

Communication par événements et bus à messages



Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)
<http://sardes.inrialpes.fr/~krakowia>

Motivations

- **Le modèle client-serveur ne répond pas à tous les besoins**
 - ◆ Le schéma de base est **synchrone**
 - ◆ La communication est essentiellement 1 vers 1 (ou n vers 1)
 - ◆ Les entités (clients, serveurs) sont **désignées explicitement**
 - ◆ L'organisation de l'application est **peu dynamique**

- **Les événements et bus à messages fournissent un modèle adapté à des besoins particuliers**
 - ◆ Communication **asynchrone**
 - ◆ Communication possible **n vers p**
 - ◆ Possibilité de **désignation non explicite** des entités
 - ◆ Organisation **dynamique** des applications (facilité d'évolution, adjonction et retrait d'entités participantes)

Plan de la présentation

- **Exemple introductif**
- **Modèles de communication asynchrone**
 - ◆ Files de messages
 - ◆ Événements (*publish-subscribe*)
- **Exemples**
 - ◆ JMS (*Java Message Service*)
 - ◆ Le service d'événements de CORBA
 - ◆ Autres réalisations (*Message Oriented Middleware*)

Exemple : administration d'un réseau

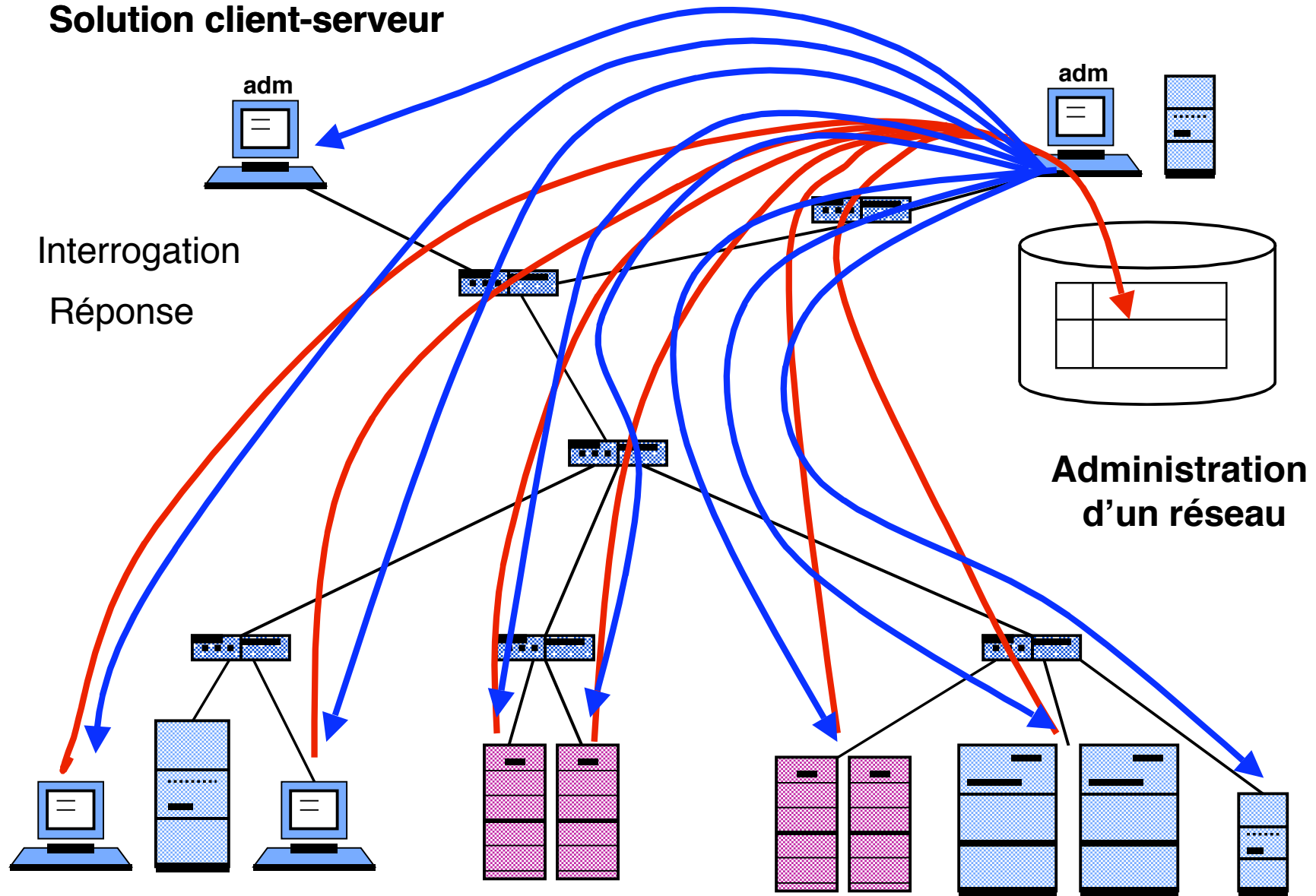
(surveillance des équipements)

