

# MRI-COURS 6: MOM

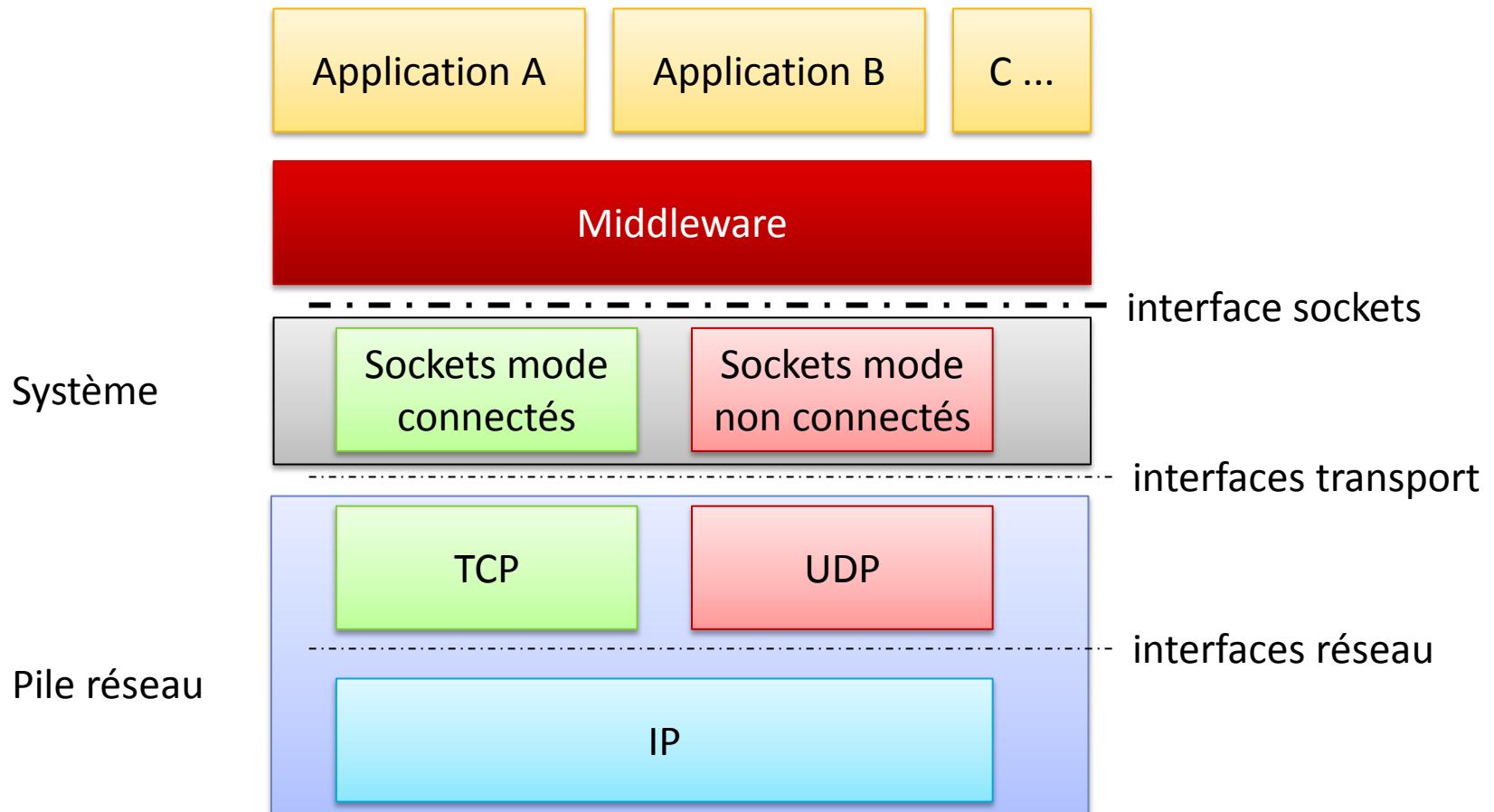
yoann.maurel@irisa.fr



# Motivations

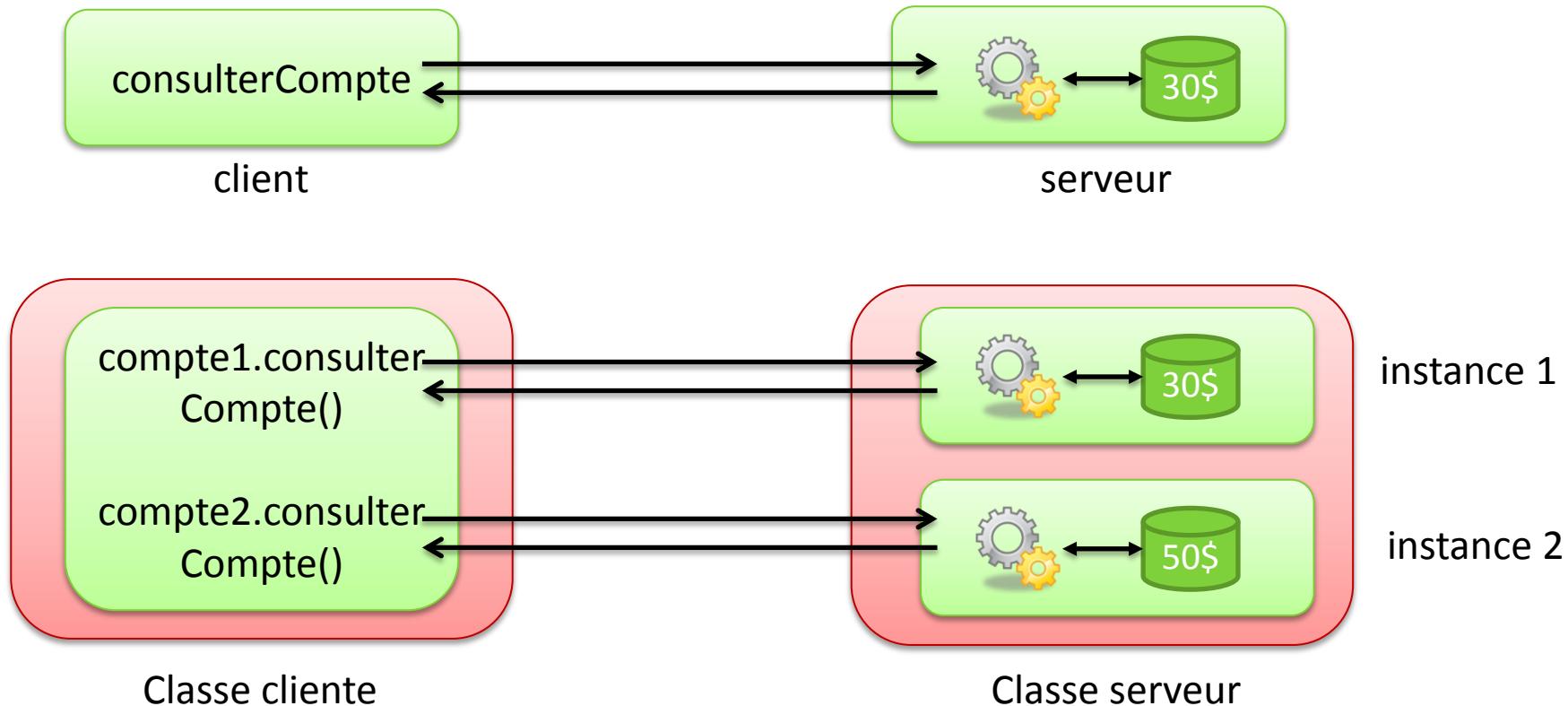
# Notion de Middleware

- ▶ On ajoute une couche entre le système et les applications



# Les appels de méthodes sur des objets distants

- ▶ Mis en œuvre par CORBA et RMI par exemple
- ▶ Extension du concept des RPC aux objets



# Défauts des RPC

---

## ▶ Dépendance temporelle:

- ▶ le client et le serveur doivent être disponibles au même moment pour pouvoir fonctionner correctement
- ▶ généralement synchrone : le client doit attendre la fin du traitement du serveur pour pouvoir continuer son travail.

## ▶ Couplage relativement fort

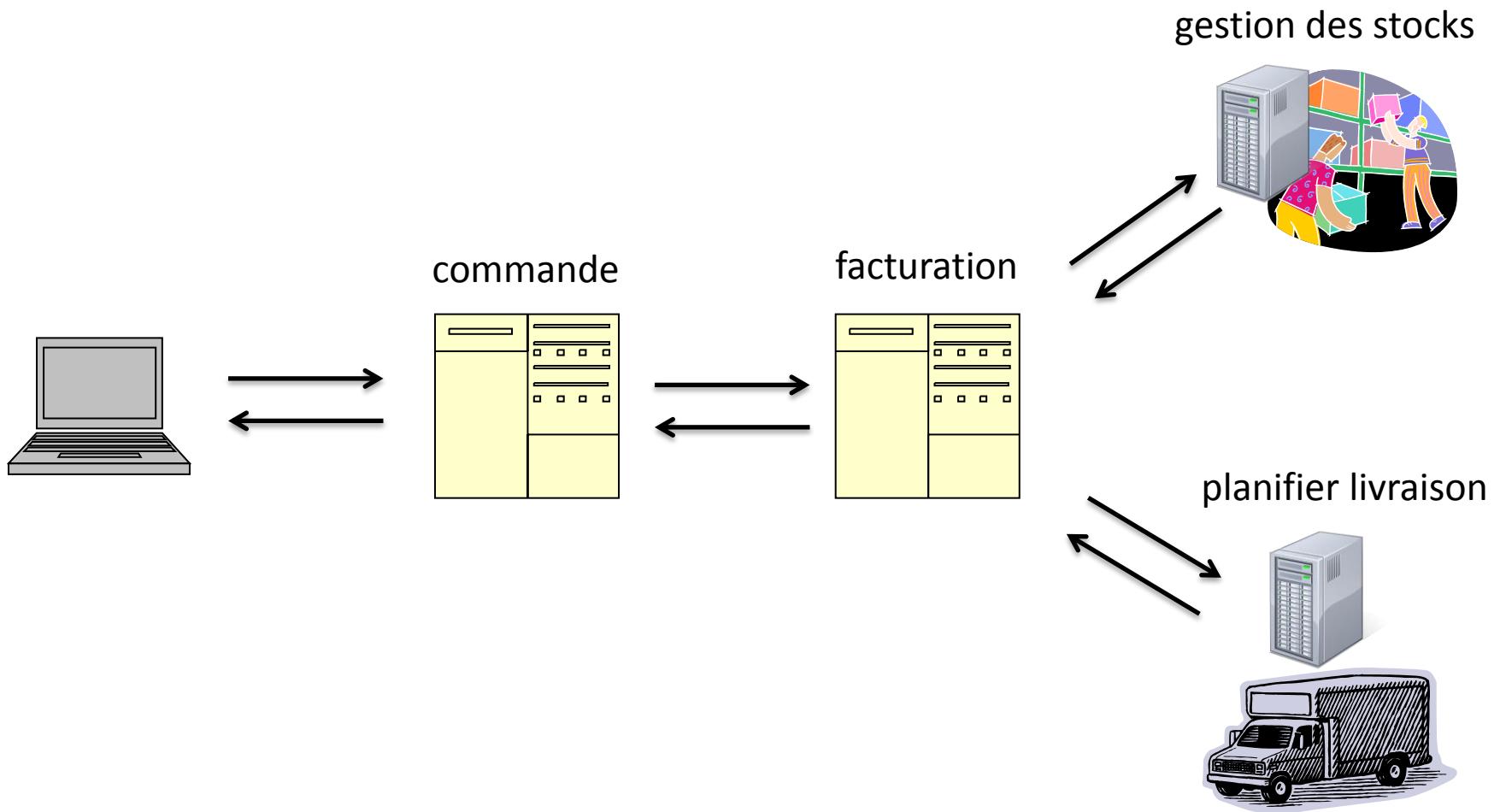
- ▶ Les deux parties doivent se mettre d'accord sur une interface pour communiquer.
- ▶ Les deux parties doivent se connaître ou connaître l'adresse d'un annuaire.

# Rappel synchrone/asynchrone

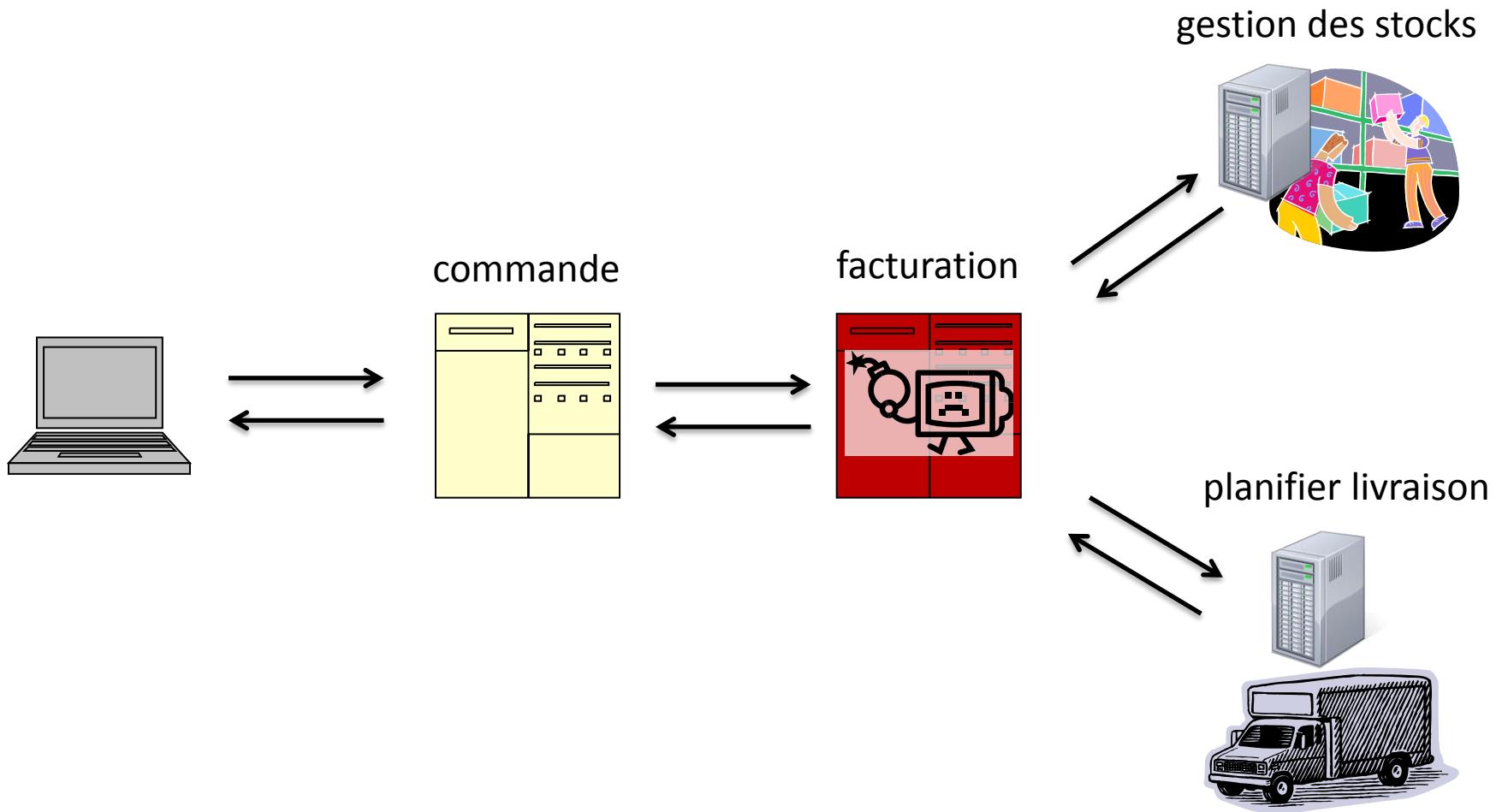
---

- ▶ Mode synchrone :
  - ▶ L'émetteur reste bloqué jusqu'à ce que le destinataire réponde
- ▶ Mode asynchrone :
  - ▶ L'émetteur n'est pas bloqué et soit :
    - est prévenu de l'arrivée d'une réponse (push)
    - vérifie à intervalle de temps régulier la disponibilité d'une réponse (pull)

