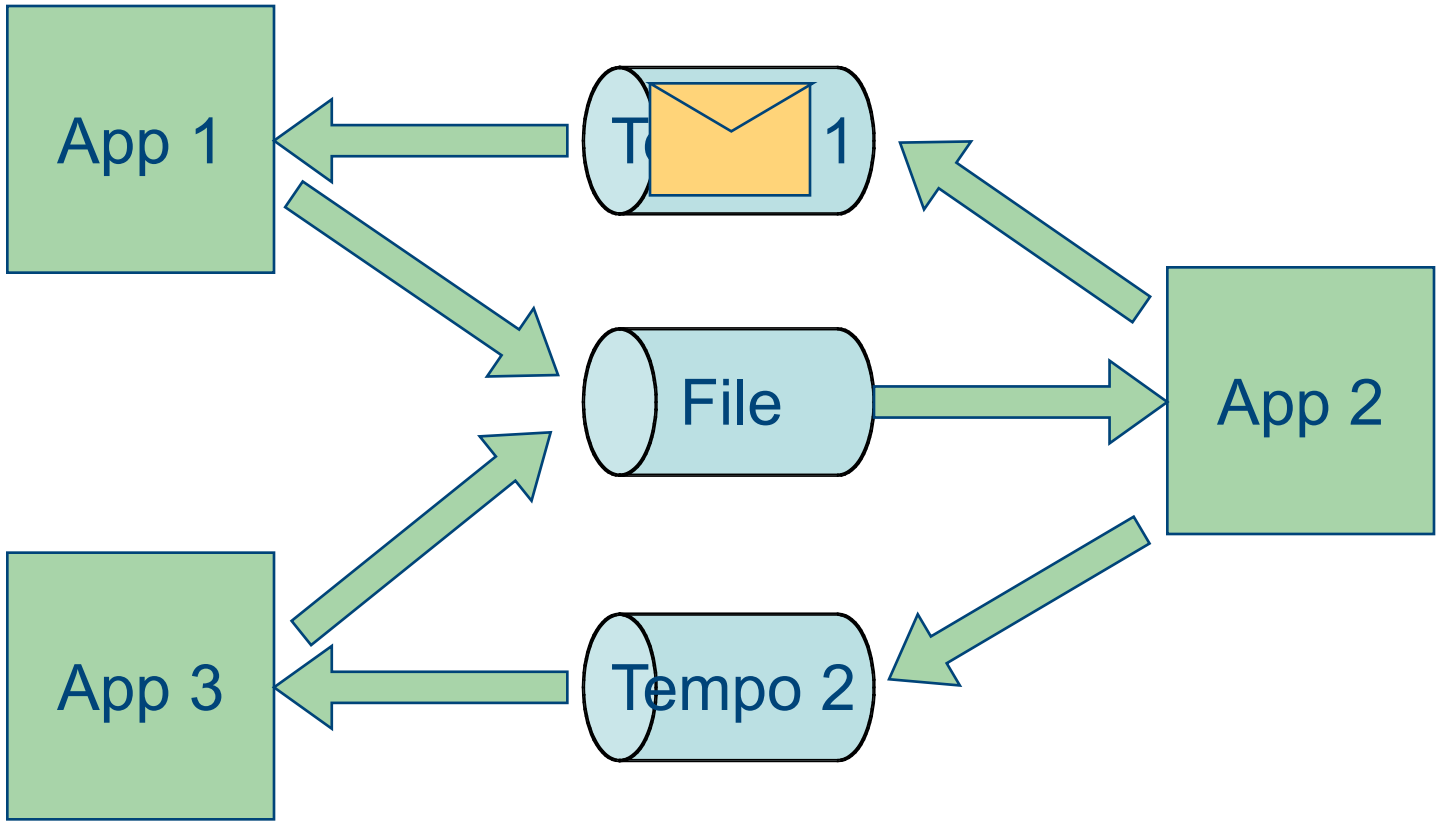
# Mécanisme de Requête/Réponse



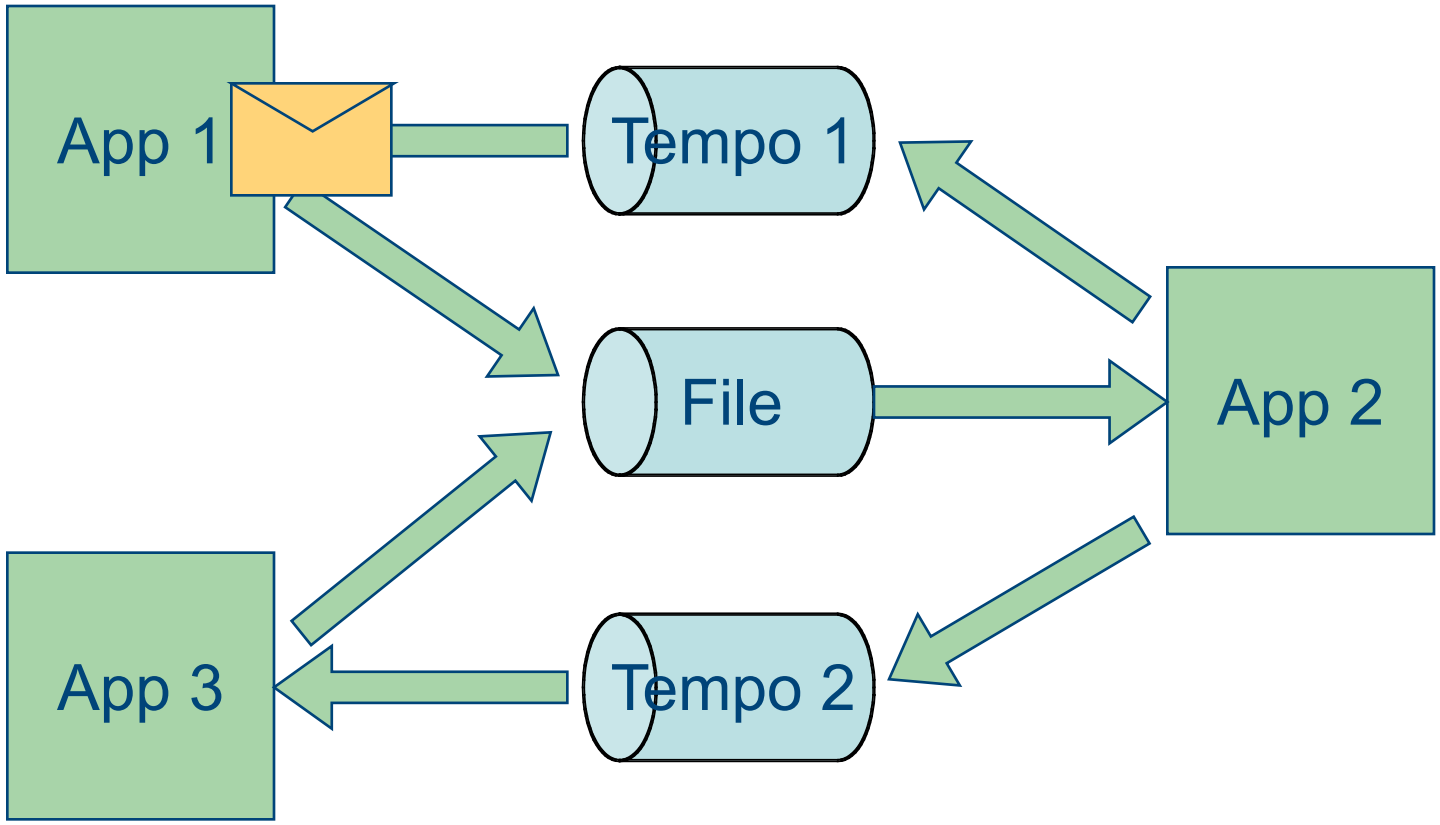
# Mécanisme de Requête/Réponse



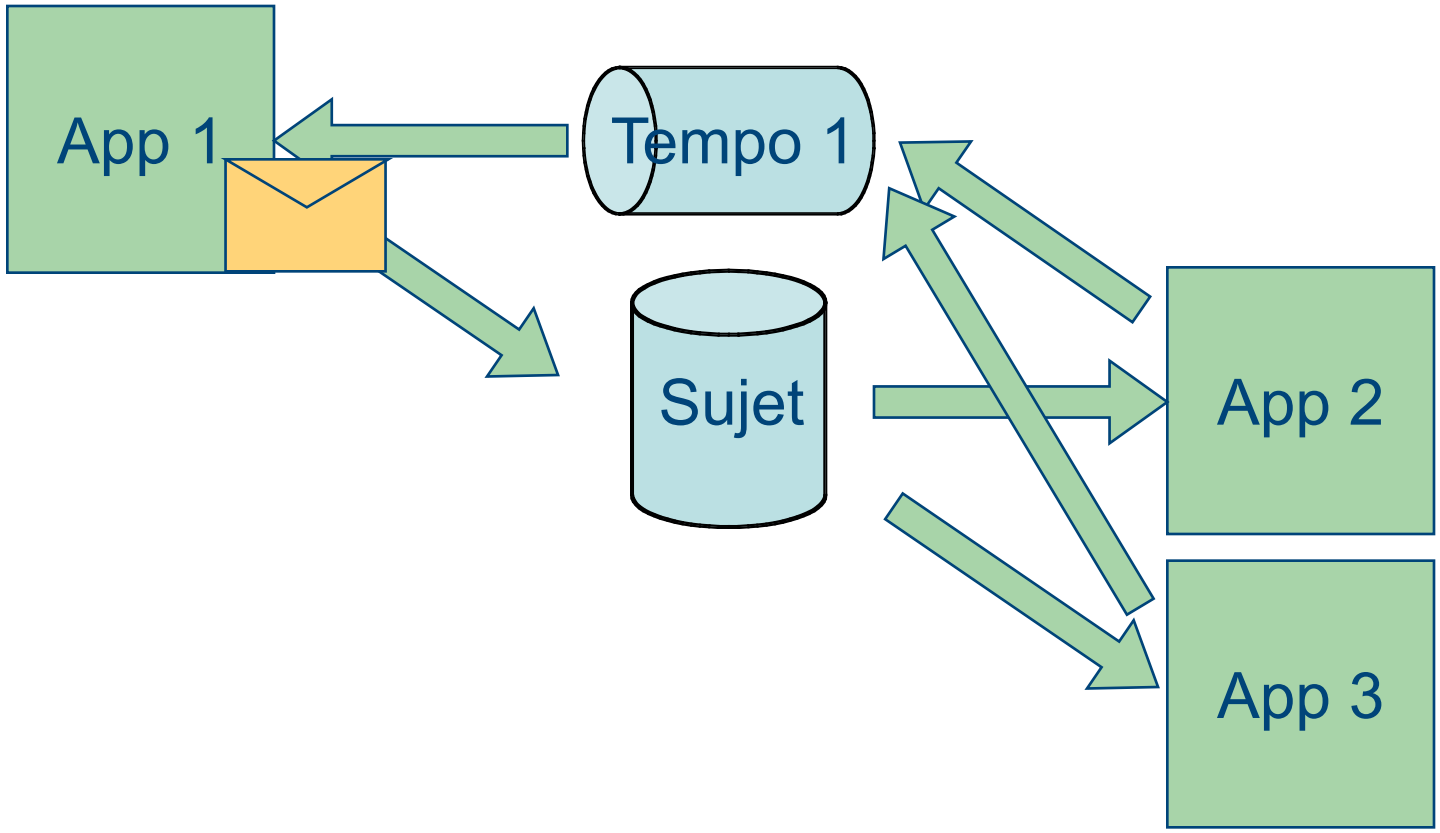
# Mécanisme de Requête/Réponse



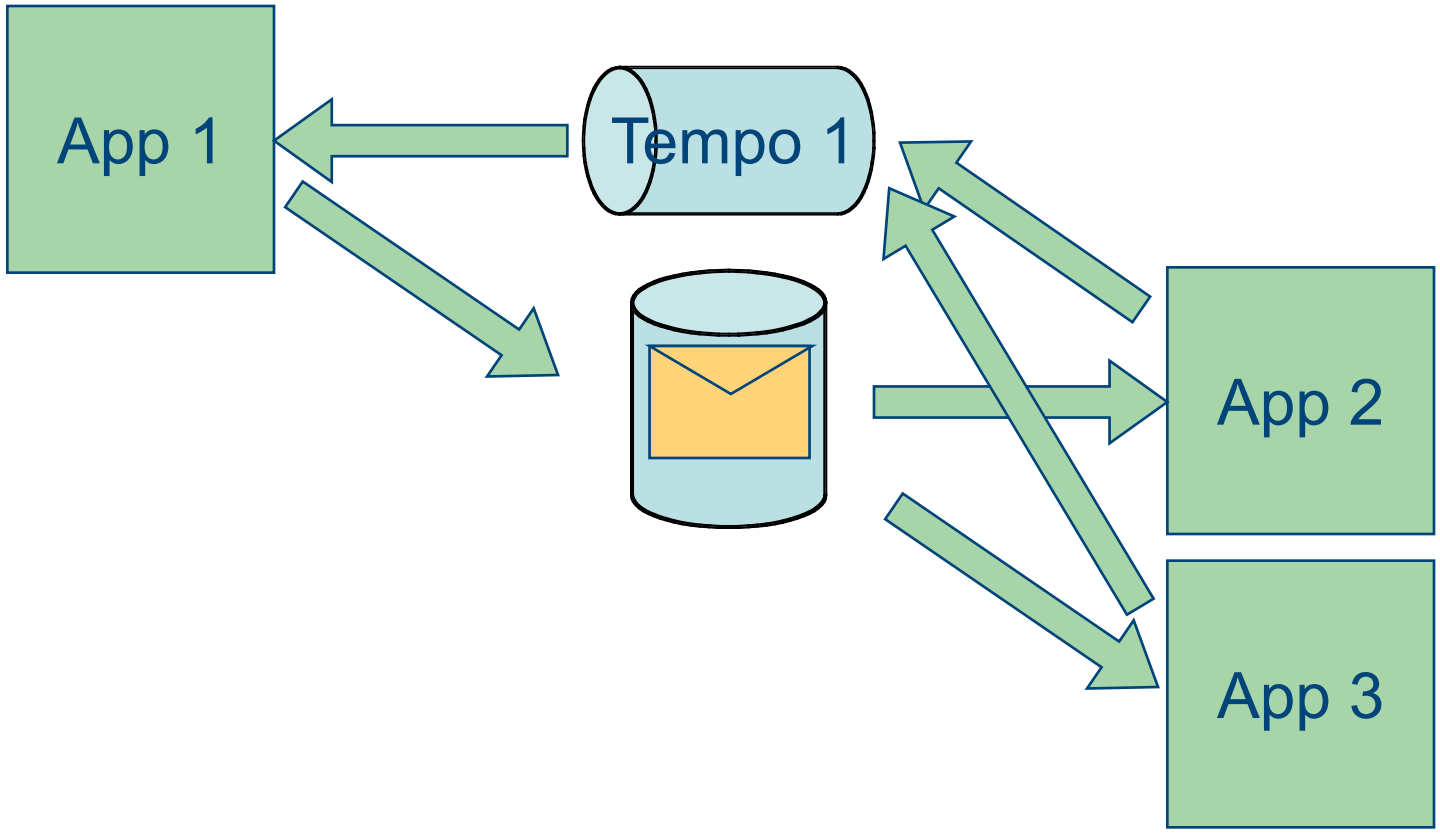