■ **Problème**

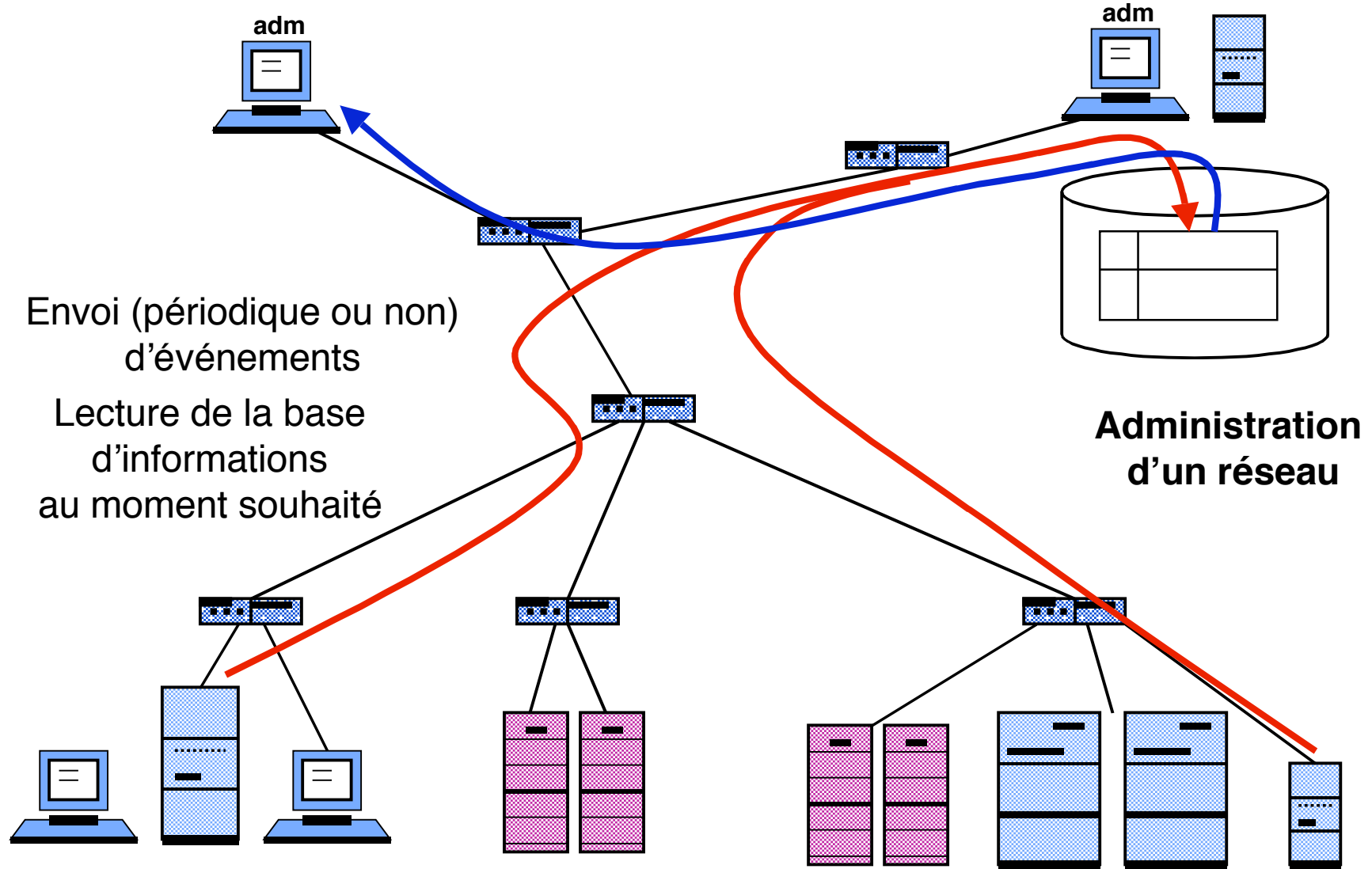
- ◆ **Surveillance de l'état de machines, systèmes et applications dans un environnement distribué**
- ◆ **Flot permanent de données en provenance de sources diverses sur le réseau**
- ◆ **Modification permanente possible (ajout, suppression, déplacement des équipements)**
- ◆ **Possibilité d'accès des administrateurs depuis n'importe quel poste de travail**

Source : M. Riveill, A. Freyssinet, Modèles à bus de messages
École "Construction d'Applications Réparties", 1999

Solution client-serveur



Solution bus à messages



Solution bus à messages

■ Propriétés caractéristiques

- ◆ Les différents éléments administrés émettent des messages déclenchés par des événements
 - ❖ horloge (surveillance périodique)
 - ❖ changement d'état ou de configuration
 - ❖ alertes
- ◆ Un ou plusieurs processus cycliques (démons) reçoivent ces notifications et maintiennent l'état courant du système
 - ❖ suivi des changements de configuration dynamiques
 - ❖ émission de messages signalant les changements d'état et les mises à jour
 - ❖ statistiques, journal de fonctionnement

Modèles à messages et événements

■ Messages et événements : deux notions voisines

◆ Aspects communs

❖ Asynchronisme

❖ Désignation explicite ou non des destinataires

- ▲ Diffusion sélective

- ▲ Groupes

◆ Différences

❖ Interface de programmation (API)

- ▲ Primitives d'utilisation

- ▲ Ajout et retrait de participants

- ▲ Association entre notification et traitement

Communication par messages

■ Principes directeurs

- ◆ Communication asynchrone
- ◆ Désignation du destinataire
 - ❖ directe
 - ❖ indirecte (via porte, file de messages)
- ◆ Structuration des messages
 - ❖ messages éventuellement typés