# Mode synchrone et RPC

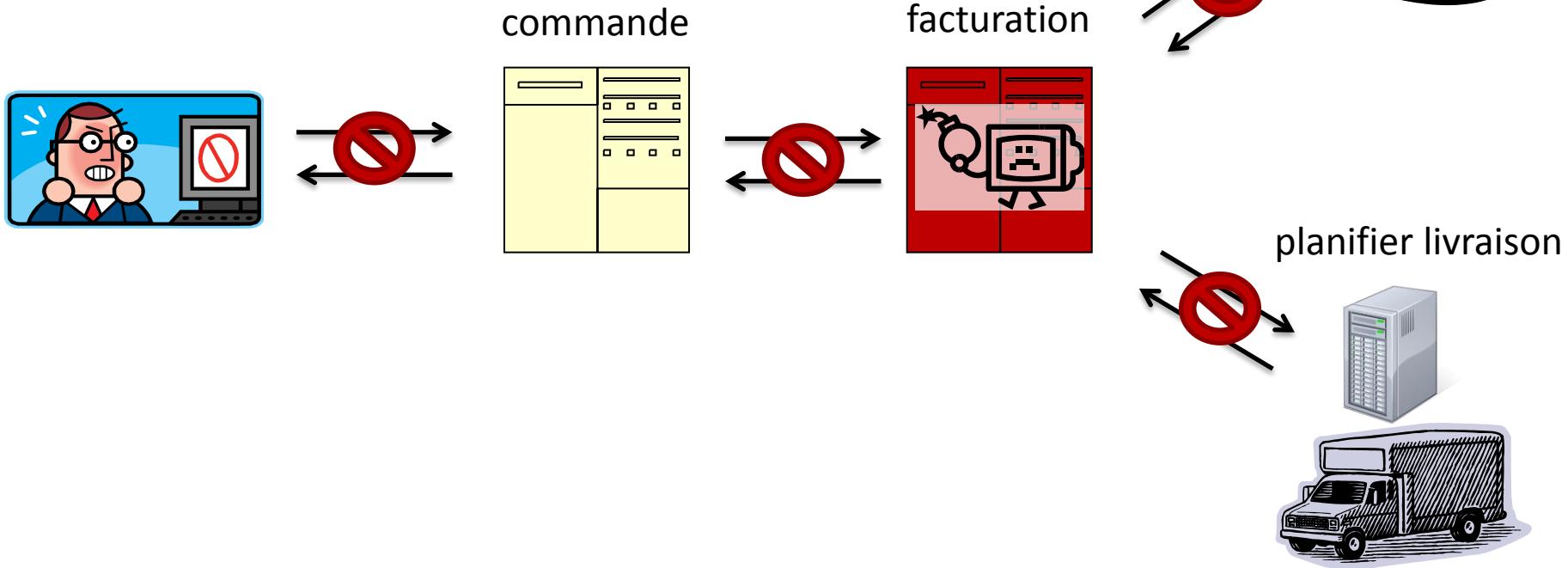


# Mode synchrone et RPC



# Mode synchrone et RPC

- tous les messages sont affectés
- l'activité est suspendue partout
- la réinitialisation peut être complexe



# Les contraintes des applications à grande échelle

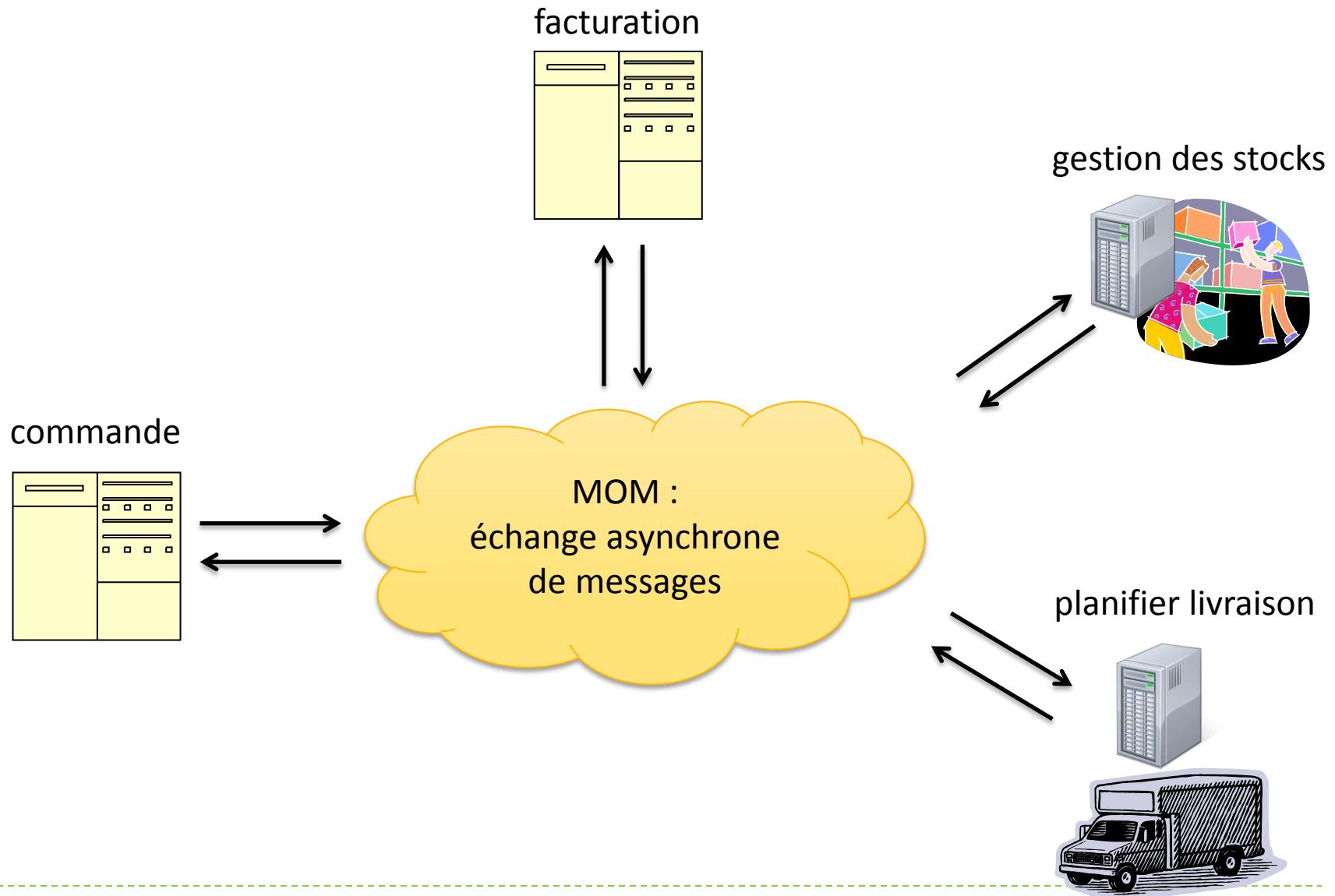
---

- ▶ Fortement distribuées et pas toujours disponibles au même moment
  - ▶ grand nombre de commandes à traiter mais capacité ponctuellement inadaptée (ex facturation ne peut pas traiter le flot des commandes)
  - ▶ panne d'un composant
  - ▶ maintenance
- ▶ Le mode synchrone bloquant n'est pas adapté
  - ▶ certains traitements prennent plus de temps que d'autres (ex, vérification facturation, planification livraison)
    - ▶ le client ne veut pas attendre 3h que toutes les étapes soient validées pour avoir confirmation
  - ▶ les composants doivent pouvoir continuer à faire des traitements sans attendre les autres
- ▶ Les couplages sont contraignants :
  - ▶ les composants ne devraient pas avoir à se connaître pour communiquer
    - ▶ en RPC on doit connaître l'IP des clients
    - ▶ en RMI, on doit connaître le nom des objets
  - ▶ les composants doivent se mettre d'accord sur les interfaces



# Message Oriented Middleware

# Idée, mode asynchrone :



# Applications utilisatrices

---

- ▶ Diffusion de messages :
  - ▶ news, météo, la bourse
- ▶ Messageries interbancaires
- ▶ Synchronisation de logiciels :
  - ▶ antivirus, bd, ...
- ▶ Data warehouse
- ▶ Systèmes de réservations, commerce électronique, ...
  
- ▶ Chaque jour JP-Morgan échangent des milliards de messages ...

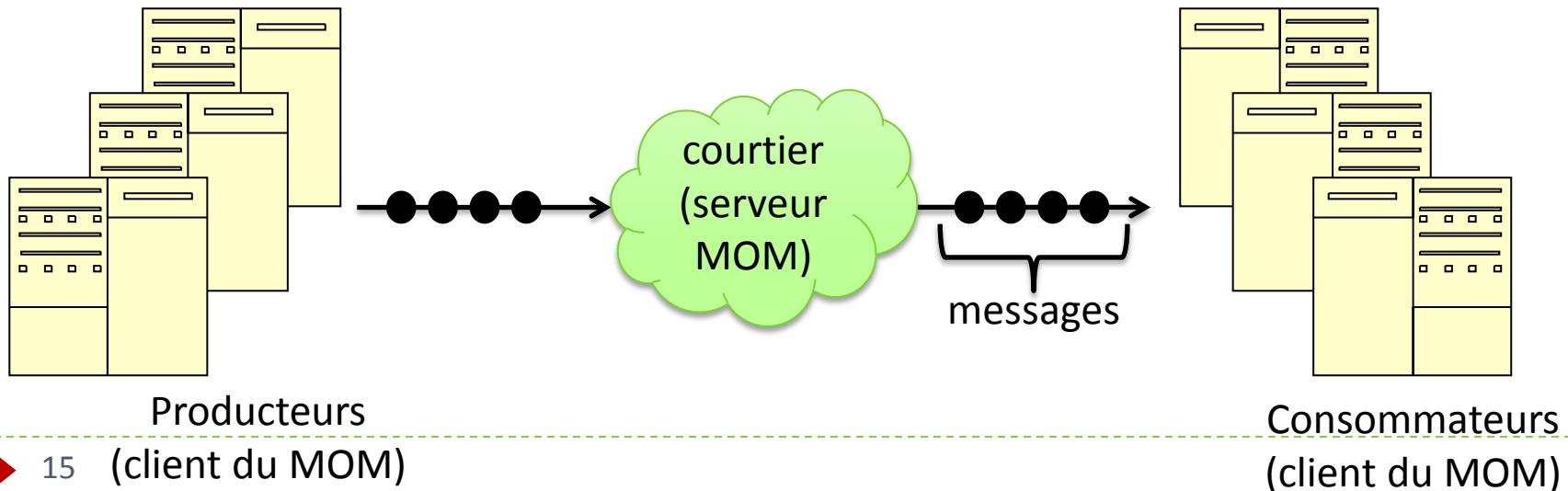
# Caractéristiques d'un MOM

---

- ▶ Garantie de délivrance des messages
- ▶ Support des transactions
- ▶ Gestion du routage des messages
- ▶ Passage à l'échelle
- ▶ Configuration diverses dont QoS (qualité de service)
- ▶ Faible couplage fonctionnel entre les composants
  - ▶ pas de références directes (pas besoin de connaître le nom de l'objet ou l'adresse IP du producteur)
  - ▶ pas de dépendances d'interfaces
  - ▶ pas de dépendance temporelle
  - ▶ mais souvent **dépendances de données**
    - ▶ généralement plus facile de se mettre d'accord
- ▶ mode asynchrone (pas de blocage) ou synchrone (parfois)
- ▶ load balancing
- ▶ supervision

# Architecture générale

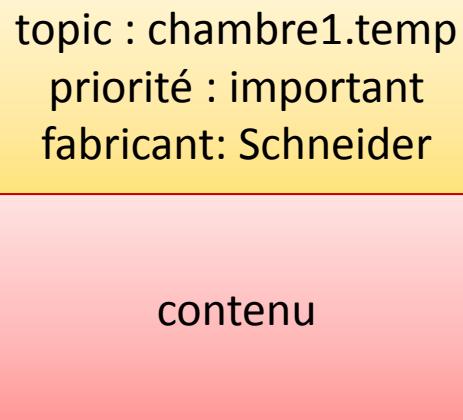
- ▶ On ne parle plus de clients et de serveurs mais de producteurs et consommateurs
  - ▶ souvent un producteur est le consommateur d'un autre producteur
  - ▶ on peut avoir des cycles
- ▶ Producteurs et consommateurs sont clients d'un courtier de message (centralisé ou pas) : le MOM
- ▶ Producteurs et consommateurs échangent des messages



# Contenu d'un message

---

- ▶ Le contenu dans différents formats :
  - ▶ type MIME : texte, son, images, binaires, ...
- ▶ Des propriétés
  - ▶ pour le routage (adresses, topics, ...)
  - ▶ pour la QoS
  - ▶ pour le filtrage
  - ▶ ...



# Différentes caractéristiques de communication

---

- ▶ Mode synchrone/asynchrone
- ▶ Données persistante/transitoire
- ▶ Unicast/Multicast
- ▶ Push/Pull

# Communication synchrone/asynchrone

---

- ▶ Mode synchrone :
  - ▶ L'émetteur reste bloqué jusqu'à ce que le destinataire acquitte la réception du message
  
- ▶ Mode asynchrone :
  - ▶ L'émetteur n'est pas bloqué et peut continuer à faire des traitements

# Données persistantes/transitoires

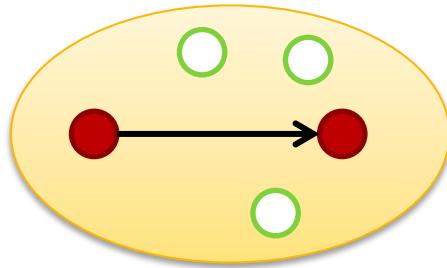
---

- ▶ Communication persistante :
  - ▶ Le MOM conserve les données jusqu'à ce que le récepteur consomme les données
    - ▶ le producteur et le consommateur n'ont pas besoins d'être présent en même temps pour faire le traitement
- ▶ Communication transitoire
  - ▶ Le MOM ne conserve pas les données, elles ne sont transmises que si le récepteur est présent
    - ▶ dépendance temporelle entre le récepteur et l'émetteur
- ▶ "Mixte"
  - ▶ le MOM ne conserve les données que pour une certaine durée (fixée par un TTL sur le message)

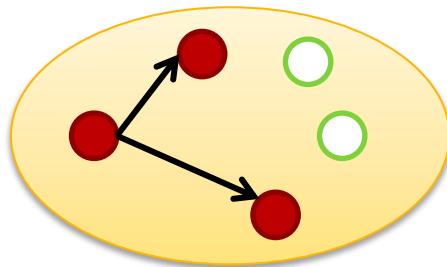
# Unicast/Multicast

---

- ▶ communication unicast :
  - ▶ le message n'est envoyé qu'à un seul destinataire à la fois



- ▶ communication multicast (diffusion)
  - ▶ le message peut être reçu par plusieurs destinataires abonnés



# Push/Pull

---

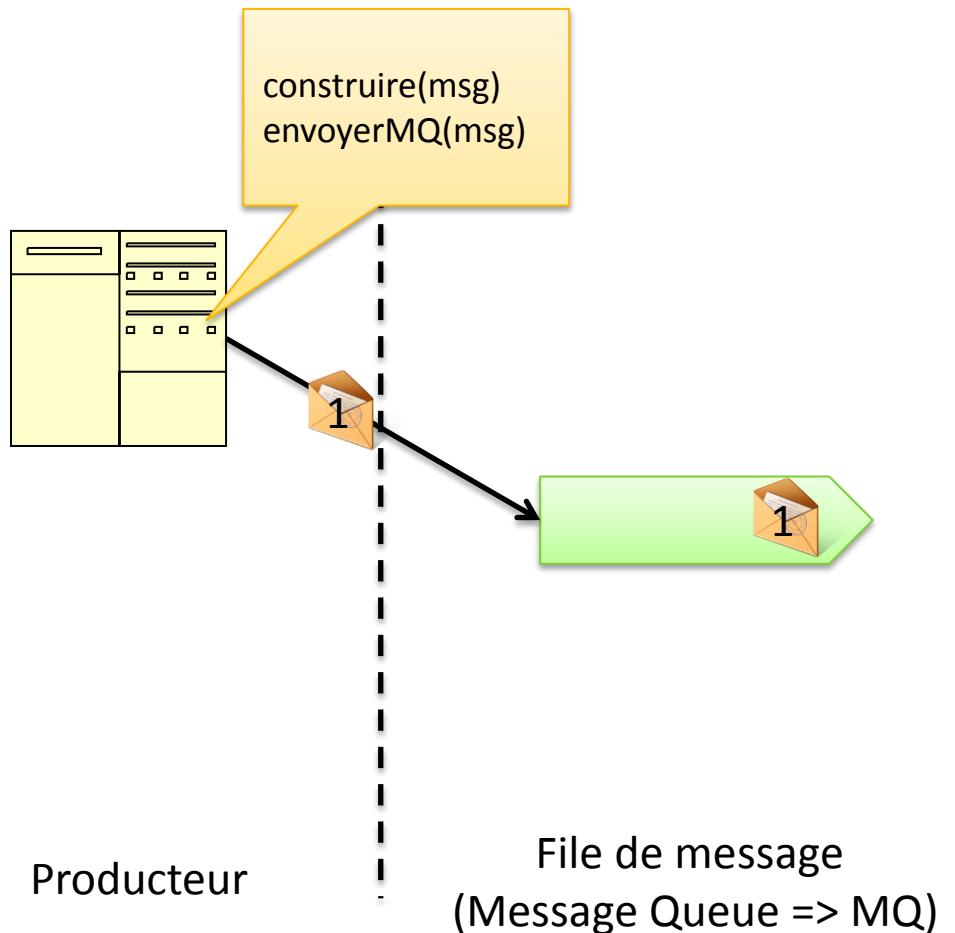
- ▶ En mode push :
  - ▶ le producteur envoie régulièrement des données
  - ▶ le consommateur enregistre un callback et est notifié de l'arrivée d'un message
  
- ▶ En mode pull
  - ▶ le consommateur demande régulièrement des données au producteur