# Mécanisme de Requête/Réponse



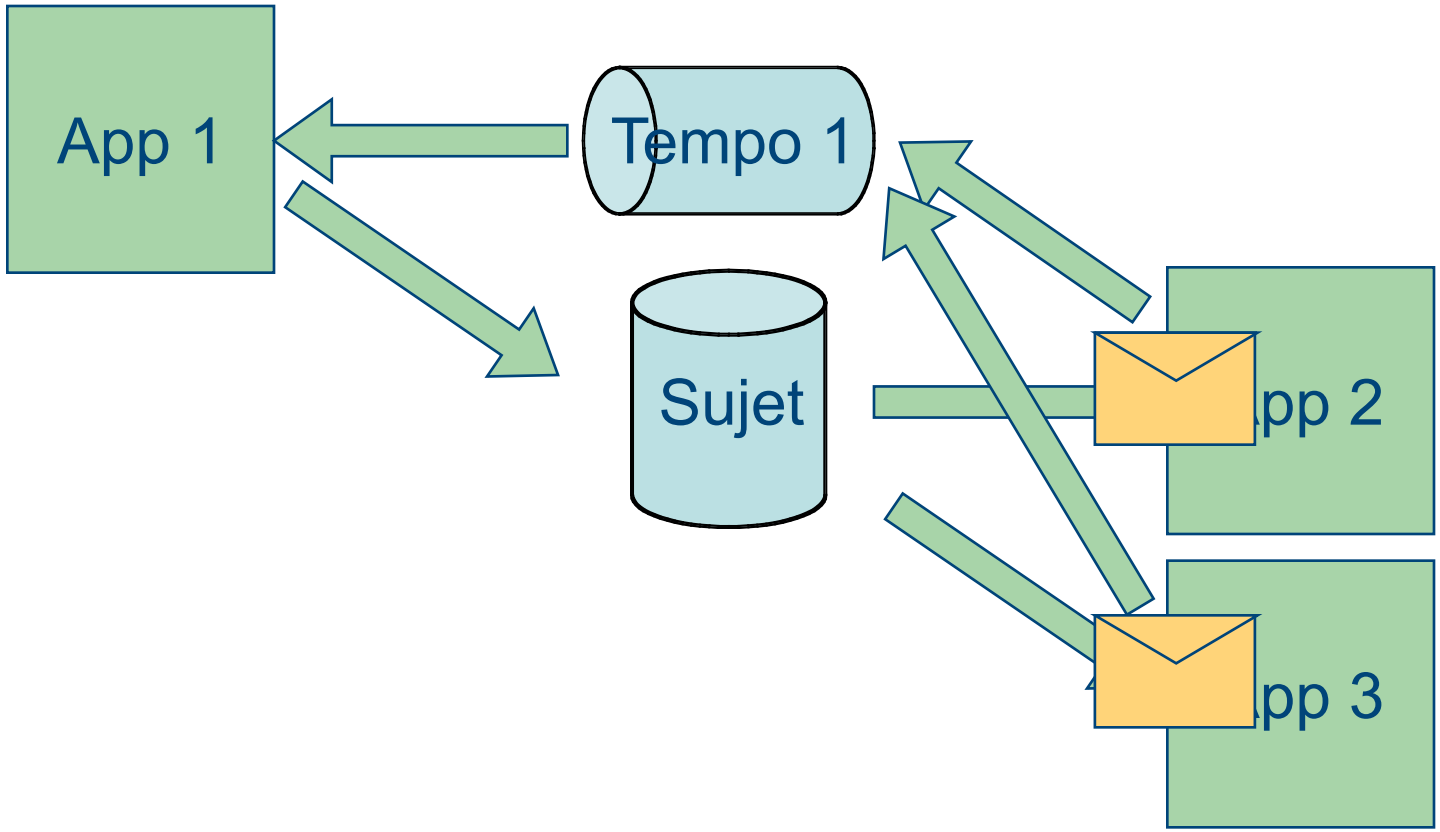
# Mécanisme de Requête/Réponse



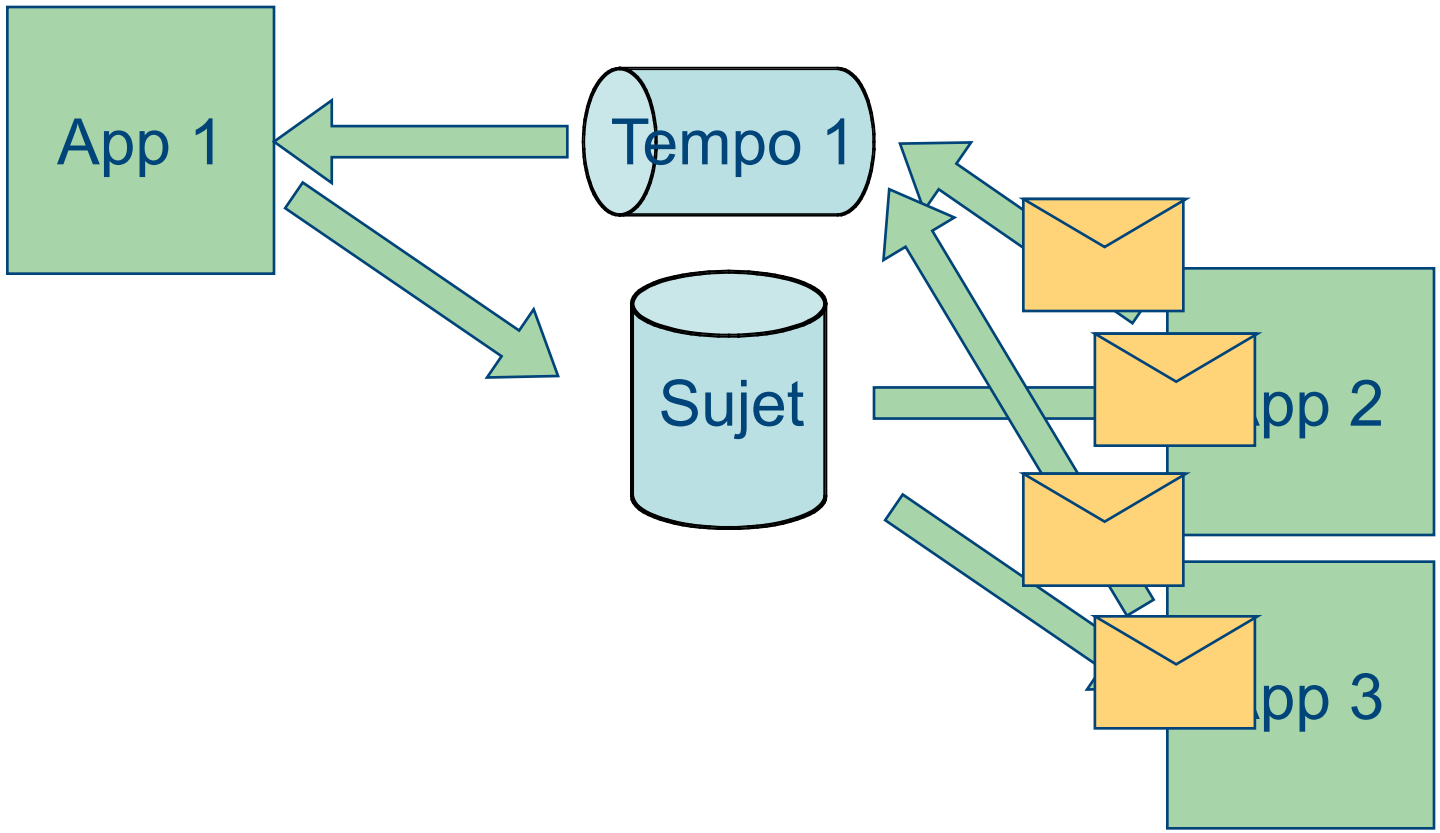
# Mécanisme de Requête/Réponse



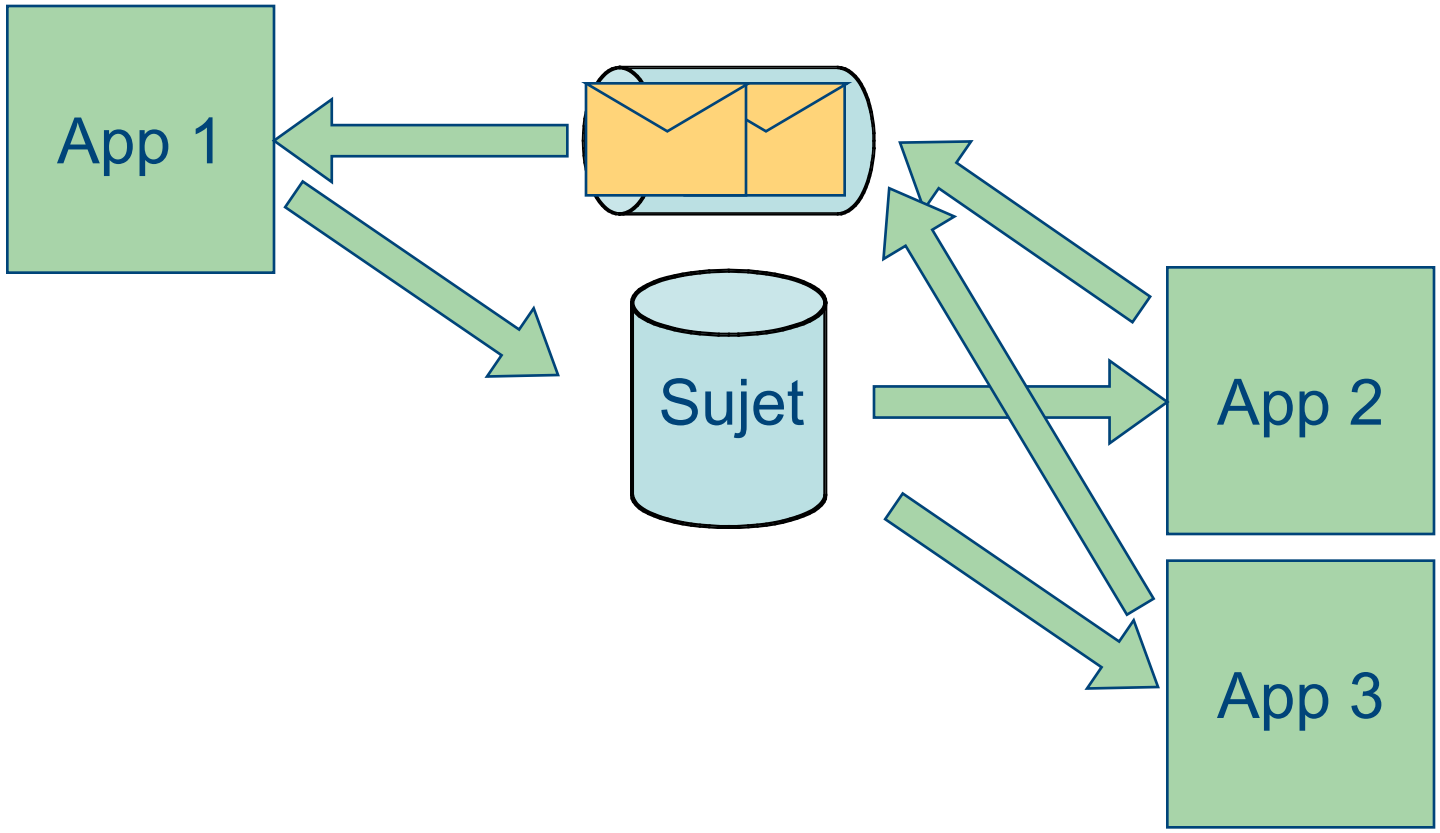
# Mécanisme de Requête/Réponse



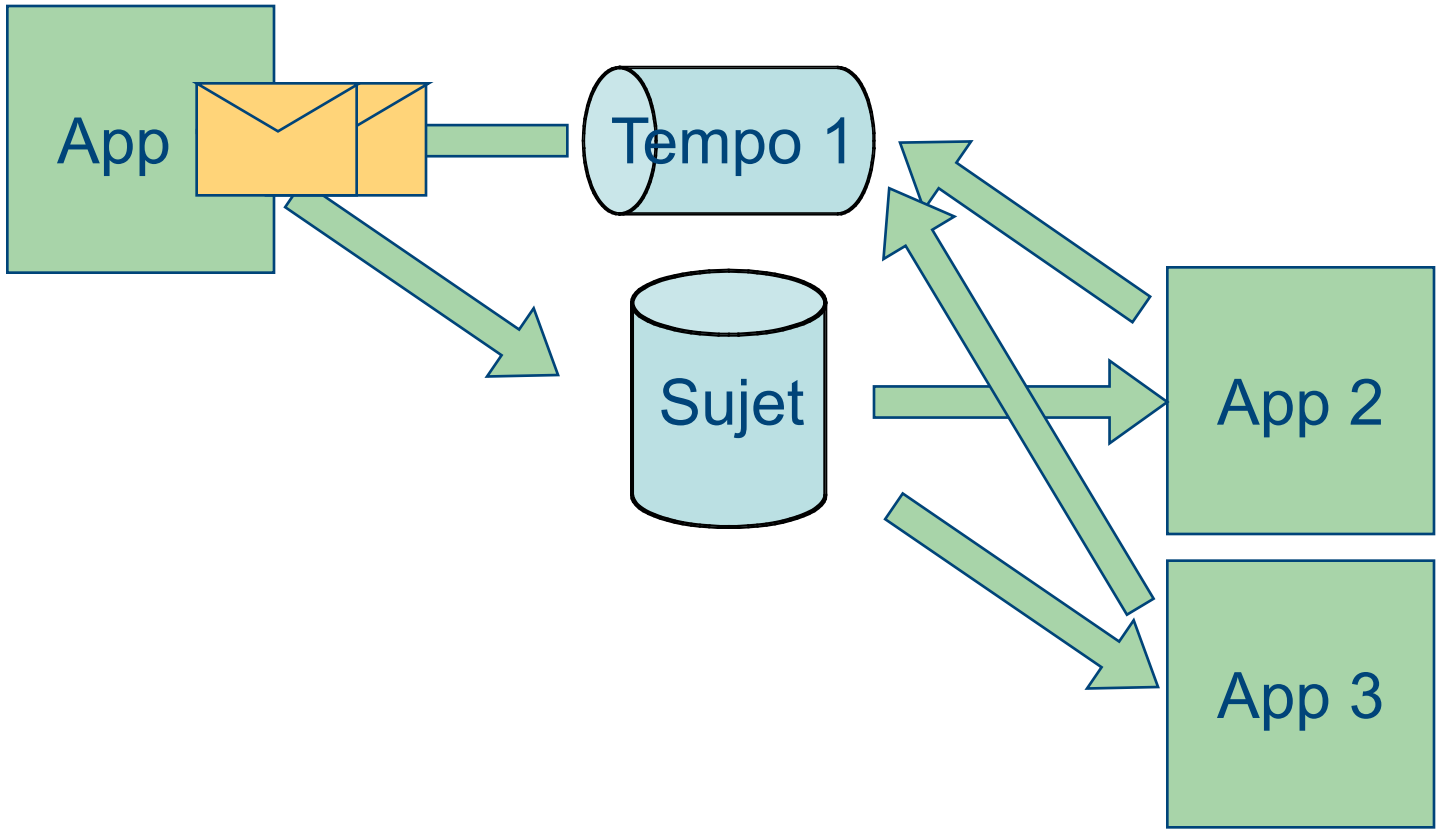
# Mécanisme de Requête/Réponse



# Mécanisme de Requête/Réponse



# Mécanisme de Requête/Réponse



# Gestion des transactions

- 2 types de transactions :
  - Locales, gérées par la session, ne concernent que JMS
  - Distribuées, gérées par JTA, peuvent concerner : JMS, JDBC, EJB...
  - NB : Dans les EJB on est obligé d'utiliser JTA
- Pour utiliser les transactions locales on créera une session ainsi :
  - Session session = connection.createSession(true, 0);
- Ensuite on utilisera :
  - session.commit();
  - session.rollback();