■ Interface de programmation

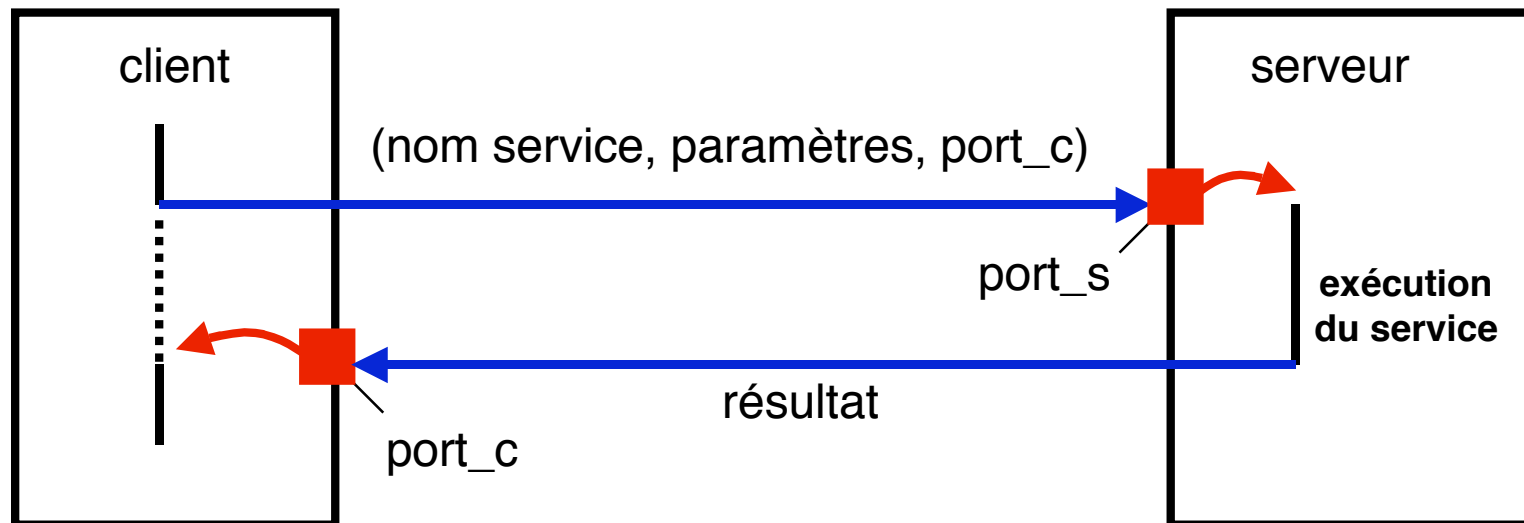
- ◆ primitives de base : envoyer - recevoir (*send, receive*)
- ◆ extensions : groupes, désignation associative

■ Mise en œuvre, outils

- ◆ interface *socket* sur niveau de transport, IP multicast
- ◆ outils de développement encore peu évolués

Communication par messages : utilisation (1)

- **Utilisation de messages pour réaliser un schéma client-serveur**
 - ◆ cf réalisation du client-serveur avec des *sockets*

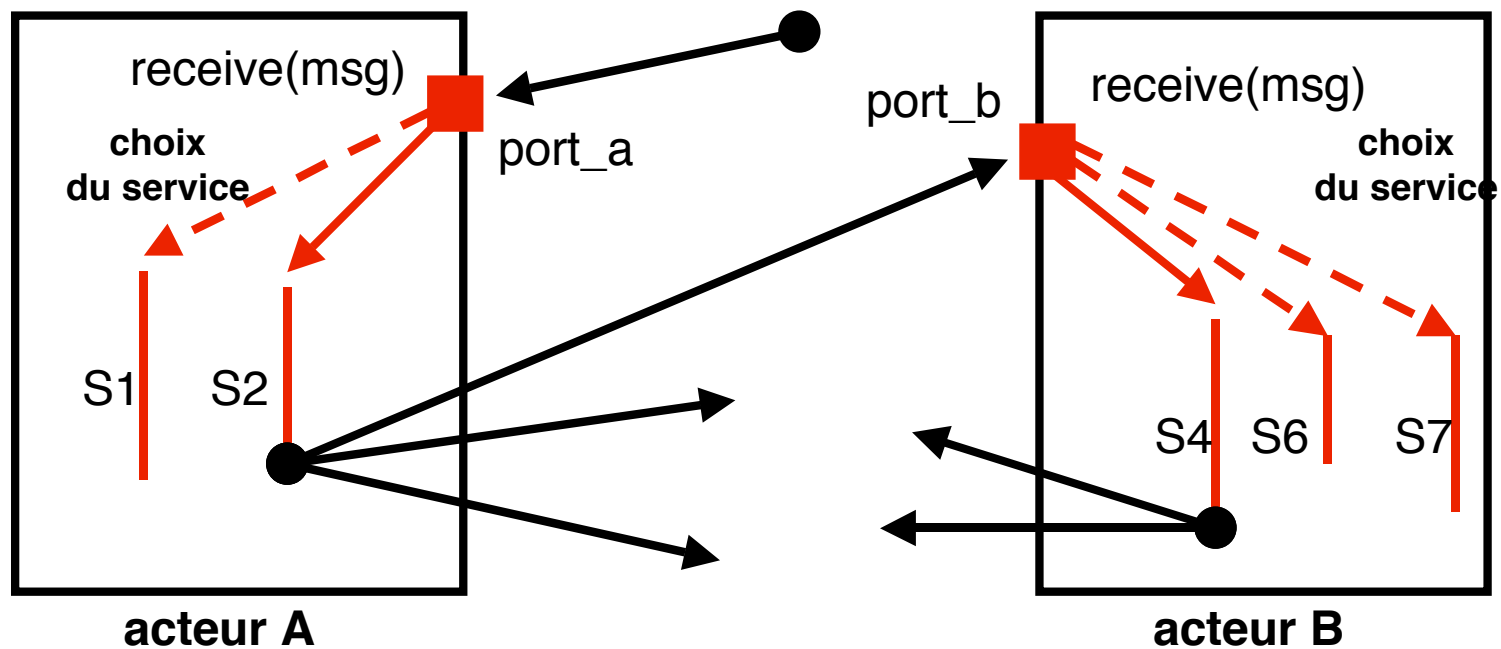


Problèmes

- sécurité (n'importe qui peut envoyer un message sur port_s, port_c)
- gestion des erreurs (cf RPC)

Communication par messages : utilisation (2)

- **“Acteur”** : action associée à la réception d’un message, pouvant déclencher l’envoi d’autres messages



Problèmes : absence de structuration, gestion des erreurs

Communication par messages : extensions

■ Communication de groupe

- ◆ groupe = ensemble de destinataires désignés par nom unique
- ◆ gestion dynamique (arrivée - départ ou défaillance de membres)
- ◆ problème (difficile !) : **maintien de la cohérence** (vue cohérente de la composition du groupe, ordre causal de réception, etc.)
- ◆ utilité : tolérance aux fautes, travail coopératif
- ◆ on en reparlera à propos de tolérance aux fautes