# Files de messages

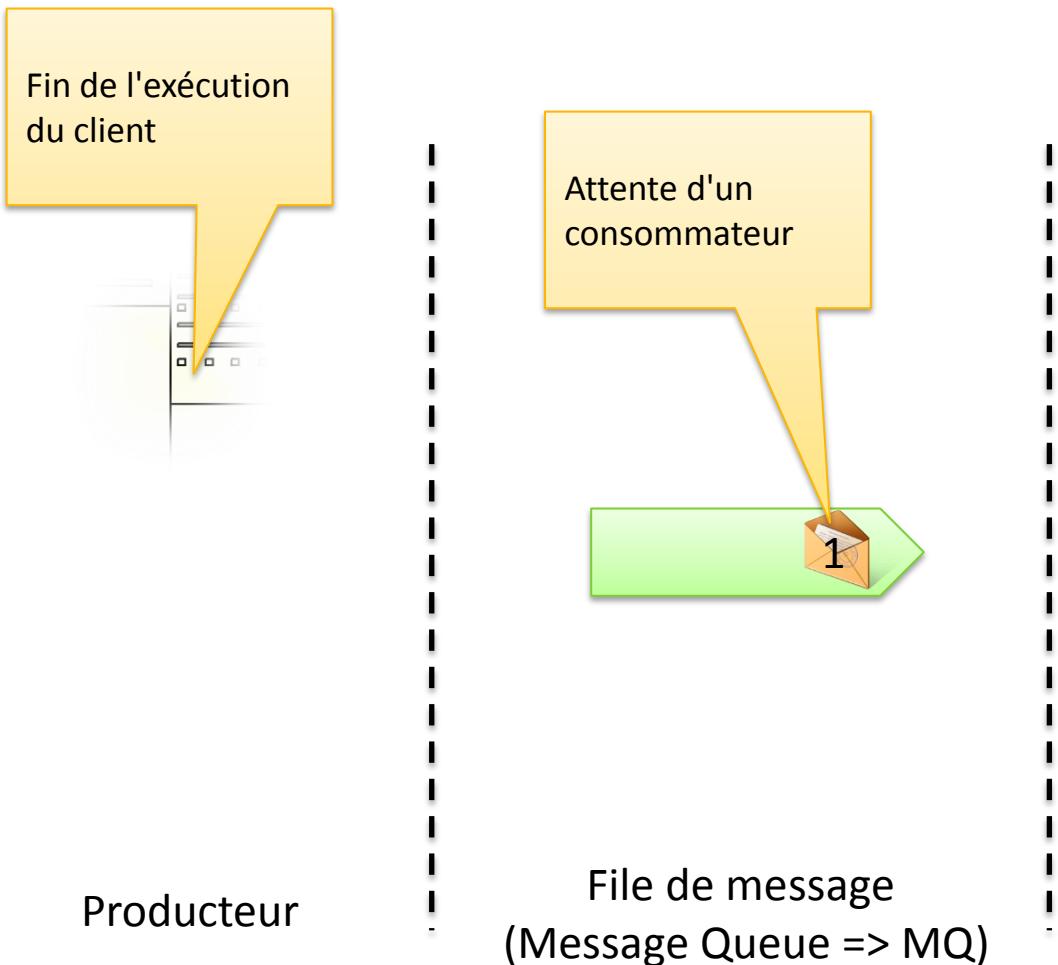
---

- ▶ Les messages sont transportés dans des **files de messages**
  - ▶ en anglais **Message Queue** (d'où le MQ dans le nom des MOM)
  - ▶ les messages sont mis dans une file d'attente en attente d'être relayée
    - ▶ généralement la file d'attente est persistante et les messages sont conservés tant qu'ils ne sont pas consommés
  - ▶ les files sont partagés entre plusieurs applications
  - ▶ les files peuvent prendre en compte de la QoS (priorité des messages)
  - ▶ les messages peuvent être filtrés à la reception