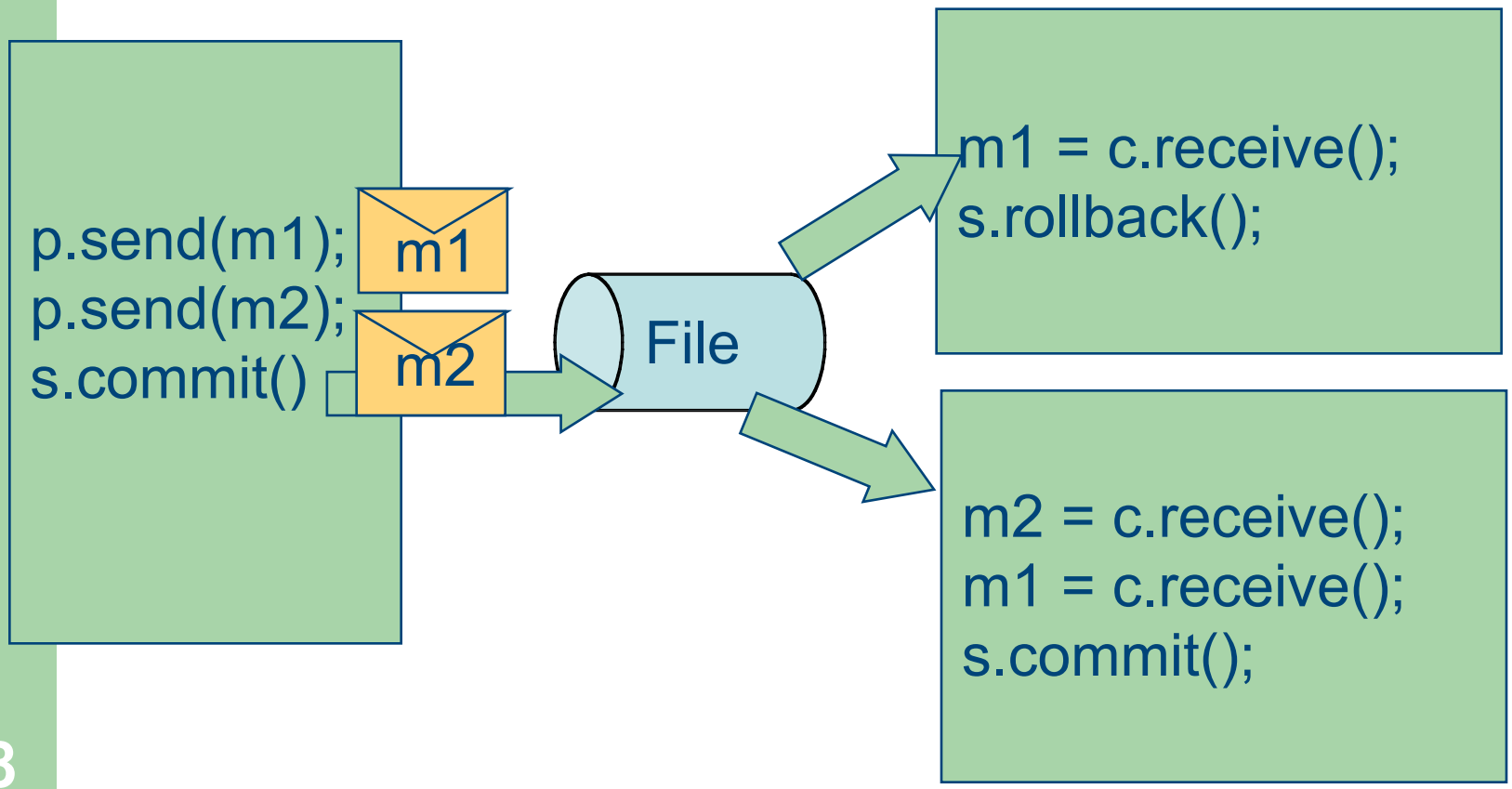
# Gestion des transactions

- Dès que la session est créée une transaction commence
- Après chaque commit ou rollback une nouvelle transaction commence
- Si on ferme la session/connexion sans avoir fait de commit toutes les opérations sont annulées !

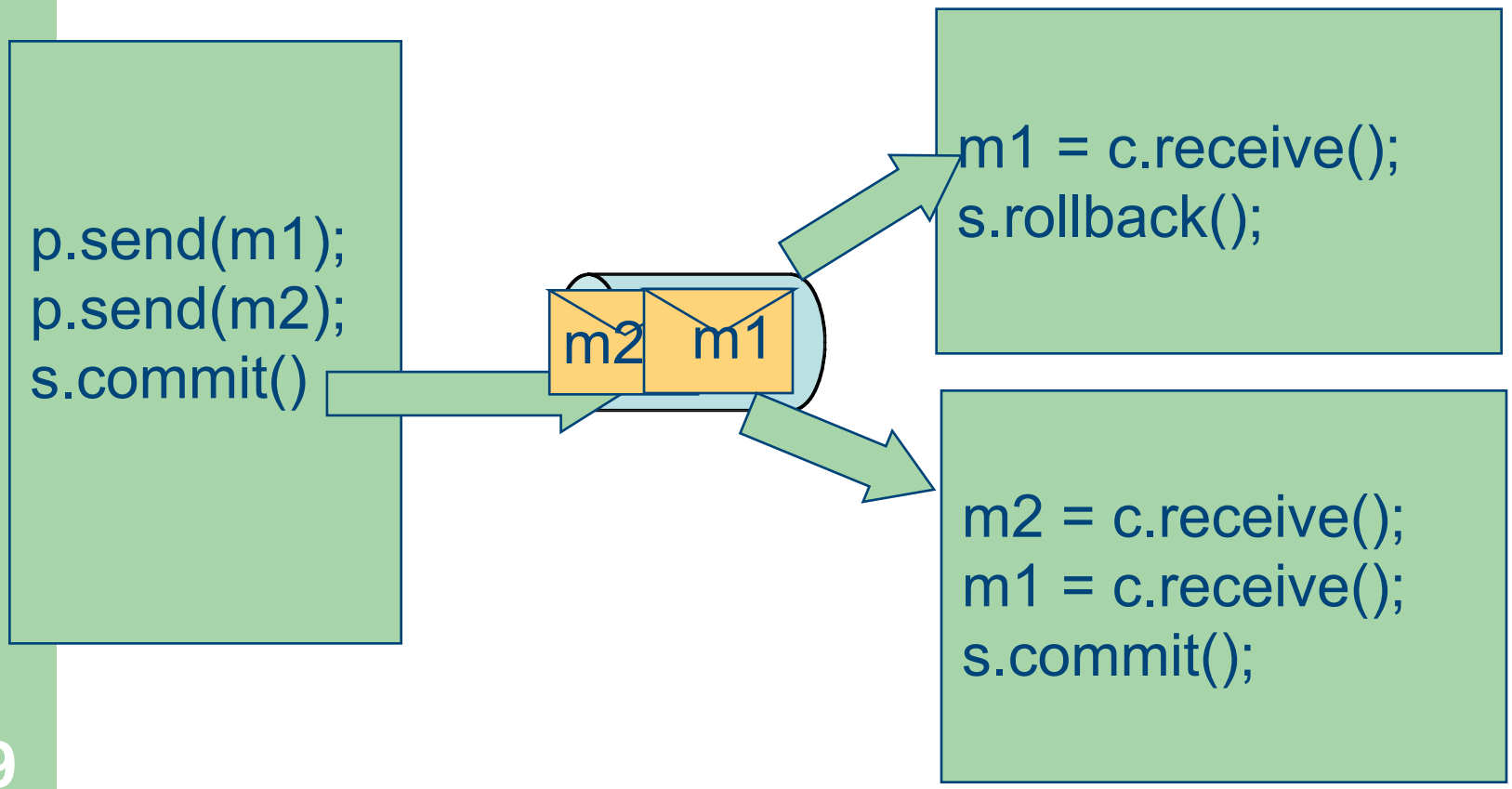
# Gestion des transactions

- Lorsqu'on fait un commit, les messages produits sont envoyés et les messages reçus sont acquittés
- Lorsqu'on fait un rollback, les messages produits sont détruits et les messages reçus sont récupérés par le fournisseur et livrés à nouveau lors de prochains receive (sauf si ils ont expirés)
- On peut faire des send et receive dans la même transaction mais il faut qu'ils soient indépendants
- Si un receive correspond au traitement d'un message envoyé ça va bloquer puisque le message n'est envoyé que lors du commit