■ Communication anonyme

- ◆ les destinataires d'un message sont identifiés par leurs propriétés, non par leur nom
- ◆ propriétés
 - ❖ définies par un attribut du message (filtrage)
 - ❖ définies de manière externe
- ◆ indépendance entre émetteurs et récepteurs (cf événements)

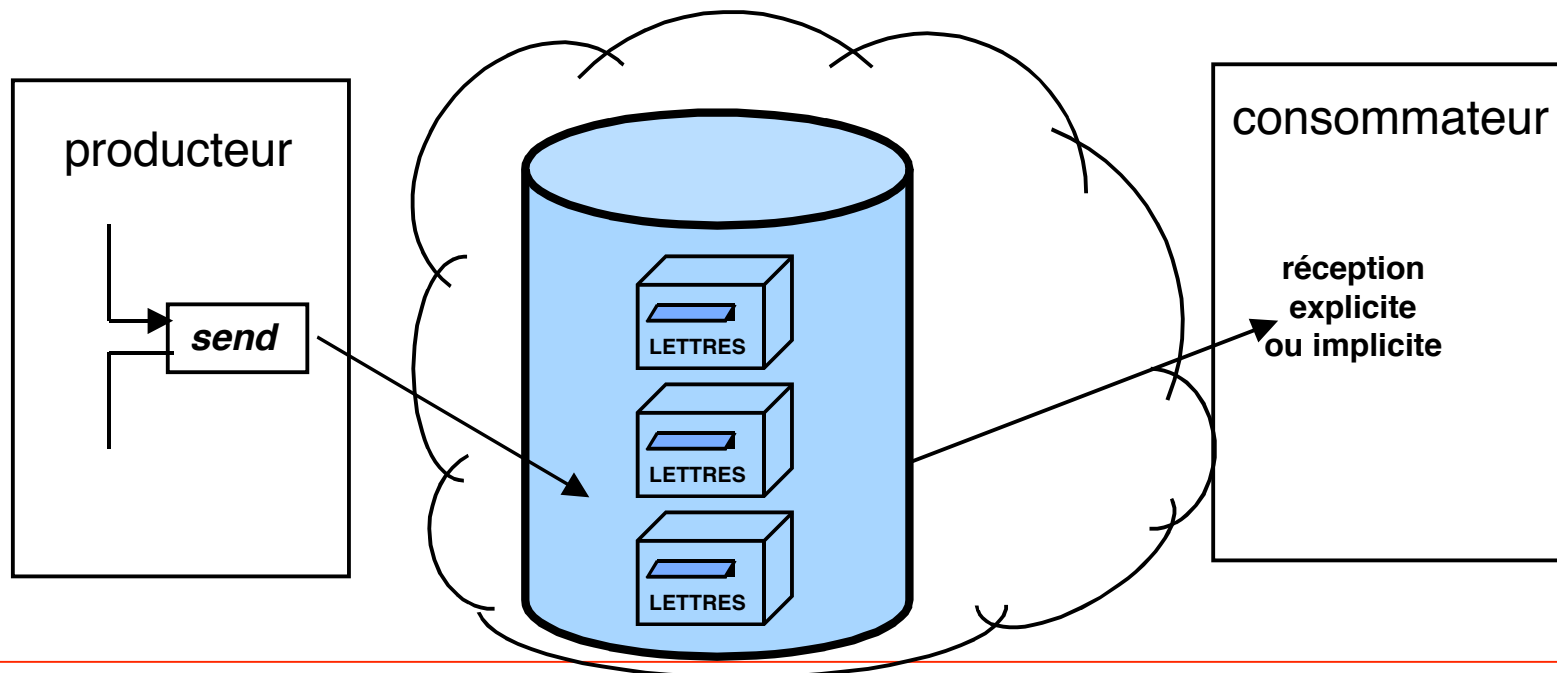
Bus logiciel à messages

■ Messages

- ◆ identification unique
- ◆ typés ou non
- ◆ persistants ou non
- ◆ délivrés avec ou sans accusé de réception

■ Queues de messages

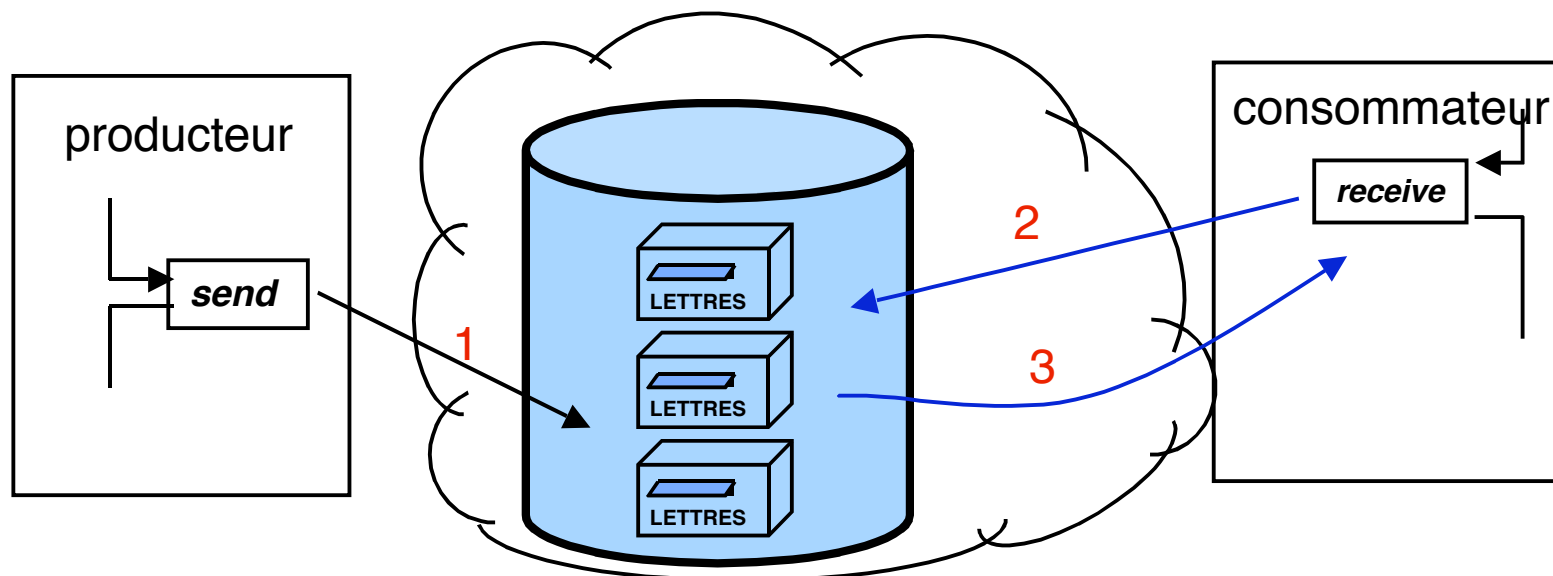
- ◆ identification unique
- ◆ partagées par les applications
- ◆ persistantes
- ◆ ordre spécifié (total, causal, selon priorité, etc.)