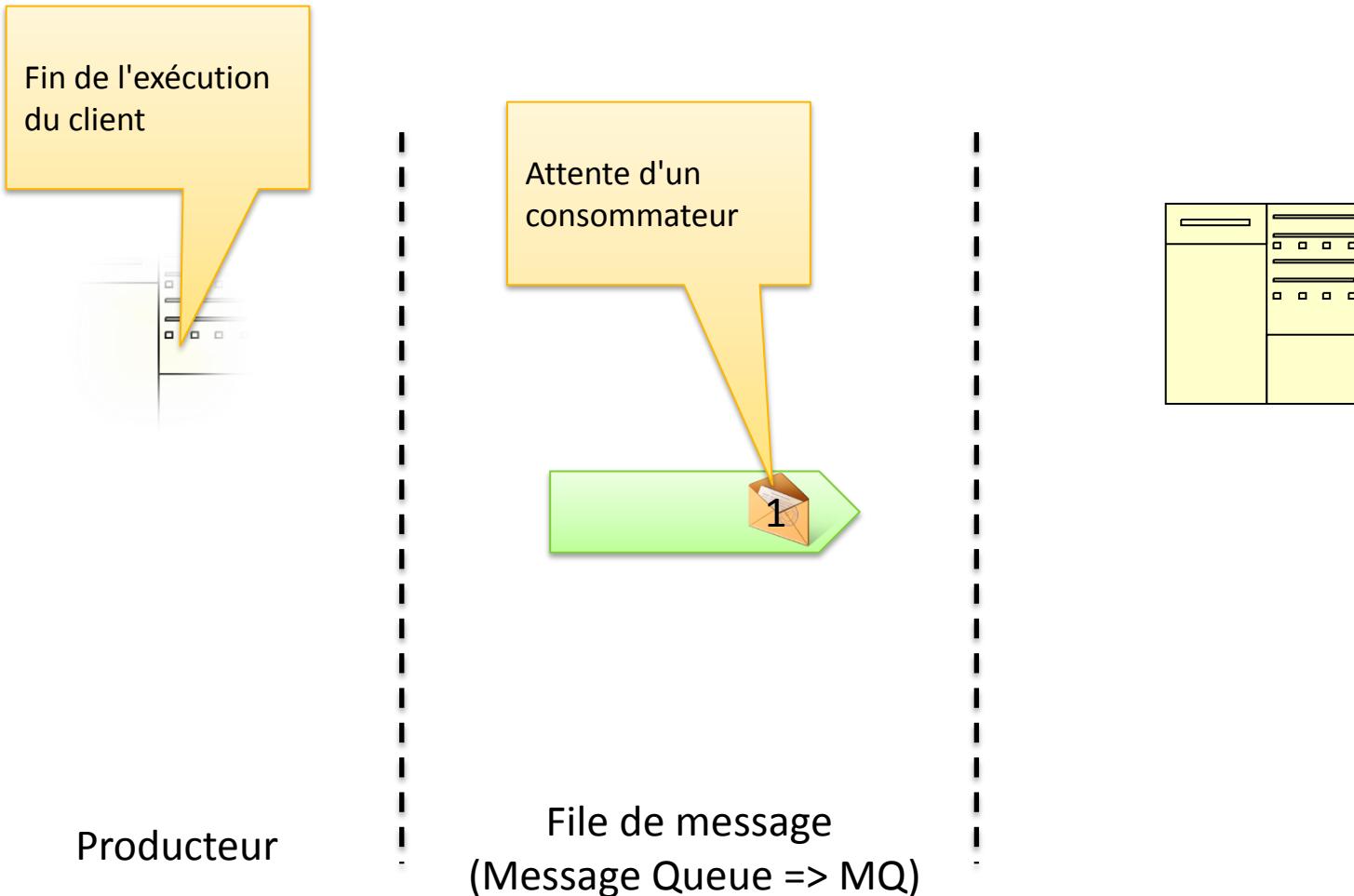
# Principe du store and forward



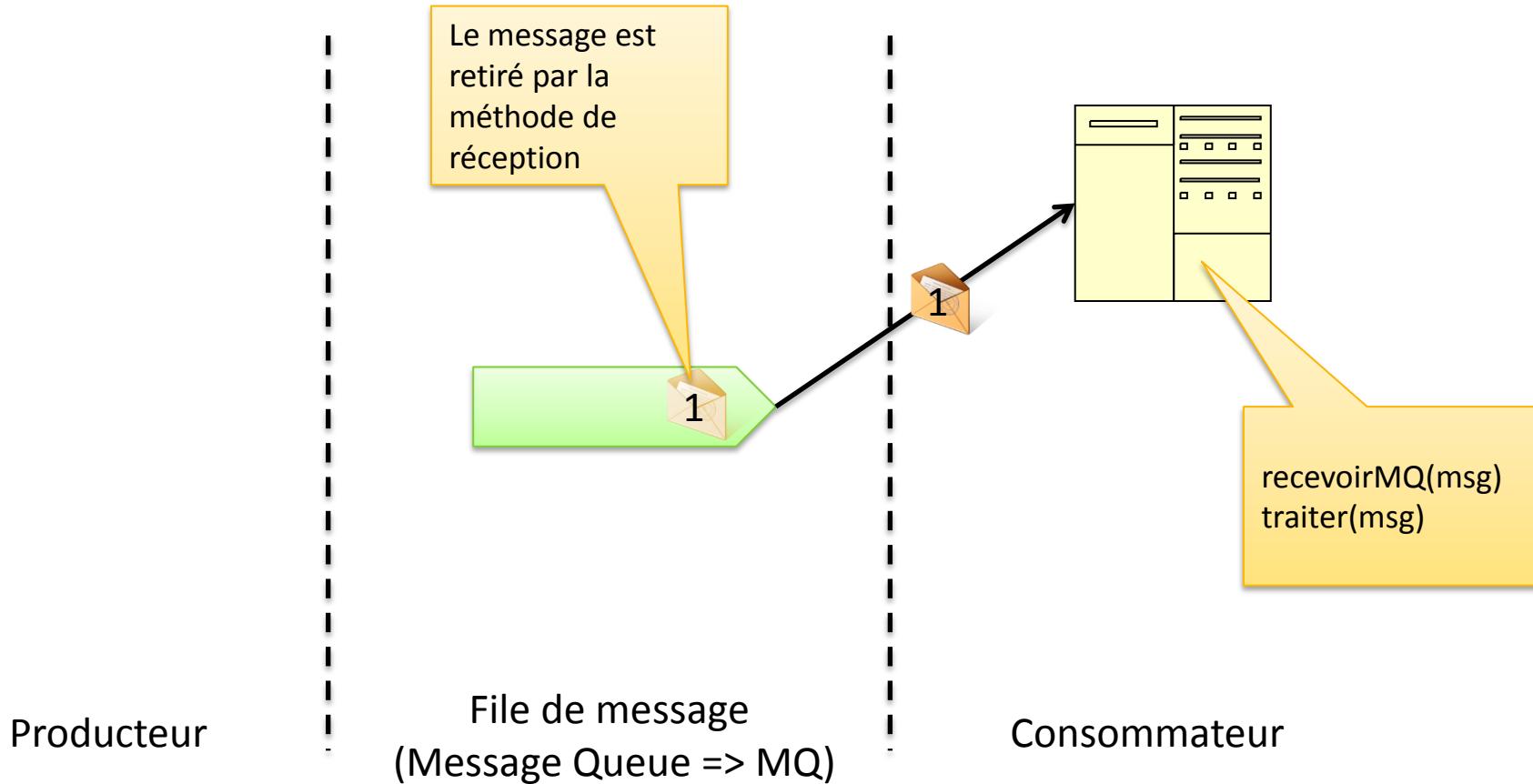
# Principe du store and forward



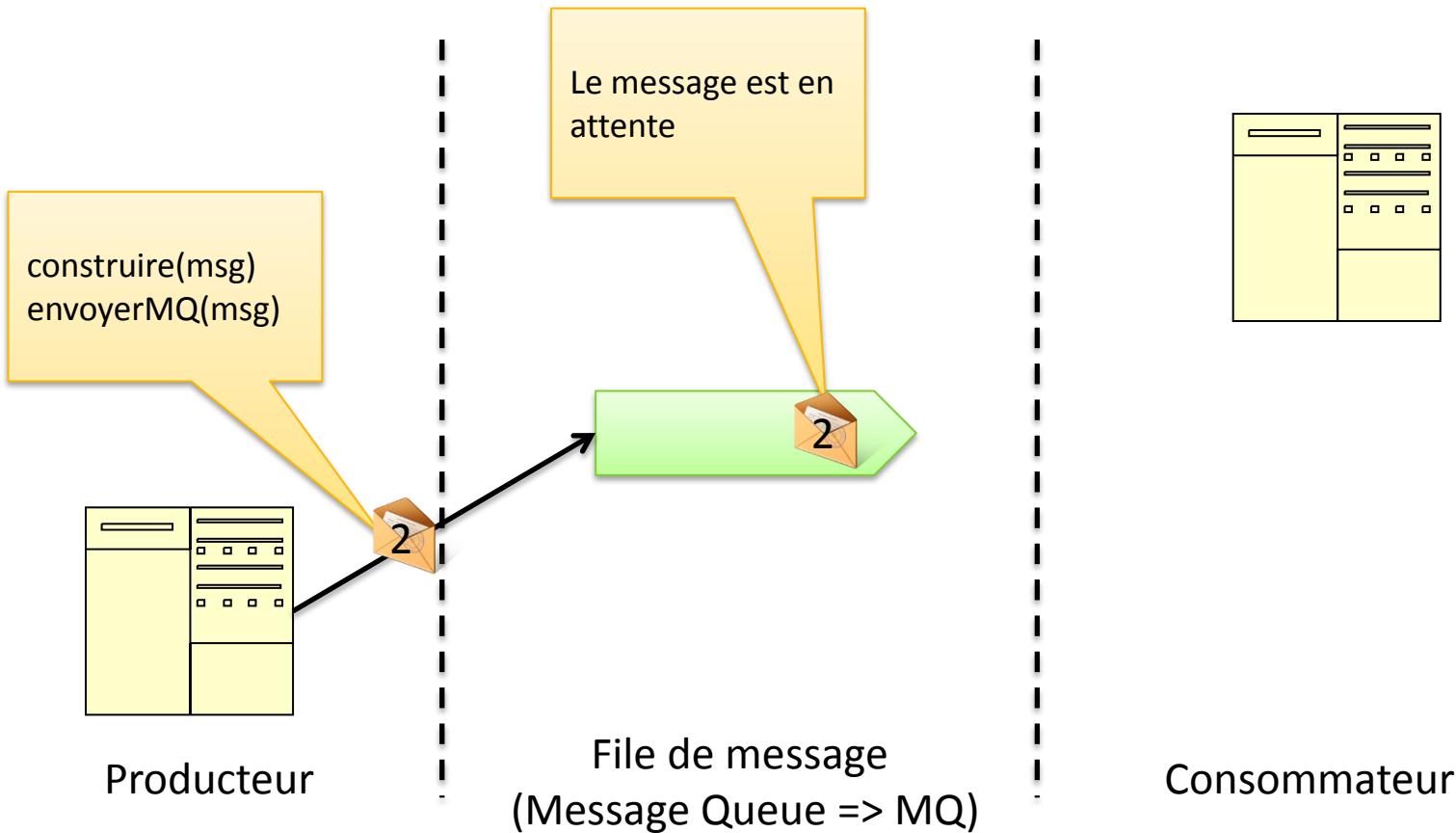
# Principe du store and forward



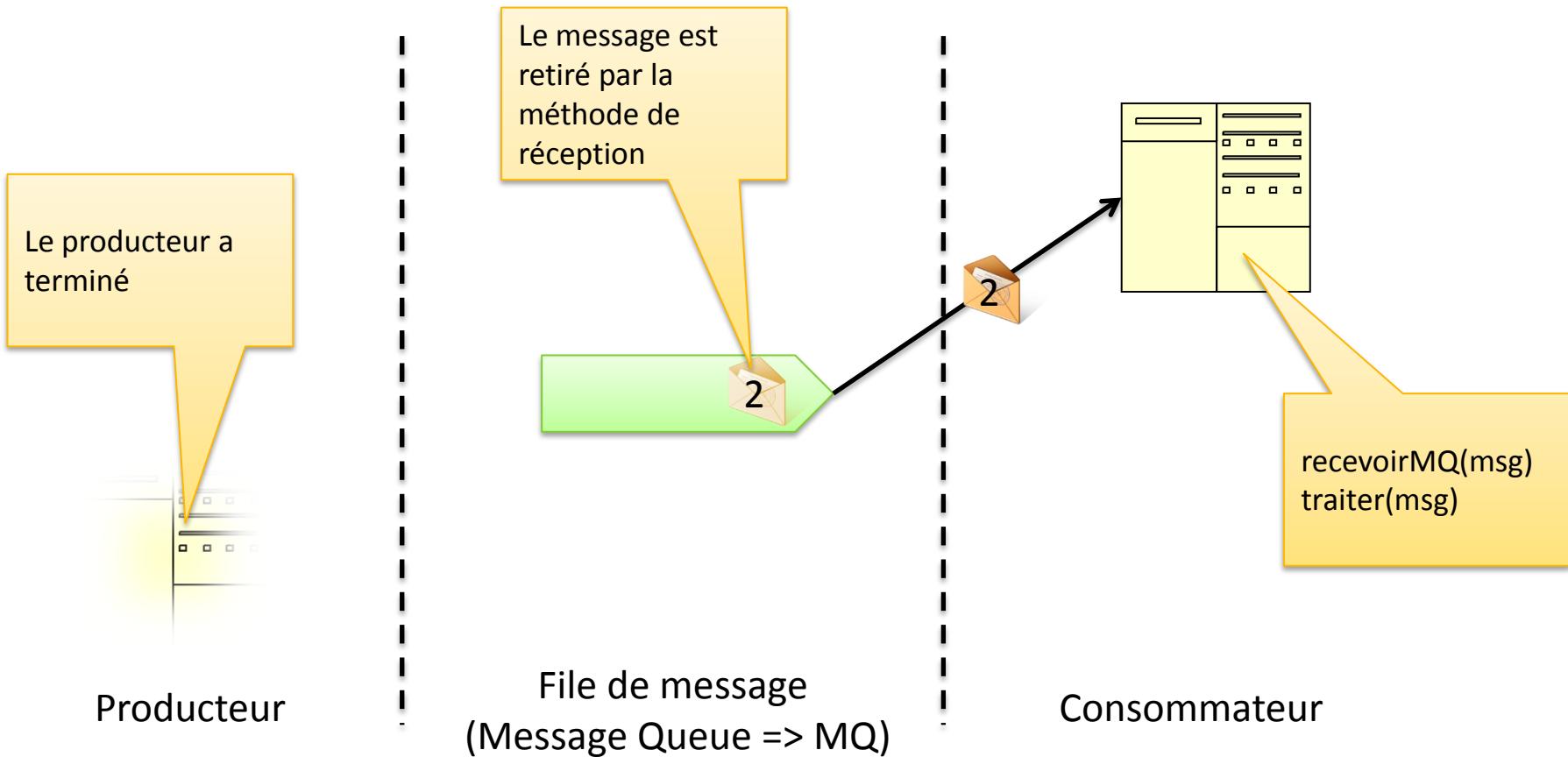
# Principe du store and forward



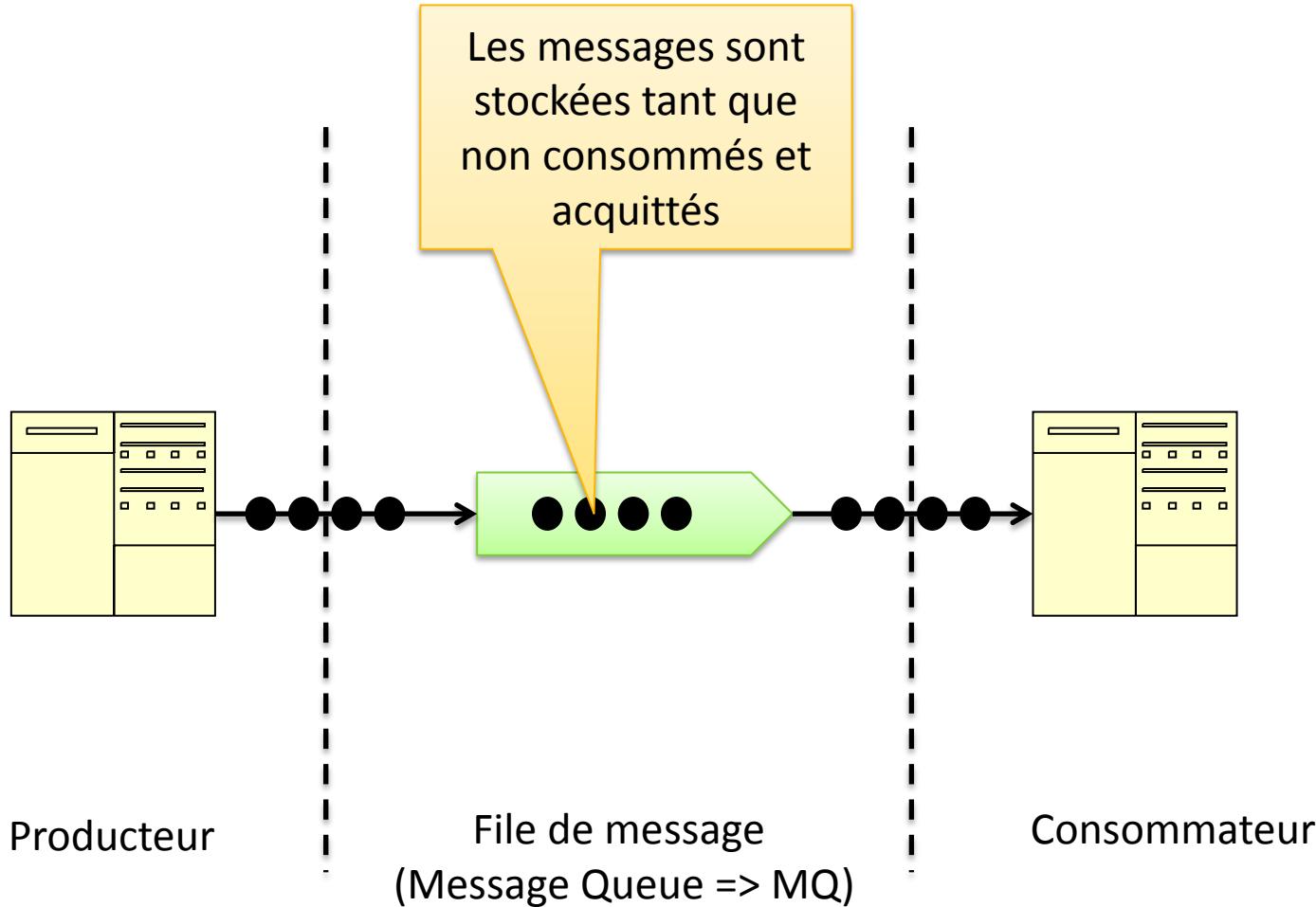
# Principe du store and forward



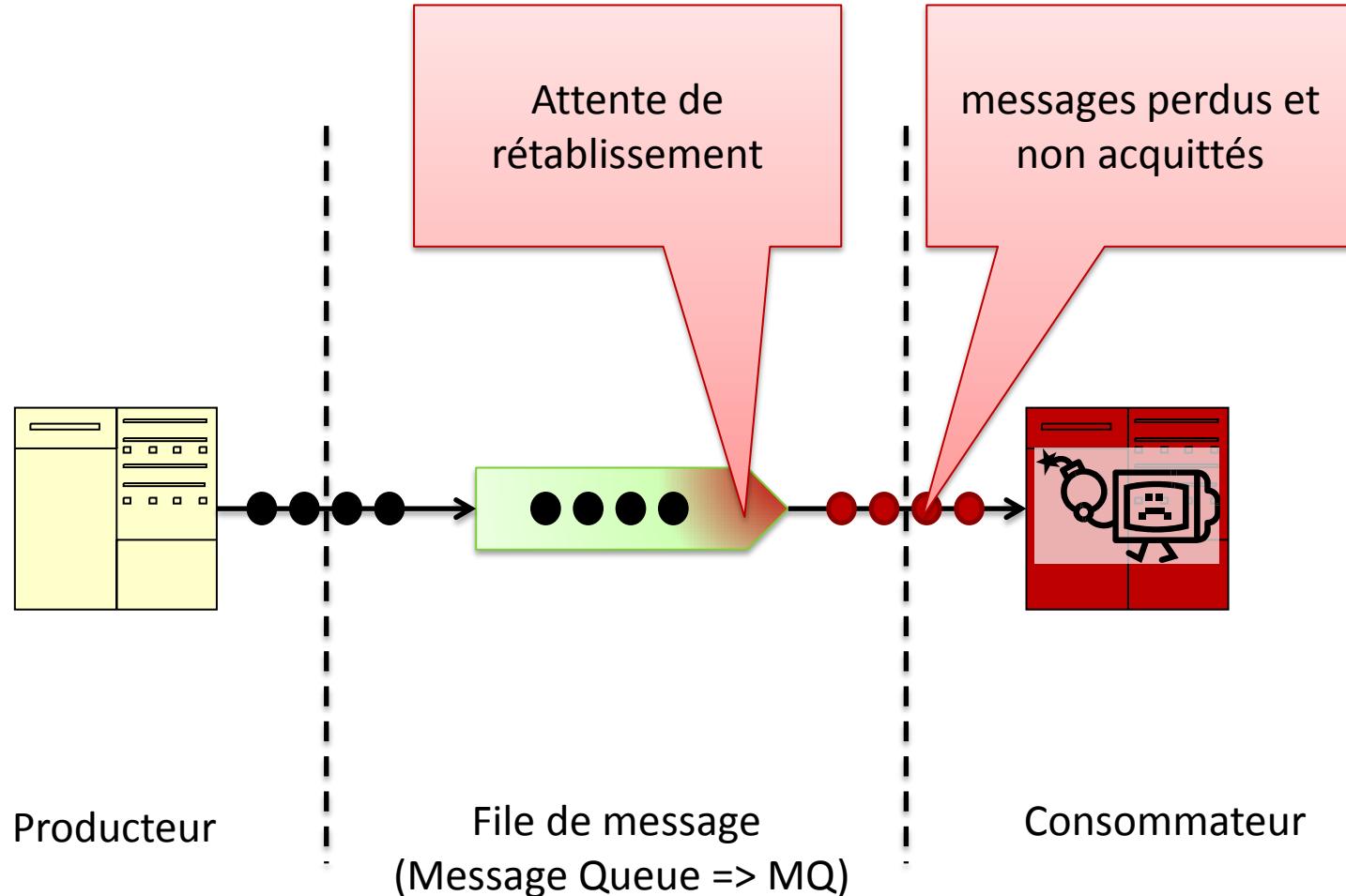
# Principe du store and forward



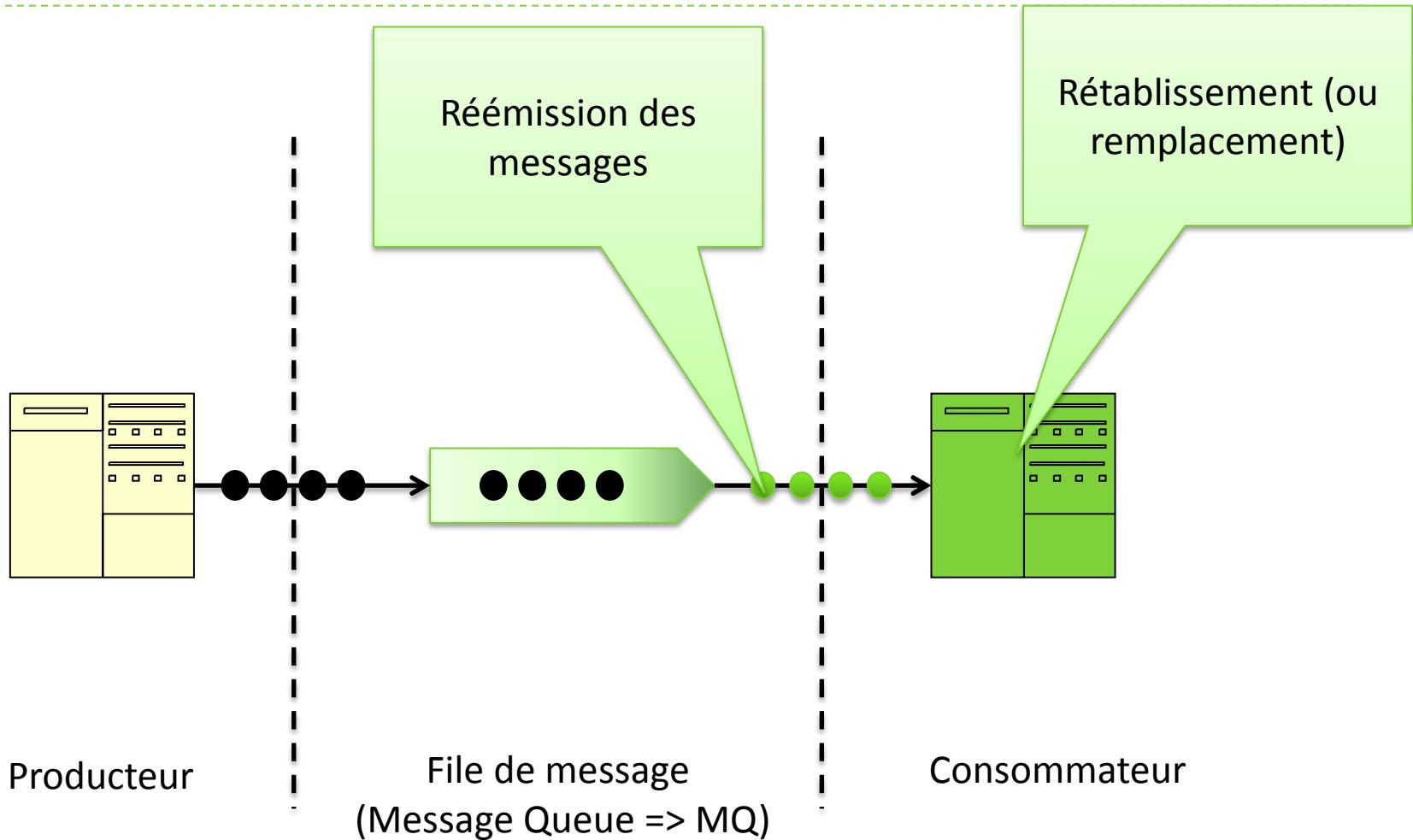
# En cas de pannes



# En cas de pannes



# En cas de pannes



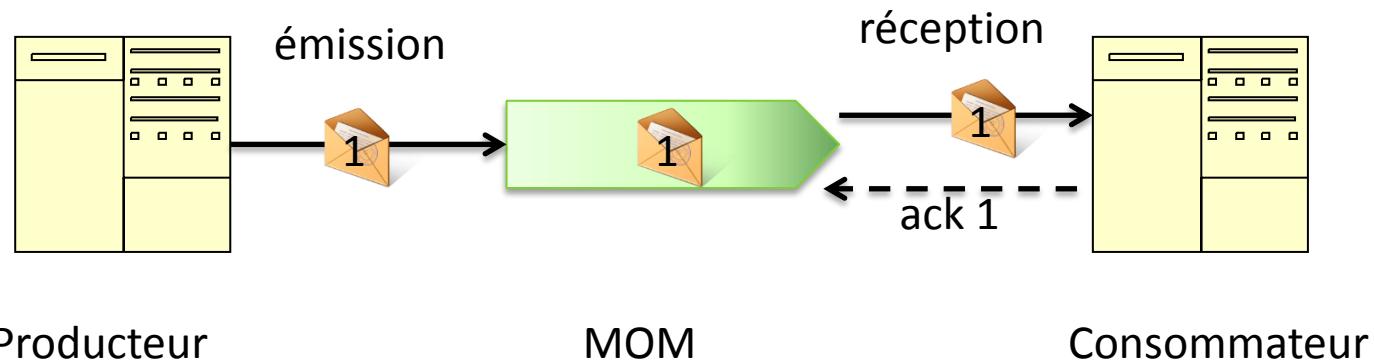
# Différents modèles de communications

---

- ▶ Point-à-point (point-to-point):
  - ▶ un message envoyé est consommé par un seul client
- ▶ Publication-souscription (publish-subscribe) :
  - ▶ un message publié est diffusé à tous les souscripteurs
- ▶ Publication-souscription par le contenu (content-based publish-subscribe)
  - ▶ un message publié est diffusé à tous les souscripteurs en fonction du contenu (ex : fabricant="Scheider", ...)
- ▶ Requête/Réponse
  - ▶ Client-Serveur en mode asynchrone (façon de faire du RPC asynchrone)

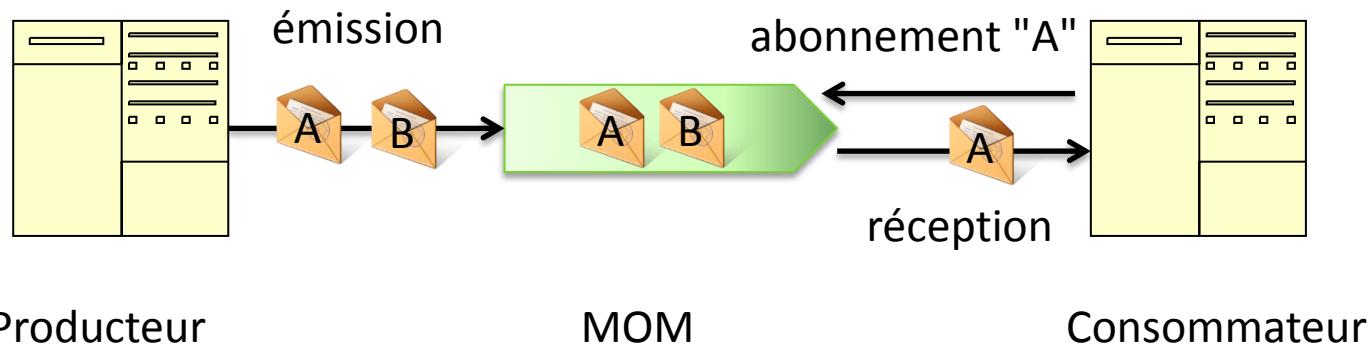
# Point-à-point

- ▶ Les messages sont stockés dans la file jusqu'à consommation
- ▶ Un message est consommé une seule fois (par un seul consommateur)
- ▶ Pas de dépendance temporelle entre producteur et consommateur du message
- ▶ Mécanisme d'acquittement possible



# Publish/Subscribe

- ▶ Les messages sont filtrés via des "topics" (sujets d'intérêts) ou le contenu
  - ▶ découplage complet entre producteurs et consommateurs
- ▶ Dépendance temporelle
  - ▶ Un producteur ne peut lire un message qu'après s'être abonné à un topic
  - ▶ Un producteur abonné à un topic doit continuer à être actif pour recevoir les messages du topic.