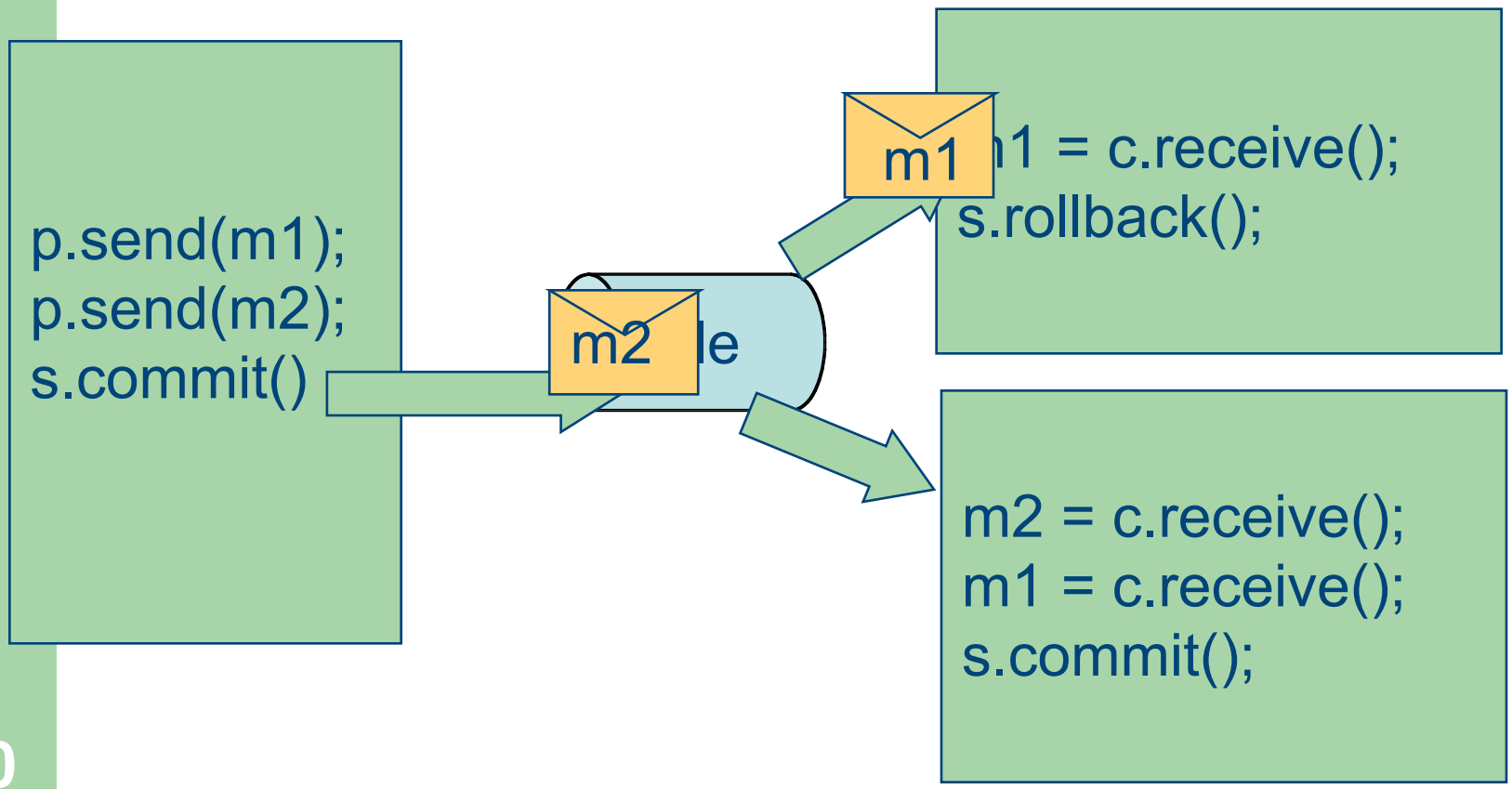
# Gestion des transactions



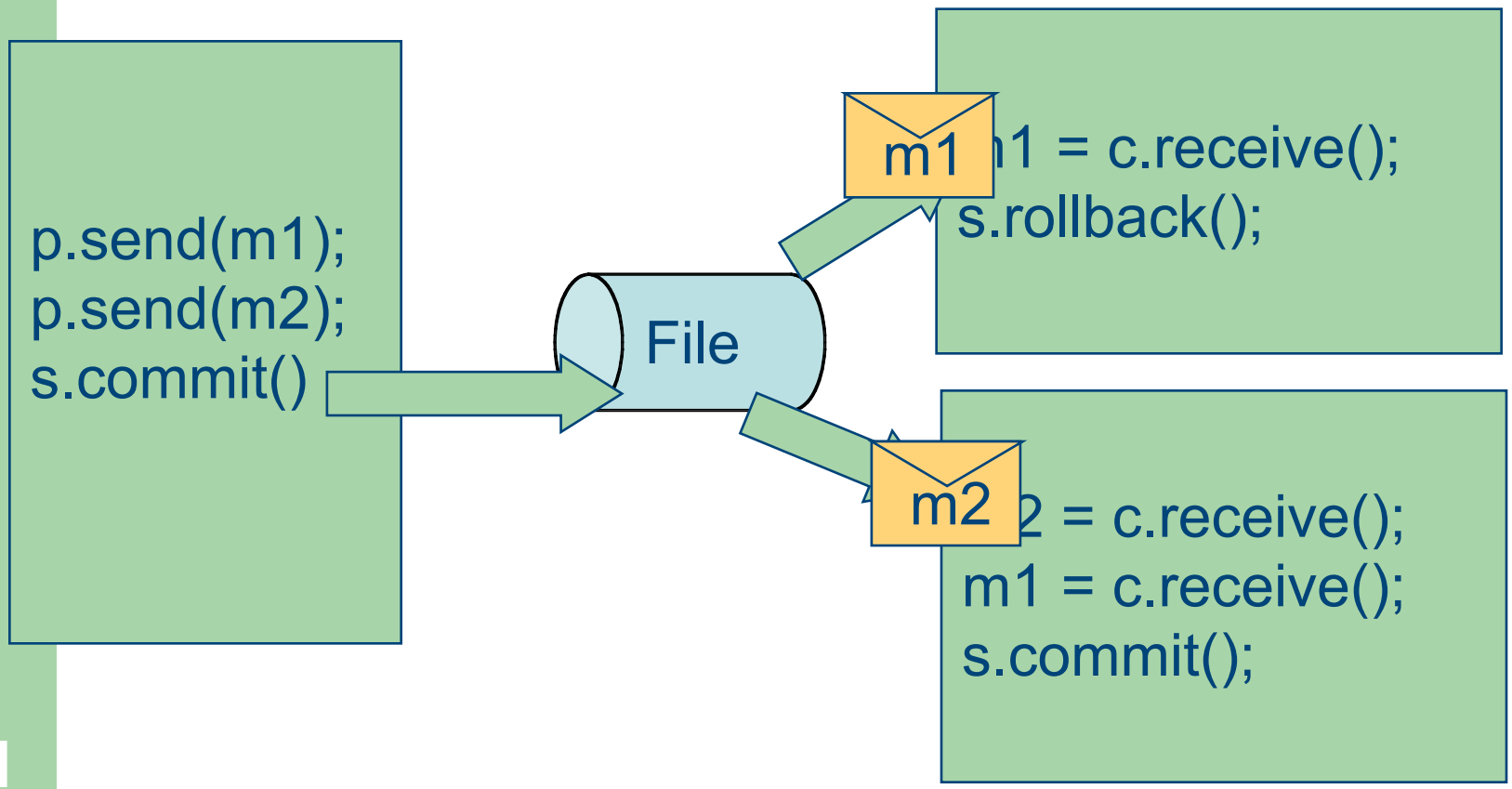
# Gestion des transactions



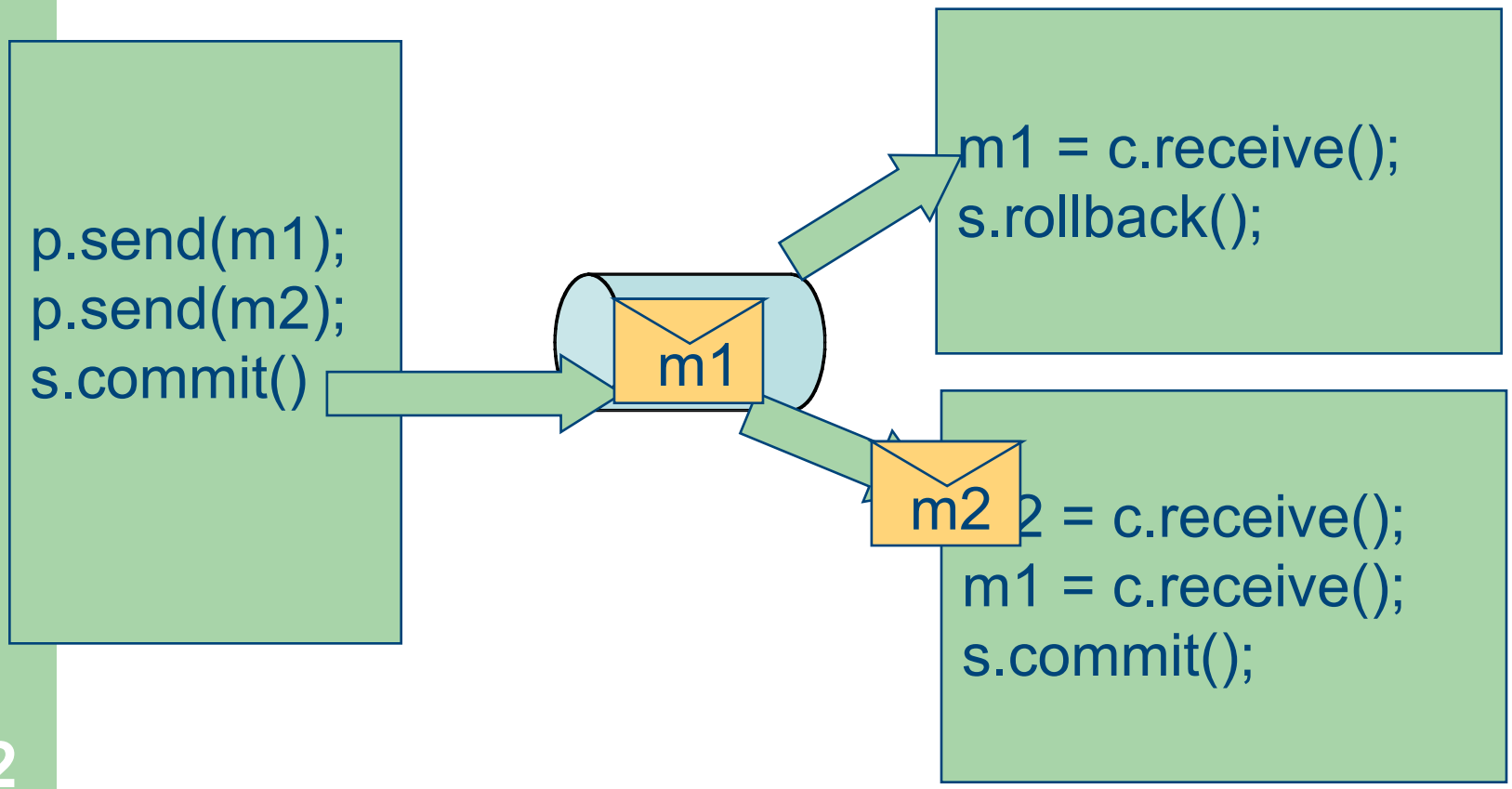
# Gestion des transactions



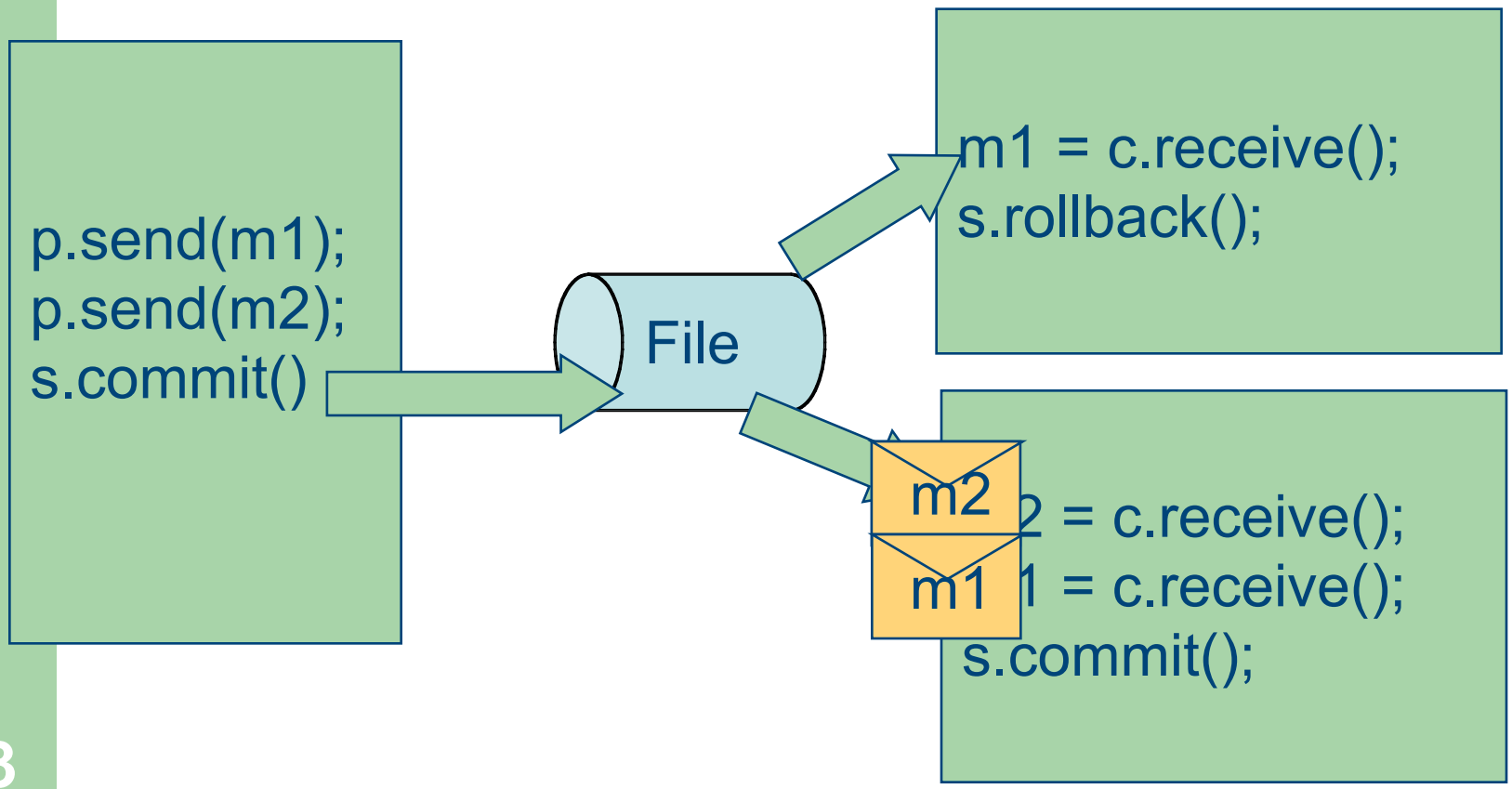
# Gestion des transactions



# Gestion des transactions



# Gestion des transactions



# Fiabilité de JMS

- Pour gérer la fiabilité des applications utilisant JMS on peut :
  - contrôler l'acquittement des messages
  - rendre les messages persistants
  - définir des priorités sur les messages
  - préciser une date d'expiration pour un message