Bus logiciel à messages : réception synchrone (pull)

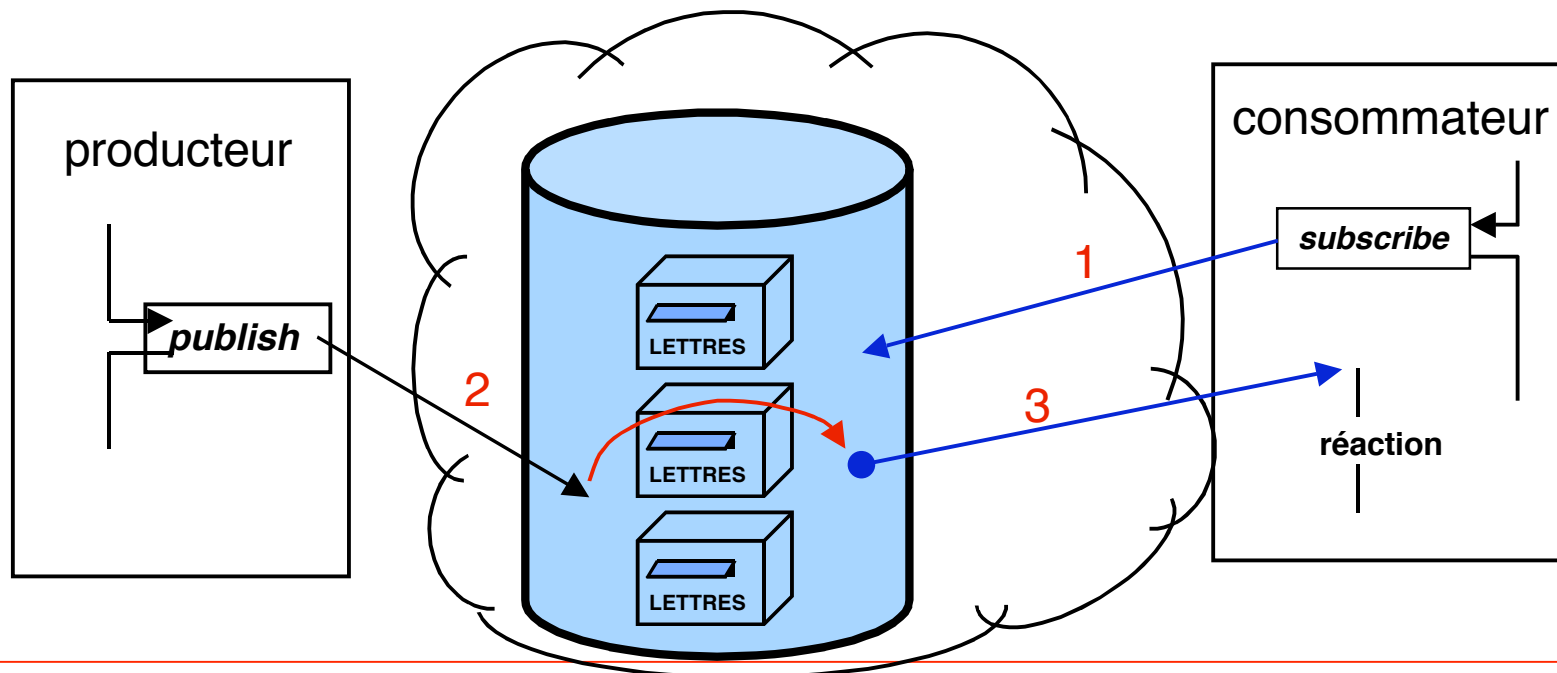
■ Réception explicite des messages (retrait)

- ◆ les clients viennent périodiquement prendre leurs messages dans une boîte aux lettres
- ◆ le consommateur n'est pas nécessairement bloqué en l'absence de message
- ◆ point à point



Bus logiciel à messages : réception asynchrone (*push*)

- 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/sont informé(s) et exécute une réaction prédéfinie (comportant la lecture du message)
 - ◆ c'est en fait un modèle "événement-réaction" cf plus loin