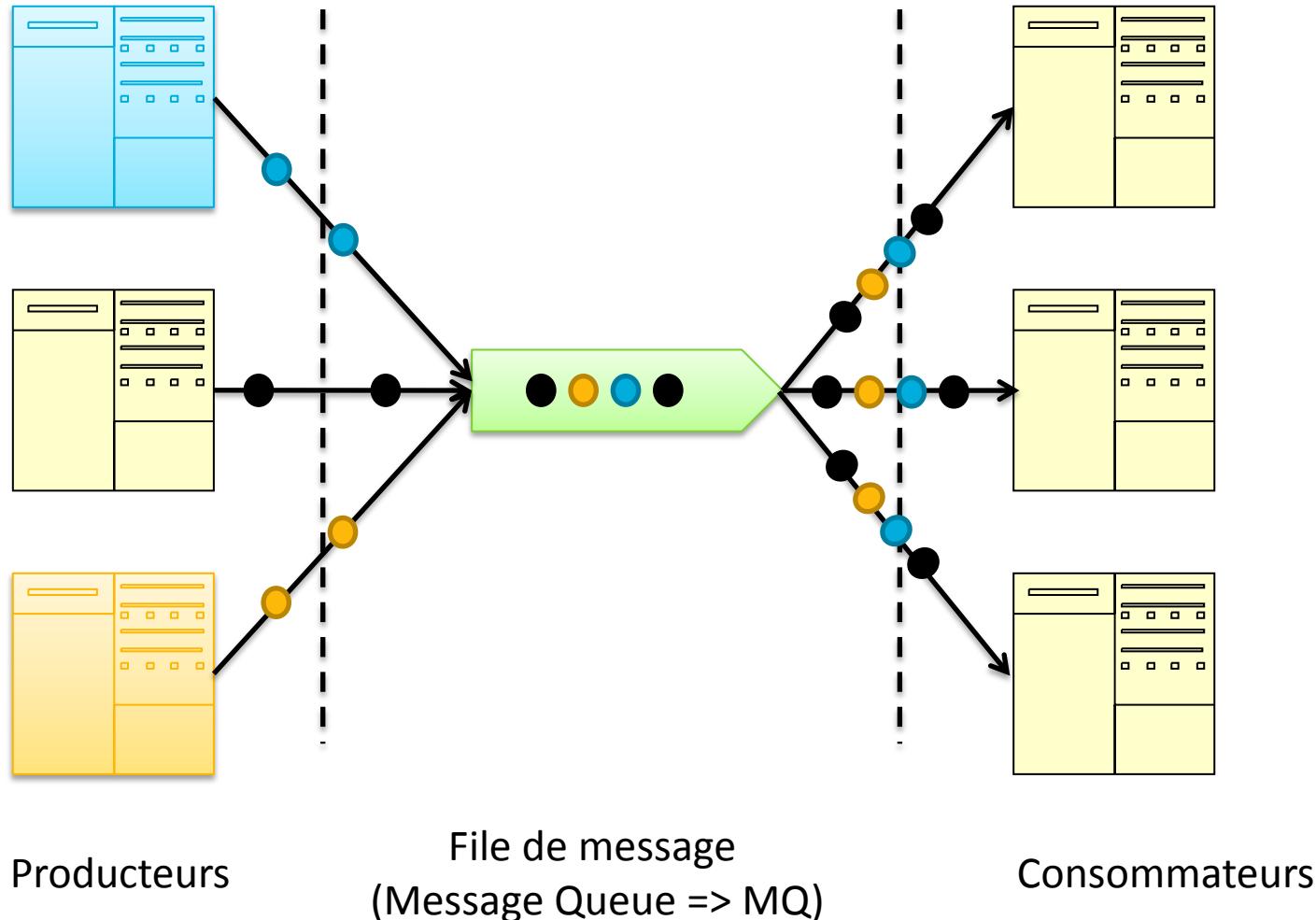


# Routage des messages

---

- ▶ Les messages peuvent être routés :
  - ▶ en fonction de l'identifiant de l'application
  - ▶ à l'aide d'une clef de routage (routing key)
  - ▶ en fonction du contenu (filtrage)
    - ▶ les applications définissent des critères sur les messages à consommer
      - ex : expression booléenne sur la valeur des champs

# Partage des files

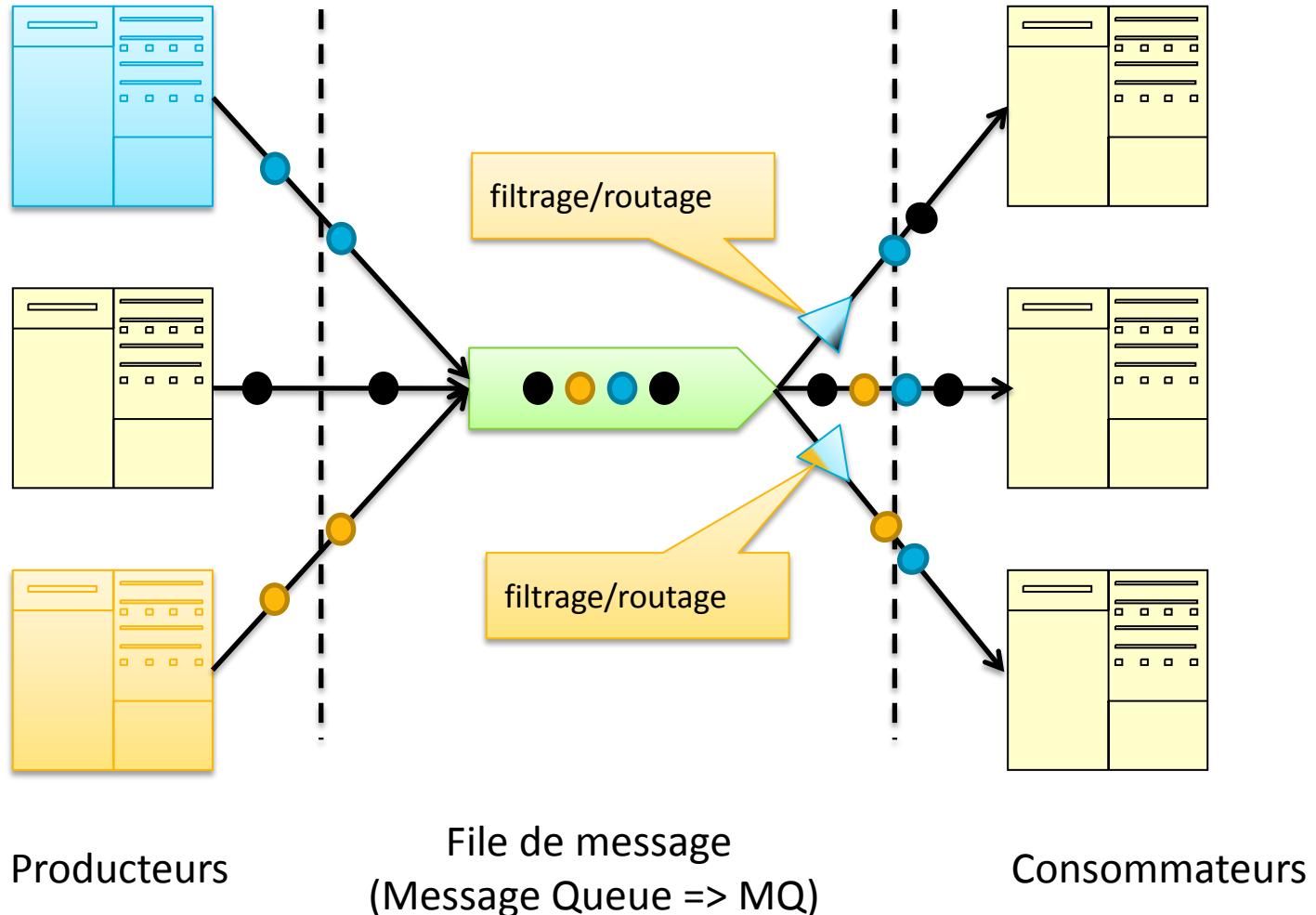


Producteurs

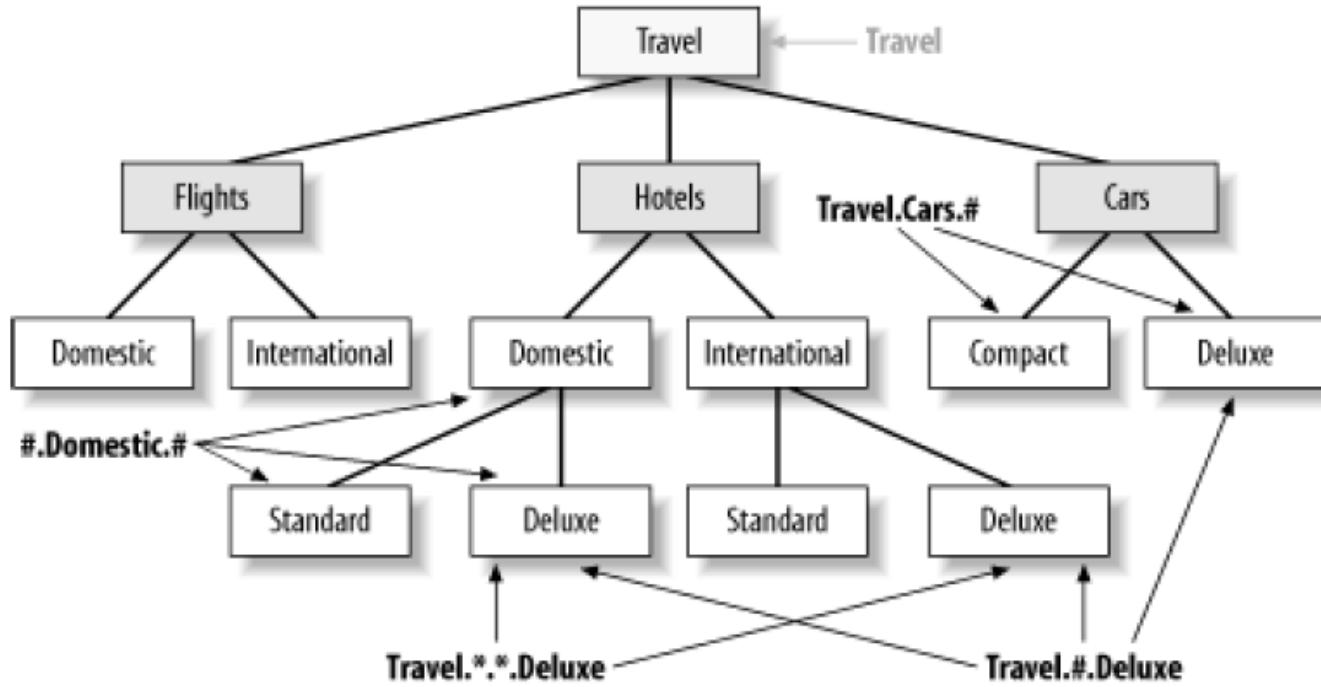
File de message  
(Message Queue => MQ)

Consommateurs

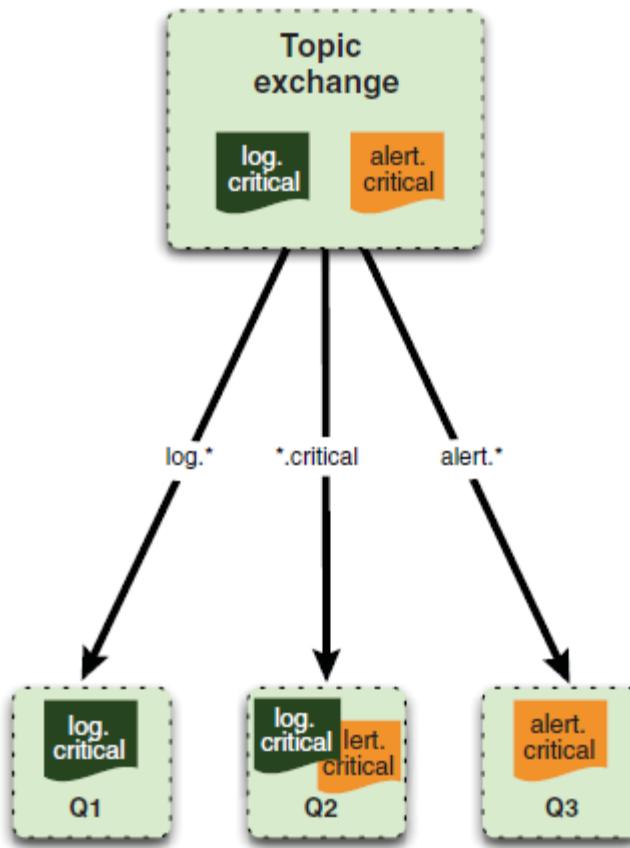
# Routage des messages



# Publication-Souscription sur des *topics* hiérarchiques



# Utilité de la hiérarchie

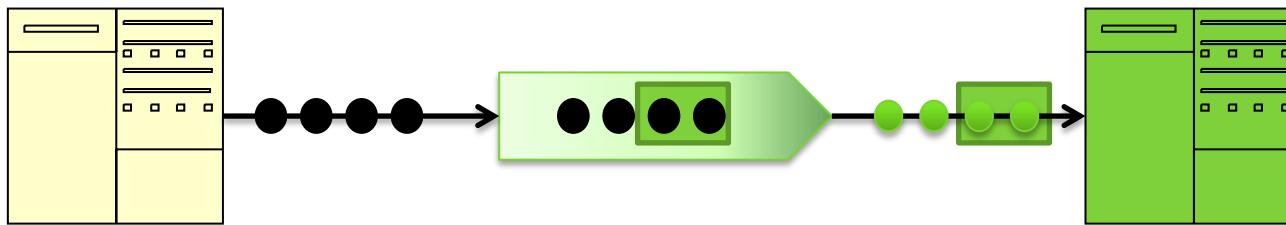


source RabbitMQ in Action

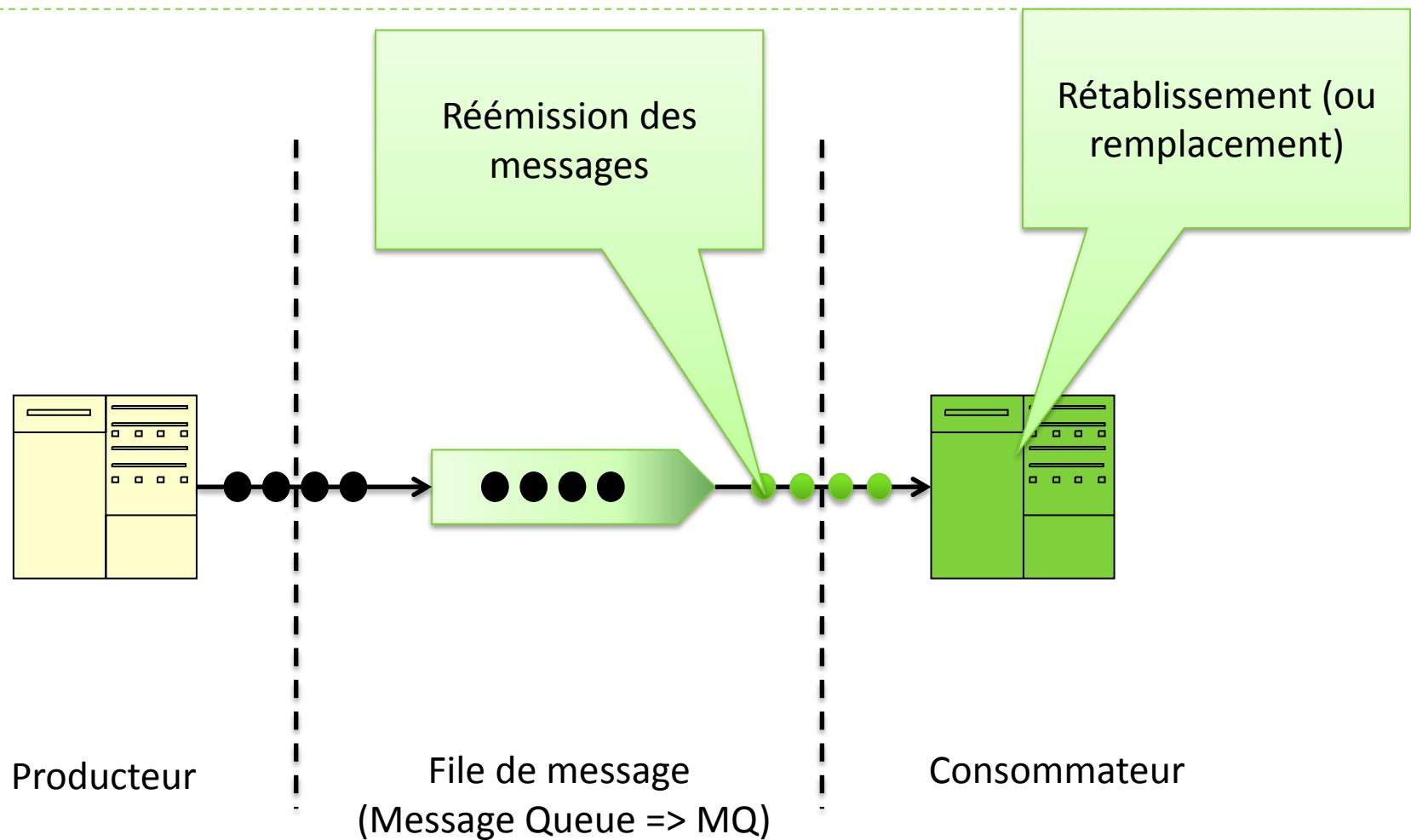
# Notion de transaction

---

- ▶ Il est possible de grouper les messages au sein d'une transaction :
  - ▶ les messages sont validés par lots
  - ▶ en cas d'abandon de la transaction, si tout les messages n'ont pas été reçus par le consommateur, les messages sont réémis pour être à nouveau traités