# Fiabilité de JMS

- Contrôler l'acquittement des messages
  - Normalement on doit faire la chose suivante
    - Recevoir un message
    - Le traiter
    - Acquitter le message
  - Cependant, si on choisi le mode automatique d'acquittement des messages et qu'on utilise l'opération receive l'acquittement se fera lors de la réception
  - Donc si le traitement conduit à un crash de l'application le message sera perdu

# Fiabilité de JMS

- Contrôler l'acquittement des messages
  - Solution : utiliser l'acquittement manuel des messages
  - Lors de la création de la session :
    - Session s = c.createSession(false, Session.CLIENT_ACKNOWLEDGE);
  - Lors de la réception d'un message :
    - Message m = c.receive(); // réception du message
    - ... // traitement du message
    - m.acknowledge(); // acquittement du message

# Fiabilité de JMS

- NB : avec les message listener on peut toujours utiliser Session.AUTO_ACKNOWLEDGE puisque l'acquittement auto se fait après l'appel de la méthode onMessage
- Il existe aussi une option Session.DUPS_OK_ACKNOWLEDGE qui indique que l'acquittement peut se faire plus tard pour améliorer l'efficacité de la communication client/ fournisseur => mais on peut avoir des traitements dupliqués si le fournisseur plante

# Fiabilité de JMS

- Si des messages d'une file ou d'un sujet avec abonné persistant n'ont pas été acquittés quand la session est fermée ils seront livrés à nouveau
- On peut vérifier cela en testant le champ d'entête JMSRedelivered qui vaut vrai si il y a déjà eu livraison
- On peut utiliser la méthode recover de Session pour redémarrer la livraison au premier message non acquitté

# Fiabilité de JMS

- Rendre les messages persistants
  - JMS permet 2 modes de livraison :
    - PERSISTENT (par défaut) signifie que le fournisseur doit stocker le message sur disque et ainsi en cas de crash fournisseur il ne sera pas perdu
    - NON_PERSISTENT signifie que le fournisseur n'a pas à stocker le message sur disque et donc qu'il peut être perdu
  - Le second mode améliore les performances mais diminue la fiabilité

# Fiabilité de JMS

- Rendre les messages persistants
  - Choix sur le producteur (pour tous les messages envoyés per celui-ci) :
    - producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
  - Choix message par message
    - producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
    - NB : les 3èmes et 4èmes paramètres sont la priorité et l'expiration

# Fiabilité de JMS

- Définir des priorités sur les messages
  - On peut choisir une priorité parmi 10 (de 0 à 9) pour indiquer des messages prioritaires ou non
  - Les messages de priorité supérieure seront livrés avant ceux de priorité inférieure
  - Par défaut les messages ont la priorité 4
  - Ici aussi on peut le faire pour tous les messages envoyés par un producteur :
    - producer.setPriority(7);
  - ou en utilisant la version longue de send

# Fiabilité de JMS

- Préciser une date d'expiration pour un message
  - Par défaut les messages n'expirent jamais
  - On peut choisir une durée de vie maximale (en milisecondes), 0 indique qu'ils n'expirent jamais.
  - Ici aussi on peut le faire pour tous les messages d'un producteur :
    - producer.setTimeToLive(60000); // expiration après 1 minute
  - ou avec la version longue de send

# Exemple complet

- Producteur (ici avec gestion de transaction) :

```
public static void main(String[] args) {
      Context context = null;
      ConnectionFactory factory = null;
      Connection connection = null;
      String factoryName = "ConnectionFactory";
      String destName = null;
      Destination dest = null;
      int count = 1;
      Session session = null;
      MessageProducer sender = null;
      String text = "Message ";

      if (args.length < 1 || args.length > 2) {
          System.out.println("usage: Sender <destination> [count]");
          System.exit(1);
      }

      destName = args[0];
      if (args.length == 2) {
          count = Integer.parseInt(args[1]);
      }
```

# Exemple complet

```
try {
    // création du contexte JNDI.
    context = new InitialContext();

    // recherche de la ConnectionFactory
    factory = (ConnectionFactory) context.lookup(factoryName);
    // recherche de la Destination
    dest = (Destination) context.lookup(destName);

    // création de la connexion
    connection = factory.createConnection();

    // création de la session avec transaction
    session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

    // création du producteur
    sender = session.createProducer(dest);

    // démarrage de la connexion (optionnel ici)
    connection.start();
```

# Exemple complet

```
    try {
          for (int i = 0; i < count; ++i) {
              TextMessage message = session.createTextMessage();
              message.setText(text + (i + 1));
              sender.send(message);
              System.out.println("Sent: " + message.getText());
          }
          session.commit();
    } catch(Exception e) {
          session.rollback();
    }
} catch (JMSException exception) {
    exception.printStackTrace();
} catch (NamingException exception) {
    exception.printStackTrace();
```

# Exemple complet

```
        } finally {
            // fermeture du contexte
            if (context != null) {
                try {
                    context.close();
                } catch (NamingException exception) {
                    exception.printStackTrace();
                }
            }

            // fermeture de la connexion
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException exception) {
                    exception.printStackTrace();
                }
            }
        }
    }
```

# Exemple complet

- Consommateur (ici asynchrone avec callback) :

```
// classe contenant le callback
public class SampleListener implements MessageListener {

     // à chaque réception cette méthode est appelée automatiquement
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage text = (TextMessage) message;
            try {
                System.out.println("Received: " + text.getText());
            } catch (JMSException exception) {
                System.err.println("Failed to get message text: " + exception);
            }
        }
    }
}
```

# Exemple complet

- Consommateur (ici asynchrone avec callback) :

```
public static void main(String[] args) {
     Context context = null;
     ConnectionFactory factory = null;
     Connection connection = null;
     String factoryName = "ConnectionFactory";
     String destName = null;
     Destination dest = null;
     Session session = null;
     MessageConsumer receiver = null;
     BufferedReader waiter = null;

     if (args.length != 1) {
         System.out.println("usage: Listener <destination>");
         System.exit(1);
     }

     destName = args[0];
```

# Exemple complet

```
try {
    // crée le contexte JNDI
    context = new InitialContext();

    // cherche la ConnectionFactory
    factory = (ConnectionFactory) context.lookup(factoryName);

    // cherche la Desination
    dest = (Destination) context.lookup(destName);

    // création de la connexion
    connection = factory.createConnection();

    // création de la session
    session = connection.createSession(
        false, Session.AUTO_ACKNOWLEDGE);

    // création du récepteur
    receiver = session.createConsumer(dest);

    // abonne un “listener”
    receiver.setMessageListener(new SampleListener());
```

# Exemple complet

```
        // démarre la connexion pour commencer la réception asynchrone des messages
        connection.start();

        System.out.println("Waiting for messages...");
        System.out.println("Press [return] to quit");

        waiter = new BufferedReader(new InputStreamReader(System.in));
        waiter.readLine();
    } catch (IOException exception) {
        exception.printStackTrace();
    } catch (JMSException exception) {
        exception.printStackTrace();
    } catch (NamingException exception) {
        exception.printStackTrace();
```

# Exemple complet

```
    } finally {
        // ferme le contexte
        if (context != null) {
            try {
                context.close();
            } catch (NamingException exception) {
                exception.printStackTrace();
            }
        }

        // ferme la connexion
        if (connection != null) {
            try {
                connection.close();
            } catch (JMSException exception) {
                exception.printStackTrace();
            }
        }
    }
}
```

# Remerciements et Questions ?

- Ce cours est basé sur :
  - Le tutoriel sur Java EE d'Oracle: http://download.oracle.com/javaee/6/tutorial/doc/index.html
  - Le cours sur les MOM et JMS de Didier Donsez : http://www-adele.imag.fr/users/Didier.Donsez/cours/
  - Le livre Java Messaging d'Eric Bruno
  - La page JMS de Wikipedia : http://en.wikipedia.org/wiki/Java_Message_Service