Bus logiciel à messages : exemples d'API

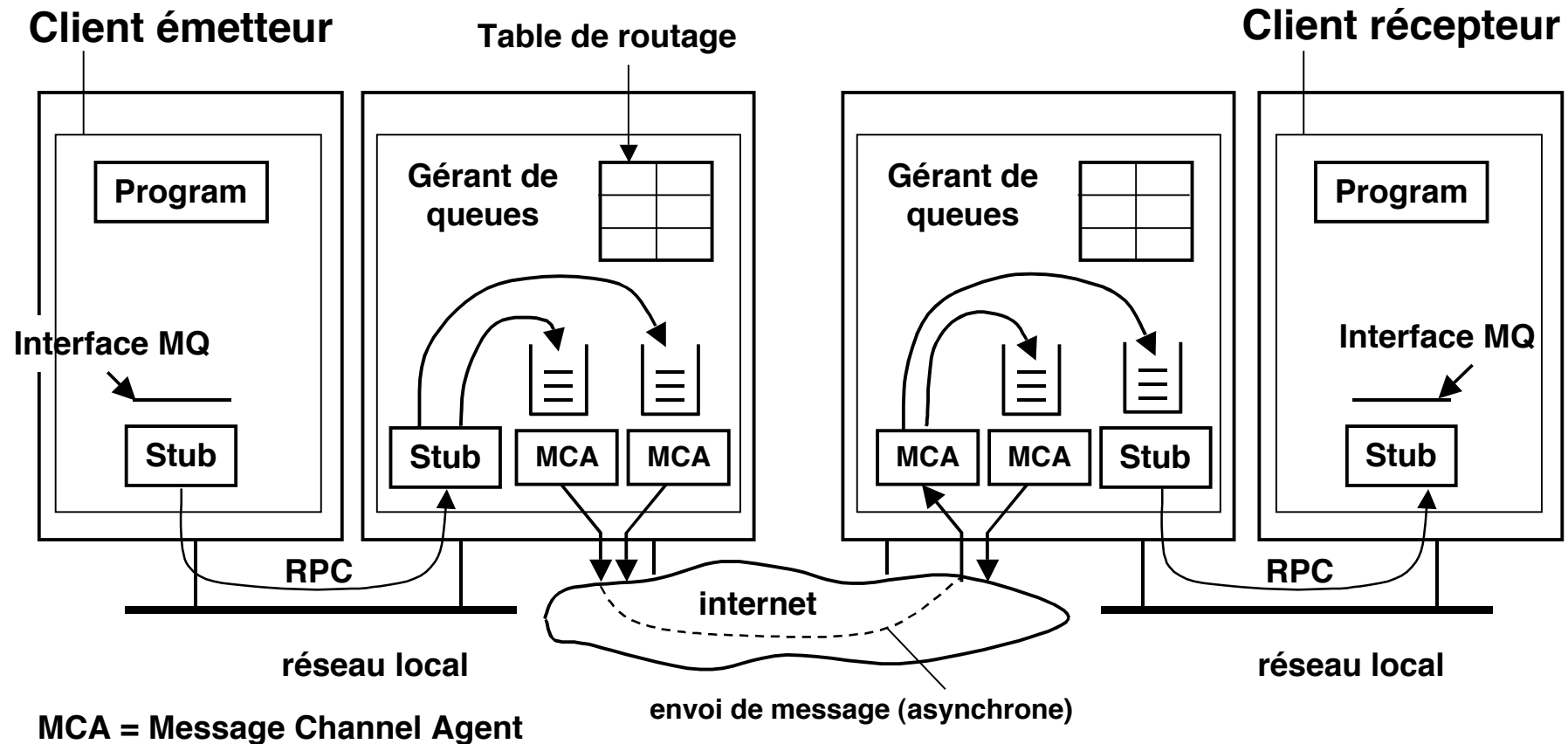
Pull

```
// créer et lier une file
msgQ:=new MsgQ(name, properties)
// produire un message
sendQ:=msgQ.attachSender(type)
sendQ.sendMsg(msg)
// consommer un message
recvQ:=msgQ.attachRecv(type)
msg:=recvQ.recvMsg(wait, select)
recvQ.confirmMsg(msg)
// délier une file
msgQ.detach()
```

Push

```
// créer et lier une file
topic=new Topic(name, properties)
// produire un message
pub:=topic.createPub()
pub.publish(msg)
// consommer un message
sub:=topic.createSub()
sub.subscribe(msg_reaction)
// délier une file
topic.detach()
```


Bus à messages : exemple IBM MQSeries



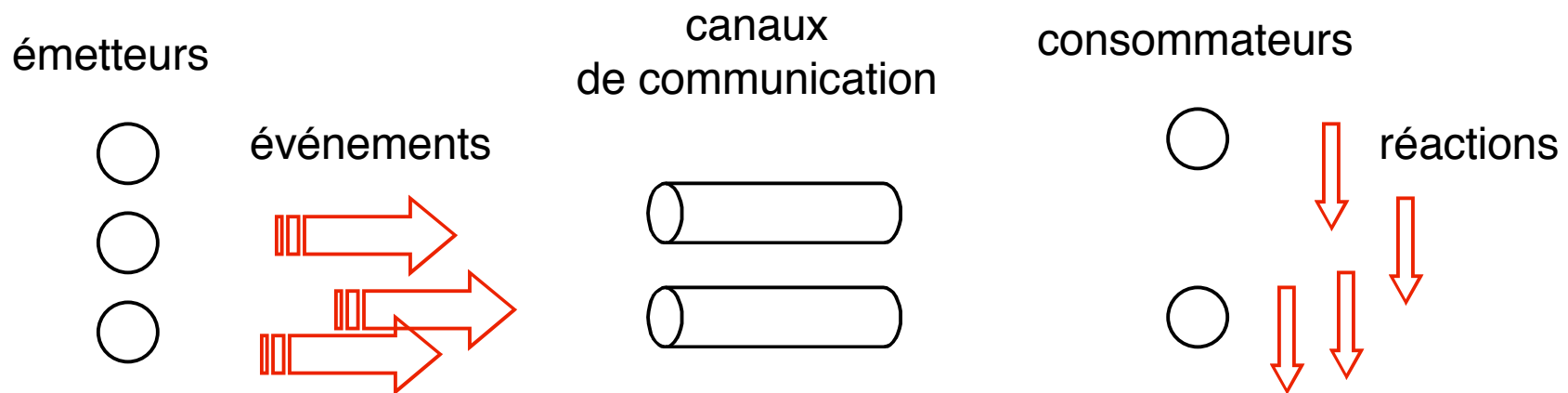
NB Pour des raisons de place, on n'a représenté que des files en émission sur l'émetteur, des files en réception sur le récepteur. En fait les 2 sortes de files sont sur tous les gérants.