# En cas de pannes



# Topologie pour les serveurs MOM

---

- ▶ Le MOM doit être robuste, car une panne causera une défaillance générale du système.
- ▶ Plusieurs topologie existent :
  - ▶ centralisée
  - ▶ décentralisée
  - ▶ hybride

# Topologie centralisée

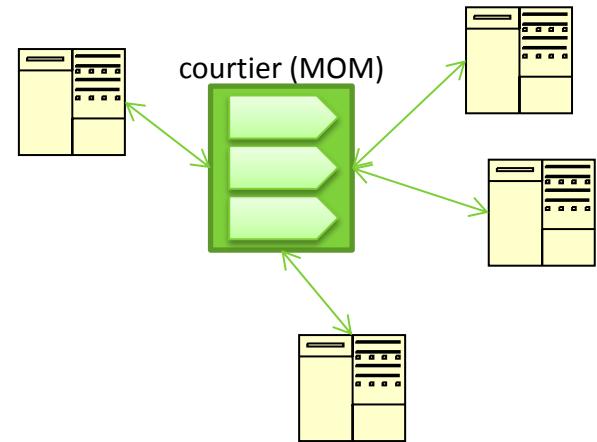
- ▶ Un serveur centralisé gère l'ensemble des messages

- ▶ Avantage :

- ▶ facile à mettre en œuvre
- ▶ garantie le découplage

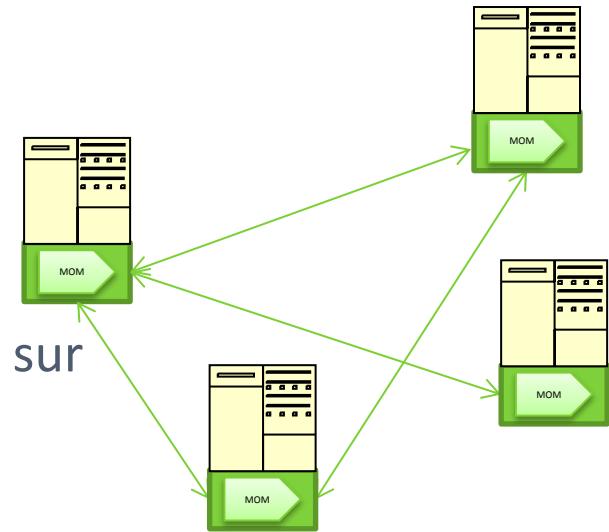
- ▶ Problèmes :

- ▶ "single point of failure" => une panne et plus rien ne fonctionne
- ▶ passage à l'échelle car le courtier doit pouvoir gérer un grand nombre de connexions
- ▶ distance Clients <-> MOM parfois grande (latence)



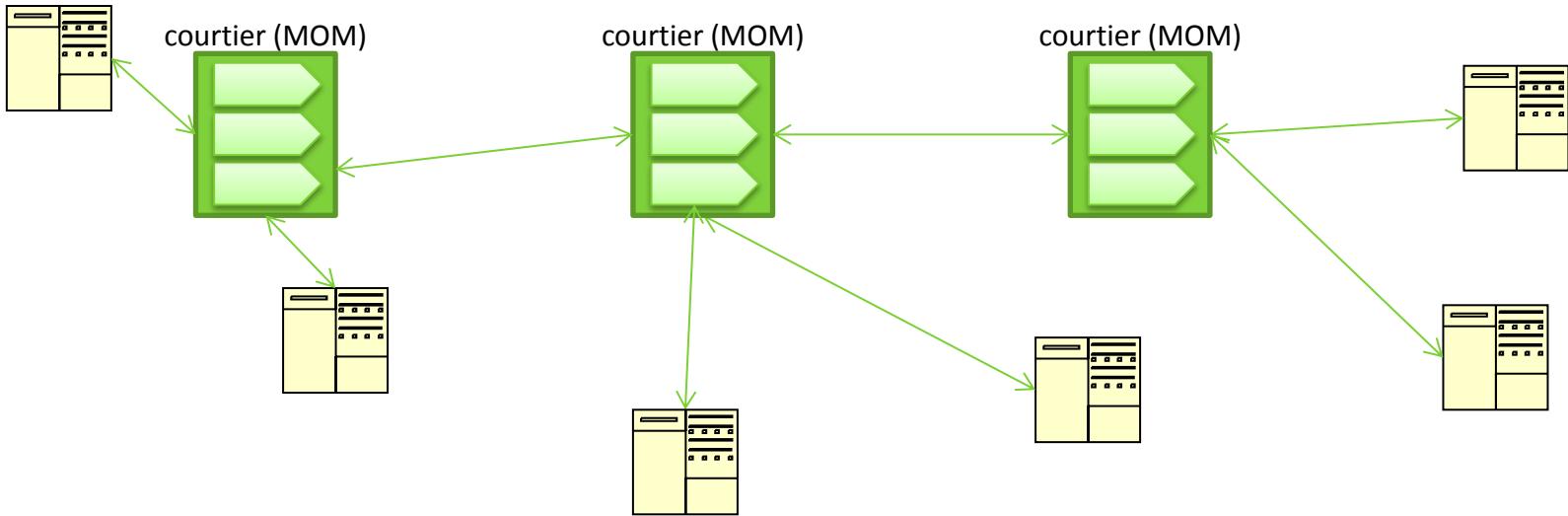
# Topologie complètement décentralisée

- ▶ Un MOM par composant (courtier local)
  - ▶ les MOM communiquent entre eux.
  - ▶ les MOM doivent se connaître ...
- ▶ Avantage :
  - ▶ la panne d'un MOM n'a pas de conséquence sur les autres
  - ▶ les performances sont meilleures à priori
- ▶ Problèmes :
  - ▶ duplication
  - ▶ lourdeur des clients
  - ▶ difficile à configurer



# Hybride

- ▶ On utilise plusieurs serveurs distribués





# Interopérabilité

# Bcp de MOM

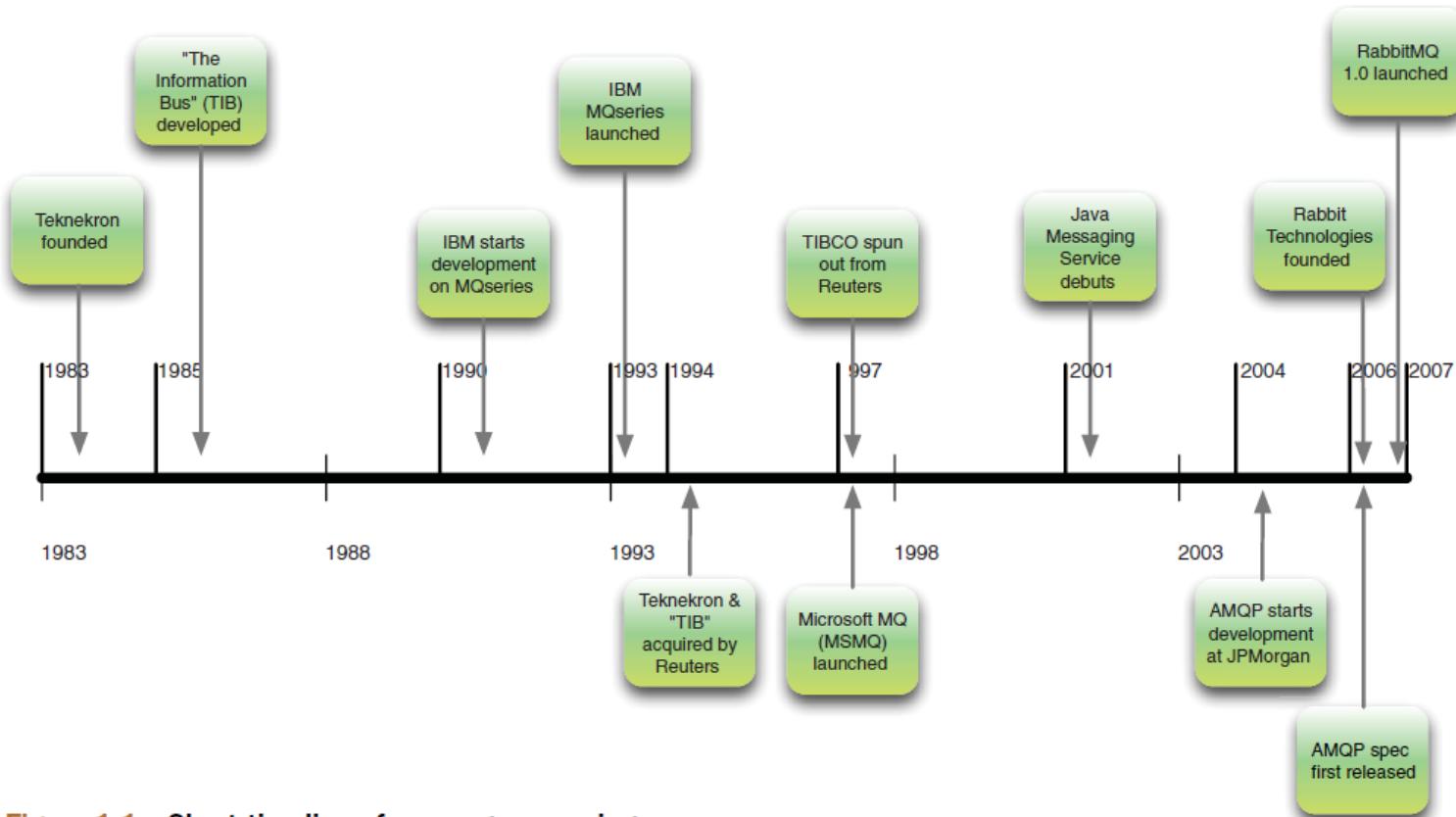


Figure 1.1 Short timeline of message queuing

source RabbitMQ in Action

# Différents fournisseurs, différentes implémentations

---

- ▶ Différents services offerts :
  - ▶ fiabilité, transaction, sécurité, acquittements
  - ▶ reconfiguration dynamique, etc ...
- ▶ Différentes API pour les clients
  - ▶ Un MOM = une API pour communiquer avec
- ▶ Différentes implémentations de protocoles de communication :
  - ▶ basé sur TCP, multicast, SSL, ...,
  - ▶ implémentation des files
  - ▶ topologies supportées, ...

# Money

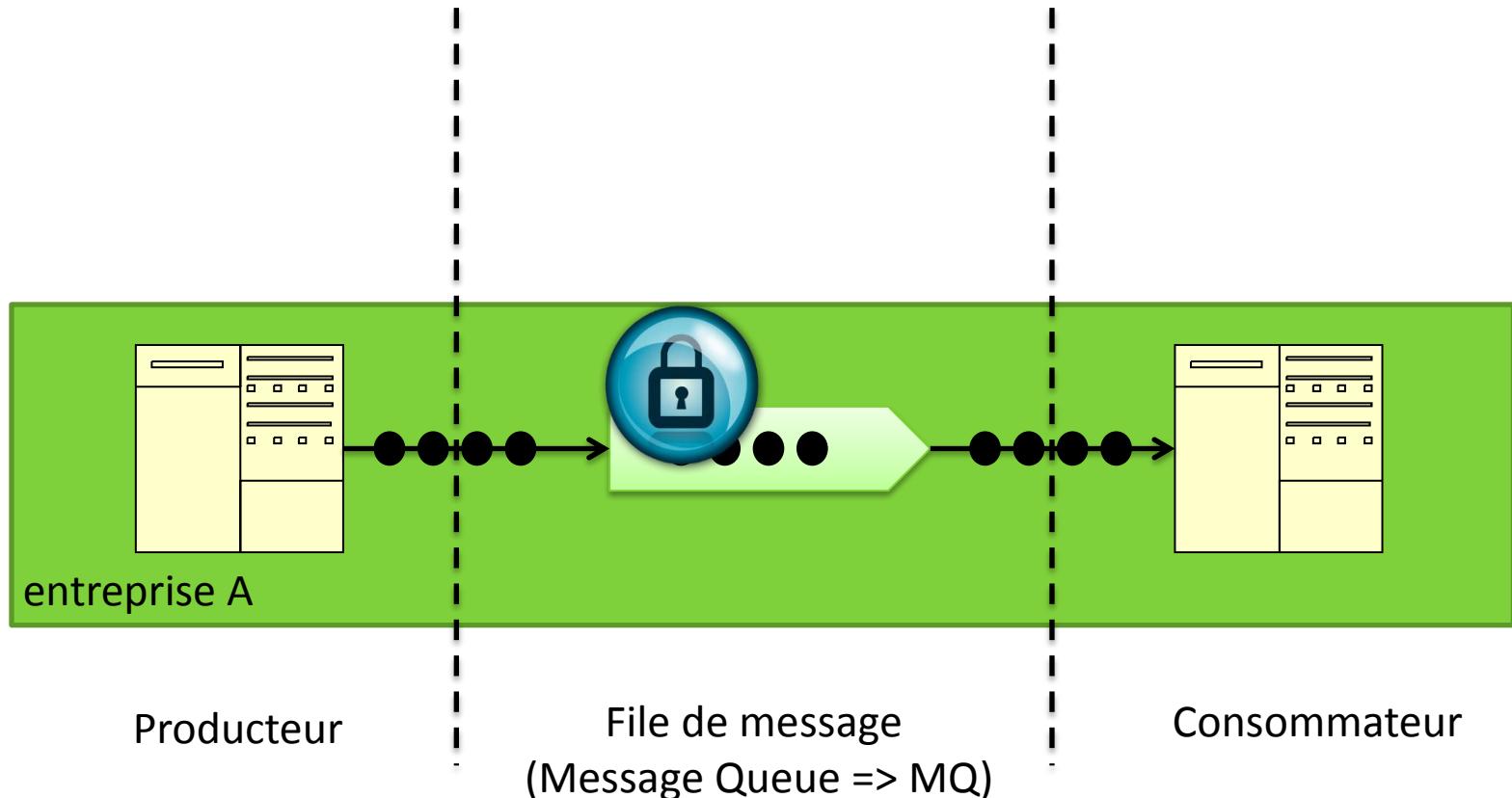
- ▶ Les solutions sont verrouillées par les fournisseurs

"Avec plus de 10 000 clients à travers le monde, Websphere MQ, ex-MQSeries, est une mine d'or pour IBM. Wintergreen Research estime qu'en 2006, Big Blue a encaissé près de 580 millions de dollars en licence et maintenance... pour un marché total du MOM (middleware orienté message) inférieur à 720 millions de dollars !"

<http://pro.01net.com/editorial/390957/amqp-rajeunit-la-messagerie-interapplicative/>

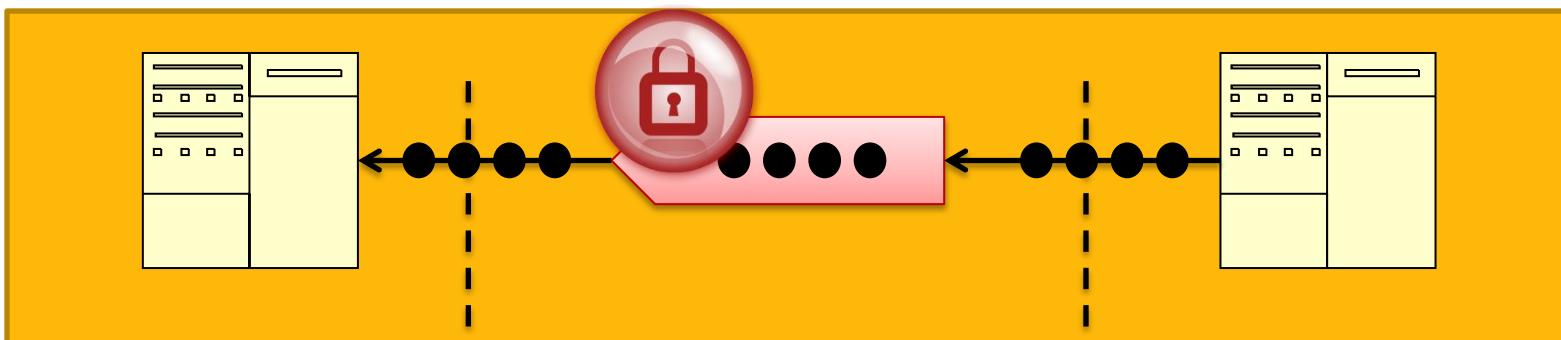


# Interopérabilité couteuse ...



# Interopérabilité couteuse ...

entreprise B (fusion/acquisition)



Producteur

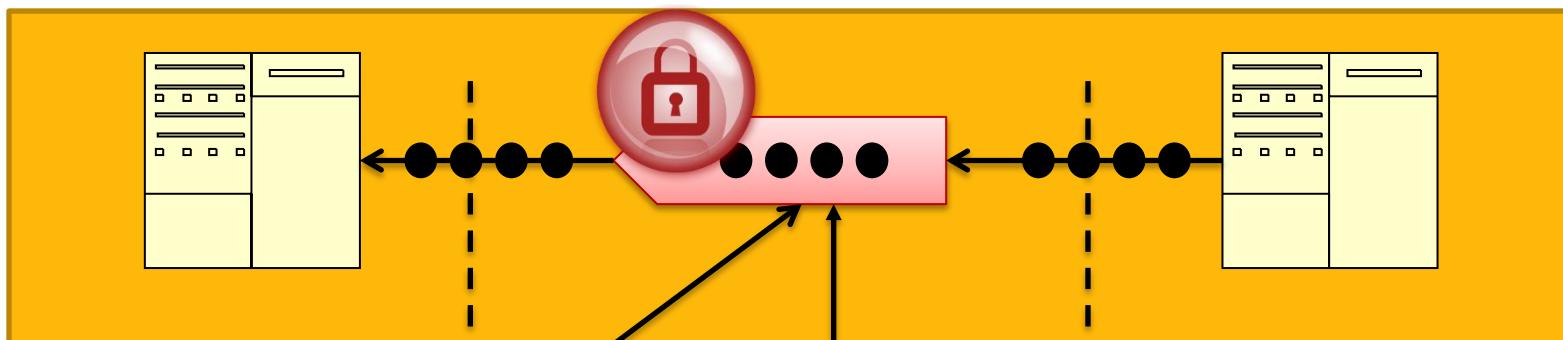
File de message  
(Message Queue => MQ)

Consommateur

# Interopérabilité couteuse ...



entreprise B (fusion/acquisition)



Producteur

File de message  
(Message Queue => MQ)

Consommateur

# Il faut des standards

---

- ▶ Pour ne pas avoir à recoder chaque application lorsqu'on change de MOM
  - ▶ il faut une API standard pour les clients
  - ▶ c'est l'idée derrière JMS
- ▶ Pour pouvoir faire communiquer les MOM entre eux
  - ▶ solutions propriétaires (chaque fournisseur implémente des bridges et fait payer ...)
  - ▶ une architecture globale et ouverte : AMQP

donc JMS et AMQP ne sont pas opposés mais complémentaires



## JMS : une API commune

# API javax.jms

---

- ▶ Offre deux modèles d'interactions :
  - ▶ point-à-point : utilise le concept de **Queue**
  - ▶ publication/souscription : utilise le concept de **Topics**
- ▶ Donc deux types de **destinations** : Queue ou Topics
- ▶ Les destinations peuvent être administrés ou non :
  - ▶ possibilité de les créer dynamiquement dans le programme
  - ▶ ou nécessité de les déclarées via une interface d'administration (sécurité)

En mode Topic, un consommateur absent lors de l'envoi du message ne le recevra pas. Il est donc considéré moins fiable que le mode Queue.

# Structure des messages

---

- ▶ Les consommateurs et producteurs ne partagent pas d'interfaces mais des messages. Les messages sont constitués :
  - ▶ d'un en-tête
    - ▶ JMSDestination, JMSDeliveryMode, JMSMessageID, JMSTimestamp, JMSExpiration, JMSRedelivered, JMSPriority, JMSReplyTo, JMSCorrelationID, JMSType
  - ▶ des propriétés qui peuvent être utilisées pour filtrer les messages
    - ▶ exemple Fabricant="Scheider"
  - ▶ du contenu (texte, binaire, ou autre)
    - ▶ JMS supportent plusieurs natures de messages : TextMessage , BytesMessage, MapMessage, StreamMessage, Object-Message

# Push et Pull

---

## ▶ Pull :

- ▶ le consommateur peut appeler une méthode **receive** sur la destination (pas le producteur)
- ▶ la méthode est bloquante jusqu'à réception d'un message

## ▶ Push

- ▶ le consommateur peut enregistrer un callback **onMessage** qui sera appelé à la réception d'un nouveau message.
- ▶ le broker JMS se charge de l'appel à on Message

# Notion de session et connexion

---

- ▶ Connexion :
  - ▶ permet de faire le lien avec le broker (masque les détails nécessaires à la connexion avec le broker)
  - ▶ permet de gérer des aspects d'authentifications
  - ▶ Une connexion peut avoir plusieurs sessions
  - ▶ les fabriques de connexions sont généralement découverte via JNDI
    - ▶ JNDI est une API Java très utilisée permettant notamment de s'interfacer avec des annuaires LDAP
- ▶ Sessions
  - ▶ délimitation temporelle permettant de grouper les envois et les réceptions de messages avec des propriétés spécifiques (QoS)
    - ▶ transactionnel ou non
    - ▶ priorités des messages
    - ▶ séquence des messages
    - ▶ acquittements

# Exemple de création de sessions et connexions

---

- ▶ Cette exemple utilise JNDI et un annuaire permettant de découvrir les ressources

```
InitialContext jndiContext=new InitialContext();
//Rechercher une fabrique de connexion :
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);
//Créer la connexion :
Connection connection=cf.createConnection();
//Creer la session :
Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
```

# Les Destinations

---

- ▶ Les messages peuvent être envoyées vers deux types de destinations les Queues et le Topics selon le mode de communication choisi.

```
//Creer une Queue et d'un Topic  
Queue myQueue = session.createQueue("MyQueue");  
Topic myTopic = session.createTopic("MyTopic");  
  
//Retrouver les objets dans JNDI :  
Destination dest1=(Queue) jndiContext.lookup("/jms/myQueue");  
Destination dest2=(Topic)jndiContext.lookup("/jms/myTopic");
```

# Producer/Consumer

- ▶ Il faut créer des producer/consumer pour pouvoir envoyer/recevoir des messages.

```
MessageProducer producer=session.createProducer(dest1);
Message m=session.createTextMessage();
m.setText("hello world !");
producer.send(m);
```

```
MessageConsumer consumer=session.createConsumer(dest1);
connection.start();
Message m=consumer.receive();
```

La méthode est bloquante

# Exemple de listener

---

- ▶ Il est aussi possible de créer des listeners côté consommateur

```
MessageListener listener=new myListener();
consumer.setMessageListener(listener);
```

- ▶ Le listener doit alors avoir une méthode onMessage(Message msg)

```
public class MyListener implements MessageListener{
    public void onMessage(Message msg){
        //traite le message
    }
}
```

# Filtrage des messages

- ▶ Il est possible de créer des filtres sur les propriétés de messages :
- ▶ ((Pays = "France") OR (Pays="UK")) AND (fabricant="Zenio")

```
TextMessage message = session.createTextMessage("Bonjour");
message.setStringProperty("lang", "fr");
publisher.send(message);
```

S'abonne aux messages qui ont la bonne valeur

ajoute une propriété au message

```
MessageConsumer consumer = session.createConsumer(topic, "lang='fr'");
connexion.start();
TextMessage mess = (TextMessage) consumer.receive();
```

# Et plein d'autres trucs ...

---

- ▶ Notion d'abonnés durables (en mode topic)
  - ▶ par défaut pas de fiabilité : les absents ont torts
  - ▶ possibilité de dire que le consommateur d'un topic doit vraiment recevoir les messages
  - ▶ `session.createDurableSubscriber(topic, "Abonnement");`
- ▶ Politiques d'acquittements
- ▶ Utilisation des transactions
- ▶ ....
- ▶ Dans la pratique pas si facile à utiliser que ça en a l'air.

# Beaucoup d'implémentations

---

- ▶ Open sources :
  - ▶ [Apache ActiveMQ](#)
  - ▶ [OpenJMS](#)
  - ▶ JBoss Messaging et HornetQ de [JBoss](#)
  - ▶ [JORAM](#), de [ObjectWeb](#) maintenant [OW2](#)
  - ▶ Open Message Queue, de [Sun Microsystems](#)
- ▶ Commerciales :
  - ▶ BEA Weblogic
  - ▶ Oracle AQ
  - ▶ [SAP NetWeaver](#)
  - ▶ SonicMQ
  - ▶ [Tibco Software](#)
  - ▶ webMethods Broker Server
  - ▶ [WebSphere MQ](#)
  - ▶ FioranoMQ de Fiorano



## AMQP/RabbitMQ

# Pourquoi AMQP ?

---

- ▶ JMS est une API
  - ▶ elle ne résout pas le problème des différents brokers
  - ▶ de leur dépendances
  - ▶ et du multi-langage/multiplateformes => c'est toujours dépendant des brokers
    - ▶ et de la bonne volonté des fournisseurs/codeurs
- ▶ JP-Morgan n'était pas satisfait des solutions proposées
  - ▶ ils ont proposés de faire une spécification de broker libre : AMQP
- ▶ AMQP 1.0 est un standard OASIS

# Plein de partenaires

---

## ▶ Financial Services

- ▶ Bank of America
- ▶ Bloomberg Finance L.P.
- ▶ Credit Suisse
- ▶ Deutsche Börse
- ▶ JPMorgan Chase Bank

## ▶ Technology Providers

- ▶ Informatica
- ▶ Microsoft
- ▶ Red Hat
- ▶ SITA
- ▶ Software AG
- ▶ US Department of Homeland Security
- ▶ WSO2
- ▶ ....

# Et plein d'implémentations

---

- ▶ Pleins d'implémentations de brokers compatibles
  - ▶ [SwiftMQ](#), a commercial [JMS](#), AMQP 1.0 and AMQP 0.9.1 broker and a free AMQP 1.0 client.
  - ▶ [Windows Azure Service Bus](#), Microsoft's cloud based messaging service
  - ▶ [Apache Qpid](#), an [open-source](#) project at the [Apache Foundation](#)
  - ▶ [Apache ActiveMQ](#), an [open-source](#) project at the [Apache Foundation](#)
  - ▶ [Apache Apollo](#), [open-source](#) modified version of the ActiveMQ project at the [Apache Foundation](#). The threading functionality of ActiveMQ has been replaced and non-blocking techniques implemented more widely.
  - ▶ [RabbitMQ](#) an [open-source](#) project sponsored by [Pivotal](#), supports AMQP 1.0 and other protocols via plugins

# Et plein de plateformes supportées

---

- ▶ Exemple pour rabbitmq :
  - ▶ Solaris
  - ▶ BSD
  - ▶ Linux
  - ▶ MacOSX
  - ▶ TRU64
  - ▶ Windows NT/2000/XP/Vista/Windows 7/Windows 8
  - ▶ Windows Server 2003/2008/2012
  - ▶ Windows 95, 98
  - ▶ VxWorks

# Et plein de langage

---

- ▶ Exemple pour rabbitmq

- ▶ [Java / JVM](#)
- ▶ [Ruby](#)
- ▶ [Python](#)
- ▶ [.NET](#)
- ▶ [PHP](#)
- ▶ [Perl](#)
- ▶ [C / C++](#)
- ▶ [Erlang](#)
- ▶ [Node.js](#)
- ▶ [Go](#)
- ▶ [Common Lisp](#)
- ▶ [Haskell](#)
- ▶ [Ocaml](#)
- ▶ [COBOL](#)
- ▶ [AS/400](#)
- ▶ ....

et ...

---

- ▶ plein d'APIs ....
  - ▶ les APIs clientes sont différentes pour chaque brokers et pour chaque langage.
  - ▶ on verra l'API RabbitMQ pour Java

# RabbitMQ en pratique

Basé sur les tutoriaux de rabbitmq  
(<https://www.rabbitmq.com/tutorials/tutorial-one-java.html>)

# L'API java dépend de

---

- ▶ commons-io-1.2.jar, commons-cli-1.1.jar, rabbitmq-client.jar
- ▶ Ils doivent être dans le classpath pour que votre programme fonctionne.

```
$ export CP=.:commons-io-1.2.jar:commons-cli-1.1.jar:rabbitmq-client.jar  
$ java -cp $CP Send
```



## Exemple de point-à-point basique

## En mode file

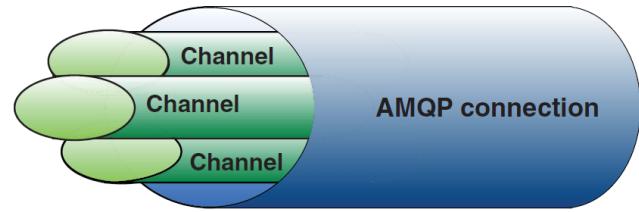
---

- ▶ On veux faire communiquer un producteur P avec un client C



- ▶ Il suffit de créer une file entre les deux, déclarée auprès du serveur RMQ

# Connexion et canaux



- ▶ Grossso modo la même idée que pour JMS
- ▶ La connexion représente la communication avec le serveur. Les canaux (Channel) permettent d'échanger des messages.
- ▶ Les Channels
  - ▶ un identifiant unique
  - ▶ notion de multiplexage de la connexion TCP. Fonctionne comme un canal virtuel à l'intérieur d'une même connexion TCP :
    - ▶ économie de ressources
    - ▶ pas de perte de temps pour créer la connexion TCP
    - ▶ configuration

# Côté producteur (connexion et canal)

## ▶ Création de la connexion :

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
```

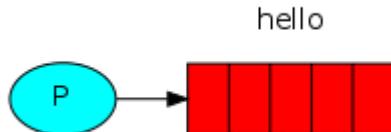
## ▶ Création du canal :

```
Channel channel = connection.createChannel();
```



# Côté producer (file)

## ► On déclare une File



effacement automatique quand plus utilisée ?

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

le nom

autre propriétés ...

durable ? La file doit elle survivre à un redémarrage du serveur

exclusive ? restreindre à la connexion

# Côté producteur (envoie d'un message)

- ▶ On peut envoyer le message (sous forme de bytes)

```
String message = "Hello World!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
```

nom de  
l'exchange  
(cf plus  
loin)

nom de la file  
(ou routing key)

propriétés

message en octet

# Côté producteur (complet)

```
private final static String QUEUE_NAME = "hello";

public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

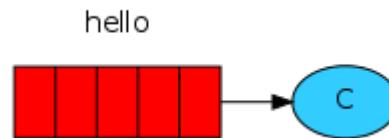
    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    String message = "Hello World!";
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
}
```

On pense à fermer

# Côté consommateur (s'attacher à la file)

- ▶ On doit maintenant relier le consommateur à la file



- ▶ Le code est le même que pour le producteur

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();
```

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
```

## Coté consommateur (déclaration d'un consumer)

- ▶ La consommation de message se fait par un objet spécial le Consumer :

```
QueueingConsumer consumer = new QueueingConsumer(channel);
```

- ▶ Le consumer est attaché au canal. On doit aussi expliquer au canal qu'on souhaite consommer dans la file :

```
channel.basicConsume(QUEUE_NAME, true, consumer);
```

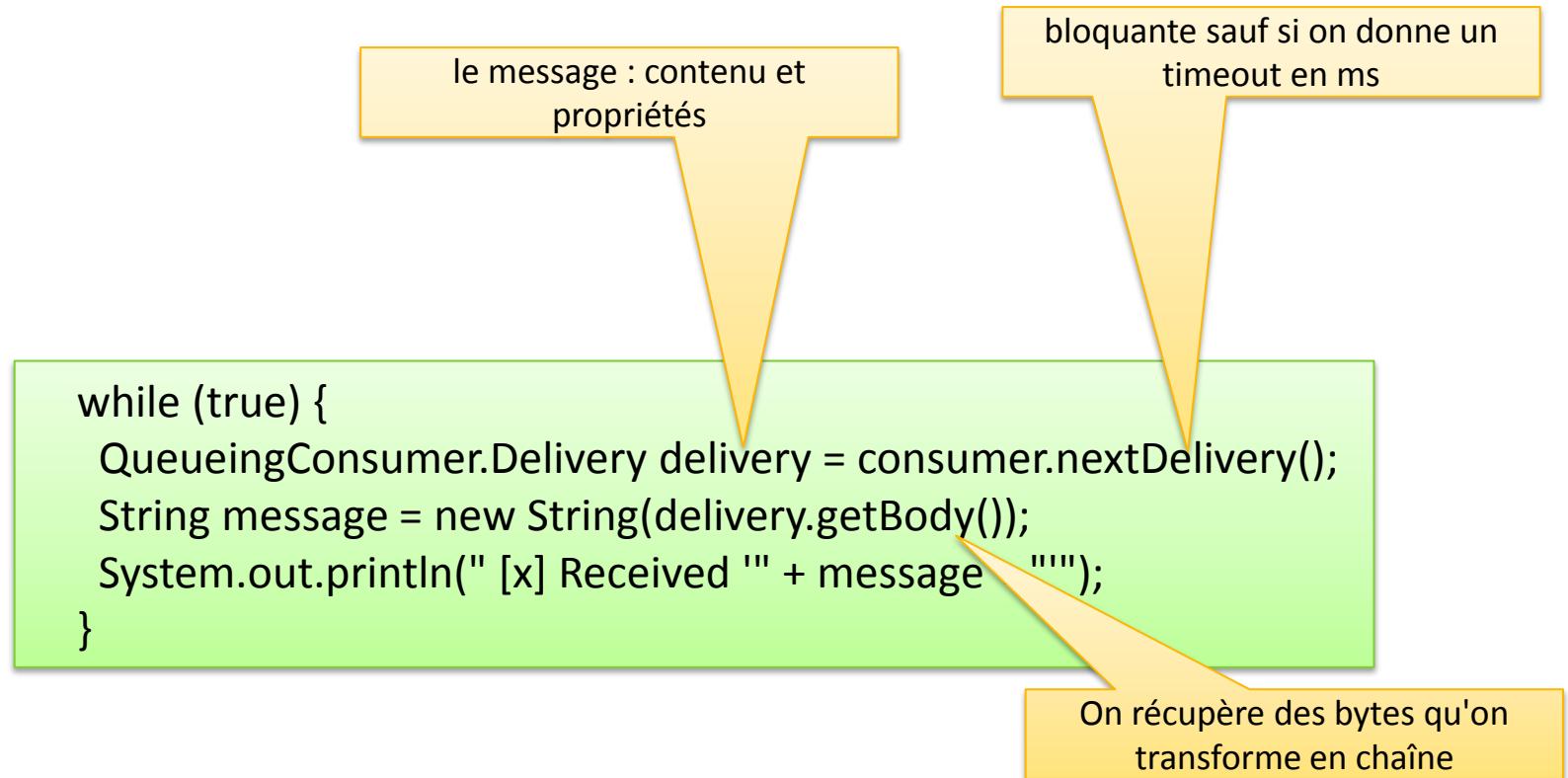
la file

Acquittement automatique sur  
réception ?

Le consommateur

# Côté consommateur (utilisation du consumer)

- Le consumer de base permet de récupérer les messages avec des méthodes bloquantes:



# Côté consommateur (utilisation du consumer)

---

- ▶ On peut aussi coder son propre consumer en étendant **DefaultConsumer**
- ▶ De cette façon on peut avoir des callbacks en recodant la méthode `handleDelivery`