Source : Tanenbaum & van Steen, Distributed Systems, Prentice Hall, 2002

Communication par événements : principes de base

■ Événements et réactions

- ◆ événement = changement d'état survenant de manière asynchrone (par rapport à ses "consommateurs")
- ◆ réaction = exécution d'une séquence prédéfinie liée à l'événement
- ◆ association dynamique événement-réaction
- ◆ communication anonyme : indépendance entre émetteur et "consommateurs" d'un événement



Communication par événements : interface

Pas d'interface normalisée, donc on donne une idée des opérations usuelles d'un service d'événements (exemples concrets plus loin)

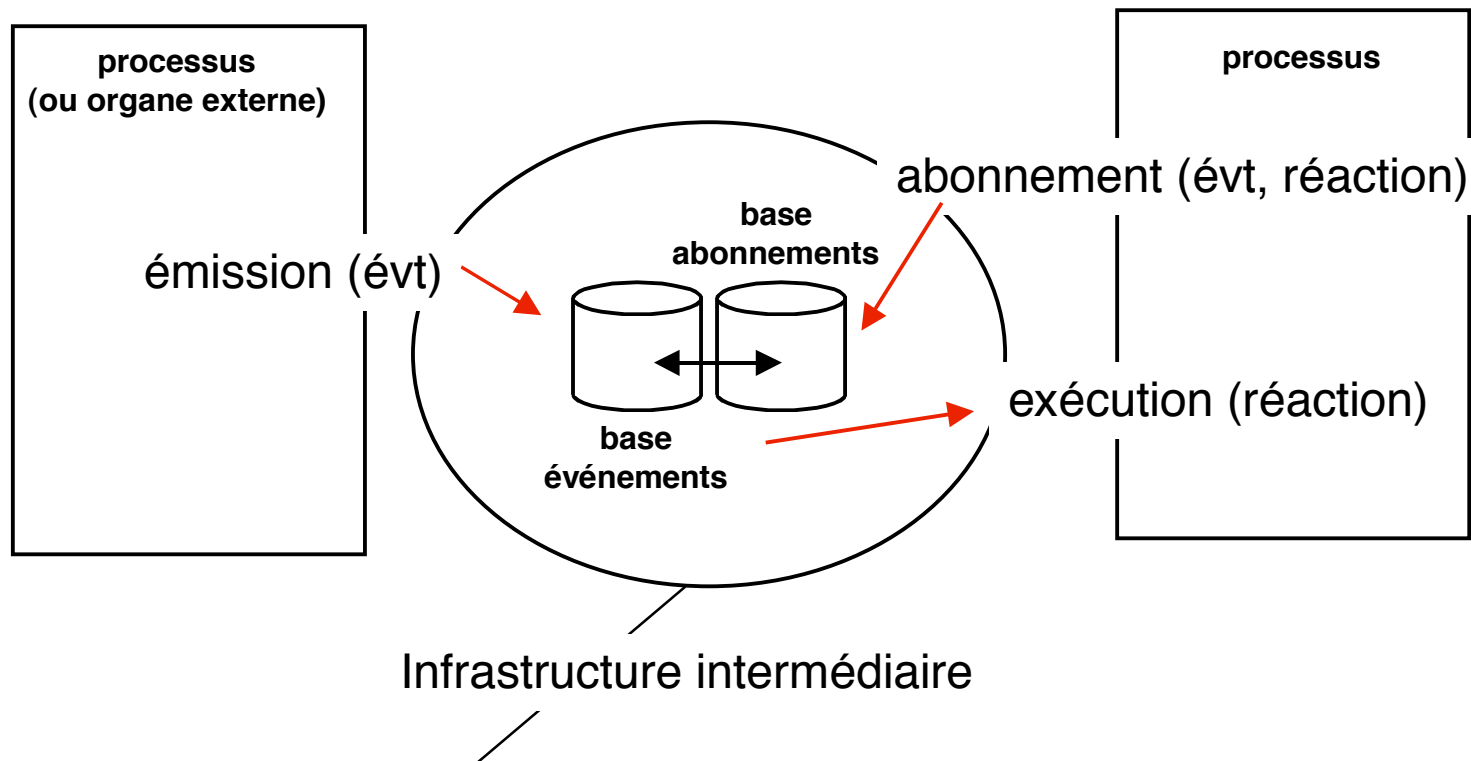
abonnement (sujet1, sujet2, ...)
désabonnement (sujet1, sujet2, ...)

associer (sujet1, réaction1 [paramètres])
dissocier (sujet1)

déclaration de types d'événements : définition des champs
(sujet, attributs, ...)

créer (événement [paramètres])
envoyer (événement)

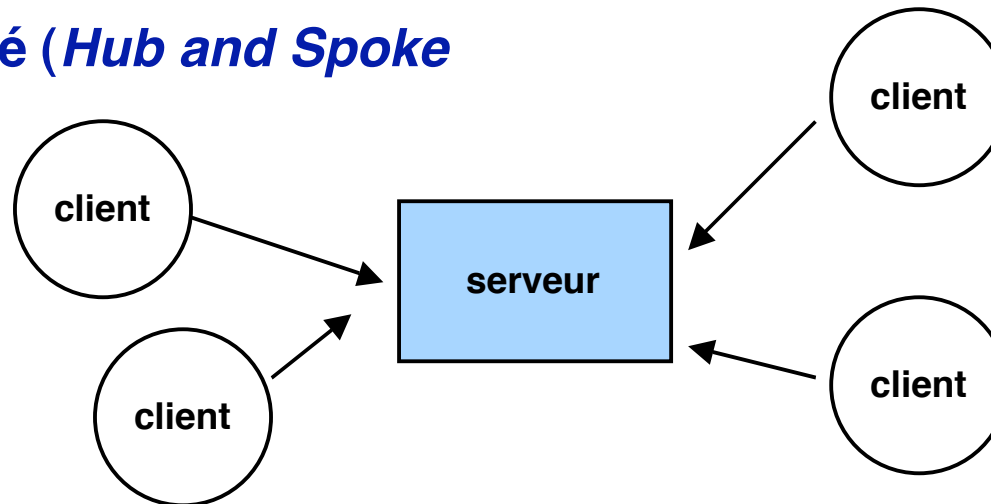
Communication par événements : mise en œuvre (1)