```
public class MyConsumer extends DefaultConsumer {  
  
    public MyConsumer(Channel channel) {  
        super(channel);  
    }  
  
    @Override  
    public void handleDelivery(String consumerTag, Envelope envelope, BasicProperties properties, byte[] body) throws IOException {  
        // fait qq chose avec le message  
    }  
}
```

# Code du consommateur complet

```
public class Recv {  
  
    private final static String QUEUE_NAME = "hello";  
  
    public static void main(String[] argv) throws Exception {  
  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
        Connection connection = factory.newConnection();  
        Channel channel = connection.createChannel();  
  
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");  
  
        QueueingConsumer consumer = new QueueingConsumer(channel);  
        channel.basicConsume(QUEUE_NAME, true, consumer);  
  
        while (true) {  
            QueueingConsumer.Delivery delivery = consumer.nextDelivery();  
            String message = new String(delivery.getBody());  
            System.out.println(" [x] Received '" + message + "'");  
        }  
    }  
}
```

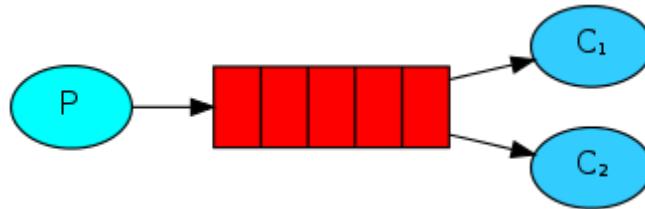


## Exemple de load balancing

# Load balancing

---

- ▶ On veux utiliser plusieurs consommateurs pour traiter les même messages.
- ▶ Les messages sont consommés *une fois par un des consommateurs* à tour de rôle
  - ▶ Répartition de la charge



# Gestion des pannes des consommateurs

- ▶ Un consommateur à récupéré des messages à traiter mais a planté avant d'avoir terminé. Que faire ?
- ▶ On commence par désactiver l'acquittement automatique:

```
boolean autoAck = false;  
channel.basicConsume("hello", autoAck, consumer);
```

Pas d'acquittements automatique

- ▶ Quand le message est traité on l'acquitte :

```
channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
```

identifiant du message

valide juste le message (false) ? ou tout les messages avant celui-ci (true) ?

# Gestion de la persistance

- ▶ Et si le serveur plante ?

```
boolean durable = true;  
channel.queueDeclare("task_queue", durable, false, false, null);
```

Demande au serveur de stocker les messages non transmis et de les retrouver au redémarrage

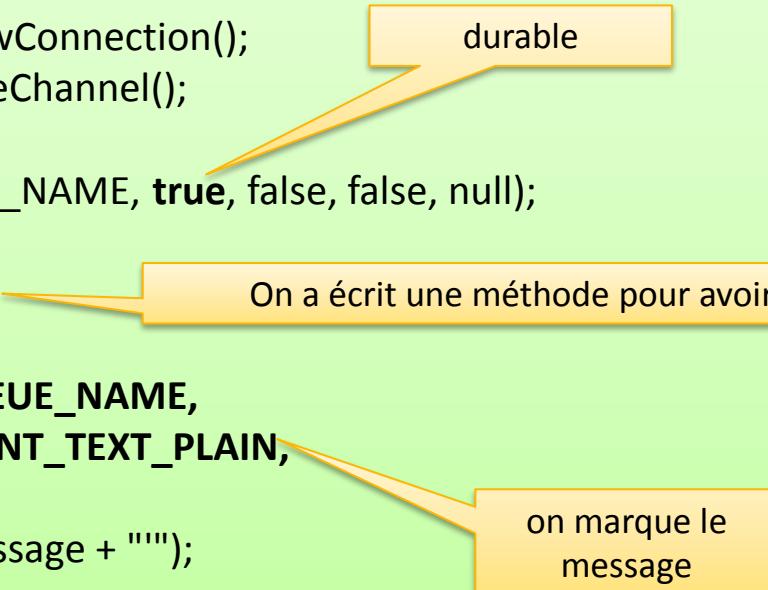
- ▶ Il faut marquer les messages comme persistants à l'envoie

```
channel.basicPublish("", "task_queue", MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());
```

On ajoute une propriété pour marquer le message

# Code du producteur

```
private static final String TASK_QUEUE_NAME = "task_queue";  
  
public static void main(String[] argv) throws Exception {  
  
    ConnectionFactory factory = new ConnectionFactory();  
    factory.setHost("localhost");  
    Connection connection = factory.newConnection();  
    Channel channel = connection.createChannel();  
  
    channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);  
  
    String message = getMessage(argv);  
    On a écrit une méthode pour avoir le message  
  
    channel.basicPublish( "", TASK_QUEUE_NAME,  
        MessageProperties.PERSISTENT_TEXT_PLAIN,  
        message.getBytes());  
    System.out.println(" [x] Sent " + message + "");  
  
    channel.close();  
    connection.close();  
}
```



- An arrow points from the word **true** in the queueDeclare line to a yellow callout box containing the word **durable**.
- An arrow points from the variable **message** in the **getMessage(argv)** call to a yellow callout box containing the text **On a écrit une méthode pour avoir le message**.
- An arrow points from the **getBytes()** method call in the basicPublish line to a yellow callout box containing the text **on marque le message**.

# Côté du consommateur

```
public static void main(String[] argv) throws Exception {  
  
    ConnectionFactory factory = new ConnectionFactory();  
    factory.setHost("localhost");  
    Connection connection = factory.newConnection();  
    Channel channel = connection.createChannel();  
  
    channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);  
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");  
  
    channel.basicQos(1);  
  
    QueueingConsumer consumer = new QueueingConsumer(channel);  
    channel.basicConsume(TASK_QUEUE_NAME, false, consumer);  
  
    while (true) {  
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();  
        String message = new String(delivery.getBody());  
  
        System.out.println(" [x] Received '" + message + "'");  
        doWork(message);  
        System.out.println(" [x] Done");  
  
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);  
    }  
}
```

on demande à recevoir les messages un par un pas de nouveau message tant que pas acquitté

on désactive l'acquittement automatique

on acquitte

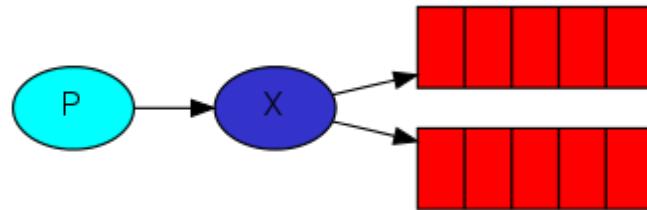


# Publish/Subscribe

# Notion d'exchange

---

- ▶ Dans la pratique le producteur envoie ses messages à un exchange (le X) qui se charge de rediriger vers les bonnes files. Jusqu'à présent, on a utilisé l'exchange par défaut "".



- ▶ C'est l'exchange qui se charge du filtrage/routage des messages :
  - ▶ envoyer vers toutes les files ou une seule ?
  - ▶ effacer le message ?
  - ▶ ...

# Déclaration d'un exchange

---

## ► Il faut déclarer les exchanges :

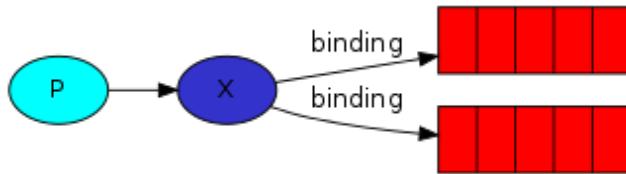


## ► Plusieurs politiques de distribution :

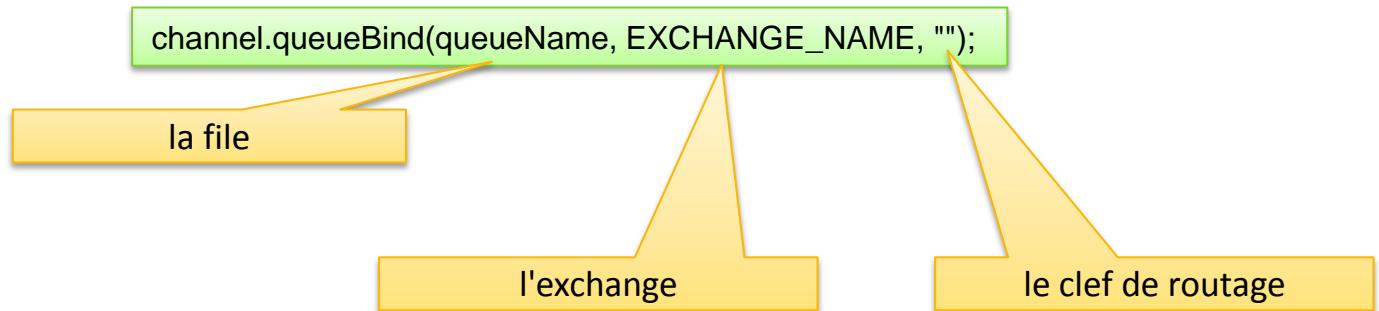
- ▶ **fanout** : envoie à toutes les files
- ▶ **direct** : envoie en faisant du routage sur la `routing_key`. Seule les files dont la liaison (**binding**) correspond exactement à cette clef recoivent.
- ▶ **topic** : même principe mais utilise des topics pour faire du routage.

# Notion de binding

- ▶ Les exchanges sont liés au file par des bindings :



- ▶ Le binding explique ce qui doit être envoyé dans la file :



# Code du producteur

```
public static void main(String[] argv) throws Exception {  
  
    ConnectionFactory factory = new ConnectionFactory();  
    factory.setHost("localhost");  
    Connection connection = factory.newConnection();  
    Channel channel = connection.createChannel();  
  
    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");  
  
    String message = getMessage(argv);  
  
    channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());  
    System.out.println(" [x] Sent '" + message + "'");  
  
    channel.close();  
    connection.close();  
}
```

on déclare un exchange

on envoie les messages  
sur l'exchange

on ne spécifie pas la file, on  
utilisera des files temporaires

# Code du consommateur

```
private static final String EXCHANGE_NAME = "logs";

public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
    String queueName = channel.queueDeclare().getQueue();
    channel.queueBind(queueName, EXCHANGE_NAME, "");

    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    QueueingConsumer consumer = new QueueingConsumer(channel);
    channel.basicConsume(queueName, true, consumer);

    while (true) {
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        String message = new String(delivery.getBody());
        System.out.println(" [x] Received '" + message + "'");
    }
}
```

on déclare un exchange

on déclare une file temporaire  
pour la réception

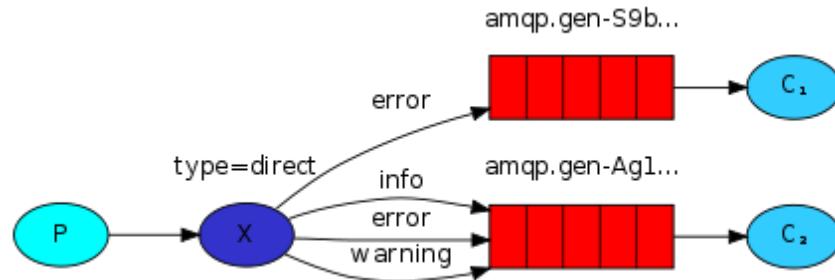
on attache la file à l'exchange



# Routage "direct"

# Utilisation de clefs de routages

- On veux router les messages en fonction de clefs de routages



- L'utilisation de message se fait avec des exchanges en mode direct :

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
```

# Envoyer/Recevoir des messages avec des clefs

- ▶ Il faut préciser la clef à l'émission (coté producteur)

```
channel.basicPublish(EXCHANGE_NAME, "warning", null, message.getBytes());
```

on précise la clef lors de  
l'émission

- ▶ Il faut s'abonner avec la clef dans le binding (coté consommateur)

```
channel.queueBind(queueName, EXCHANGE_NAME, "warning");
```

on précise la clef dans le  
binding

# Code du producteur

```
public static void main(String[] argv) throws Exception {
```

```
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();
```

```
    channel.exchangeDeclare(EXCHANGE_NAME, "direct");
```

```
    String severity = getSeverity(argv);
    String message = getMessage(argv);
```

```
    channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes());
    System.out.println(" [x] Sent '" + severity + ":" + message + "'");
```

```
    channel.close();
    connection.close();
}
```

on demande du routage mode  
"direct"

on récupère la sévérité et le  
message des arguments du main

on précise la clef de routage

# Côté consommateur

```
private static final String EXCHANGE_NAME = "direct_logs";

public static void main(String[] argv) throws Exception {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE_NAME, "direct");
String queueName = channel.queueDeclare().getQueue();

if (argv.length < 1){
    System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");
    System.exit(1);
}

for(String severity : argv){
    channel.queueBind(queueName, EXCHANGE_NAME, severity);
}

System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(queueName, true, consumer);

while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());
    String routingKey = delivery.getEnvelope().getRoutingKey();
    System.out.println(" [x] Received '" + routingKey + "'::" + message + "'");
}
}
```

on demande du routage mode "direct"

on déclare une file temporaire pour la réception

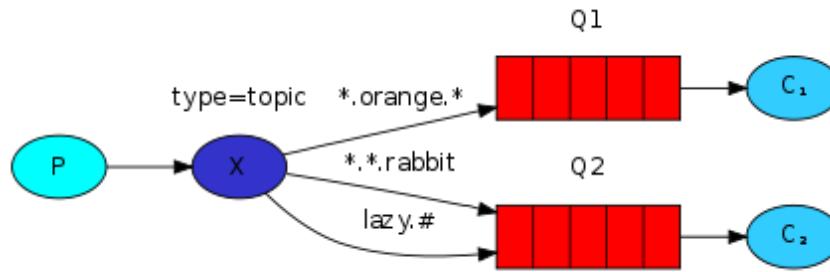
on s'abonne aux clefs passées en arguments dans main



## Routage "topics"

# Même chose mais avec des topics

- ▶ Le topic est une clefs particulière



- ▶ Des clefs organisées hiérarchiquement
- ▶ On peut utiliser des wildcards pour s'abonner :
  - ▶ \* se substitue à un seul mot (il en faut un)
    - ▶ orange.\* => orange.foo, orange.bar mais pas orange.foo.bar
  - ▶ # se substitue à zéro ou plusieurs mots
    - ▶ orange.# => orange.foo, orange.bar, orange.foo.bar, orange.foo.foo.bar, ....

# Côté producteur

```
public static void main(String[] argv) {  
    Connection connection = null;  
    Channel channel = null;  
    try {  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
  
        connection = factory.newConnection();  
        channel = connection.createChannel();  
  
        channel.exchangeDeclare(EXCHANGE_NAME, "topic");  
  
        String routingKey = getRouting(argv);  
        String message = getMessage(argv);  
  
        channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());  
        System.out.println(" [x] Sent " + routingKey + ":" + message + "");  
  
    }  
    catch (Exception e) {  
        e.printStackTrace();  
    }  
    finally {  
        if (connection != null) {  
            try {  
                connection.close();  
            }  
            catch (Exception ignore) {}  
        }  
    }  
}
```

on demande du routage mode  
"topic"

on envoie sur le topic donné

gestion des exceptions

Le topic et le message sont  
passés en argument

# Côté consommateur

```
ConnectionFactory factory = new ConnectionFactory();
factory.setHost("localhost");
connection = factory.newConnection();
channel = connection.createChannel();

channel.exchangeDeclare(EXCHANGE_NAME, "topic");
String queueName = channel.queueDeclare().getQueue();

if (argv.length < 1){
    System.err.println("Usage: ReceiveLogsTopic [binding_key]...");
    System.exit(1);
}

for(String bindingKey : argv){
    channel.queueBind(queueName, EXCHANGE_NAME, bindingKey);
}

System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume(queueName, true, consumer);

while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());
    String routingKey = delivery.getEnvelope().getRoutingKey();
    System.out.println(" [x] Received " + routingKey + ":" + message + ")");
}
```

on demande du routage mode  
"direct"

on s'abonne aux topics passées en  
arguments dans main

on passe la liste des topics auxquels  
on s'abonne en paramètre

\$ java -cp \$CP ReceiveLogsTopic "kern.\*" "\*critical"