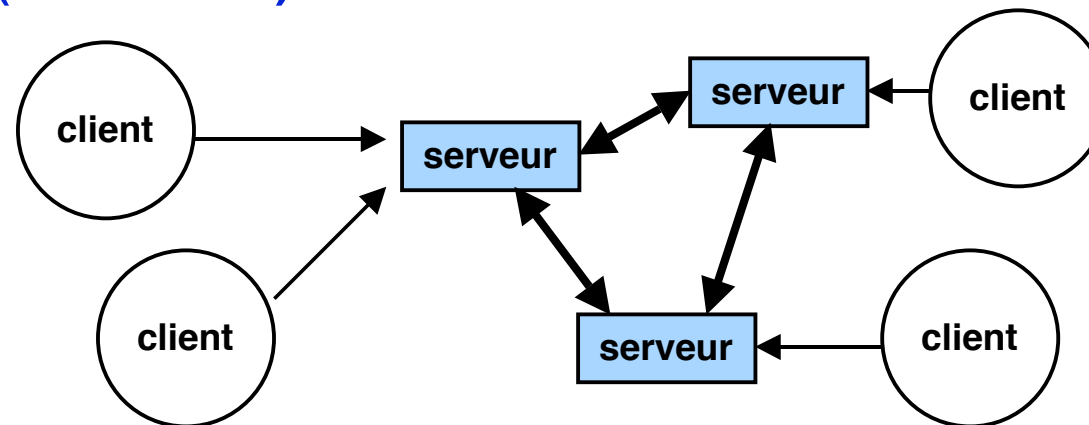
- **Serveur centralisé (*Hub and Spoke*)**
- **Serveur réparti (*Snowflake*)**
- **Service réparti (bus logiciel)**

Communication par événements : mise en œuvre (2)

■ Serveur centralisé (*Hub and Spoke*)



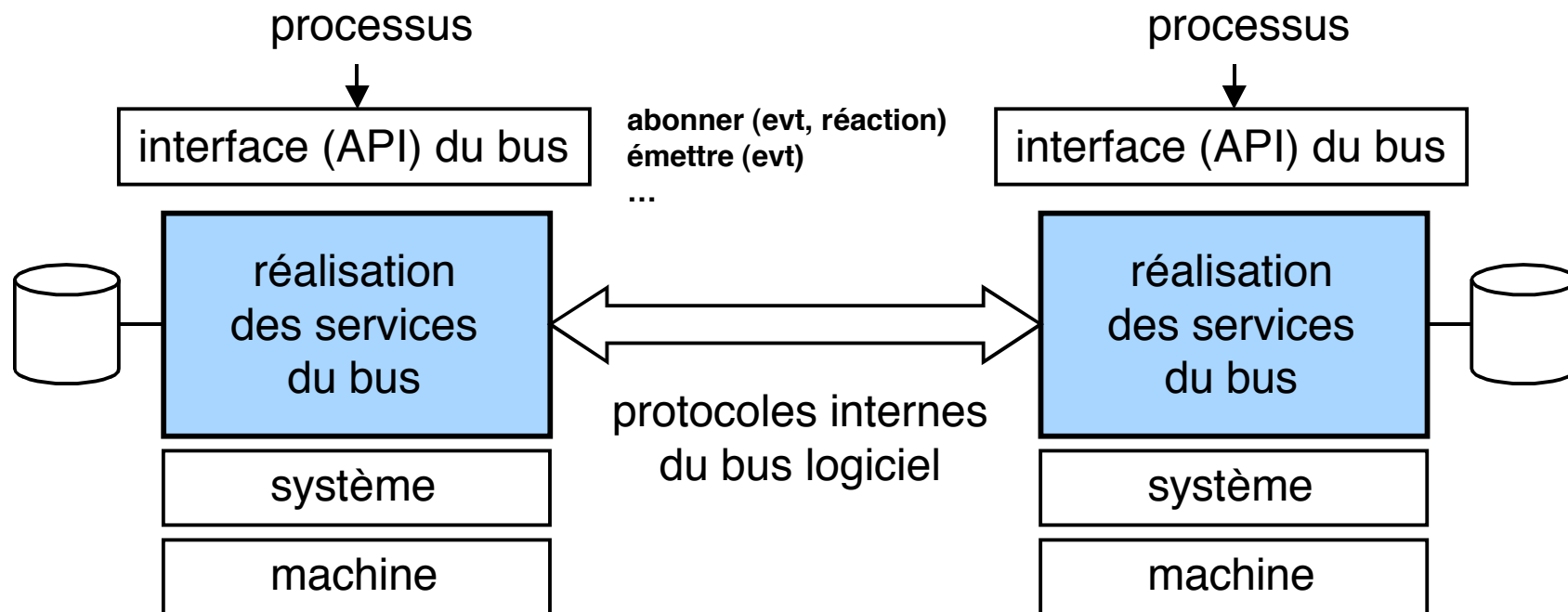
■ Serveur réparti (*Snowflake*)



Communication par événements : mise en œuvre (3)

■ Service réparti (bus logiciel)

- ◆ NB : différence entre service et serveur (les deux réalisent un ensemble de fonctions définies par une interface)
 - ❖ serveur : composant identifié
 - ❖ service : interface locale (serveur non identifié)



Exemple de système à événements : TIB/Rendez-vous

■ Principe

- ◆ Schéma *publish-subscribe* avec sujet

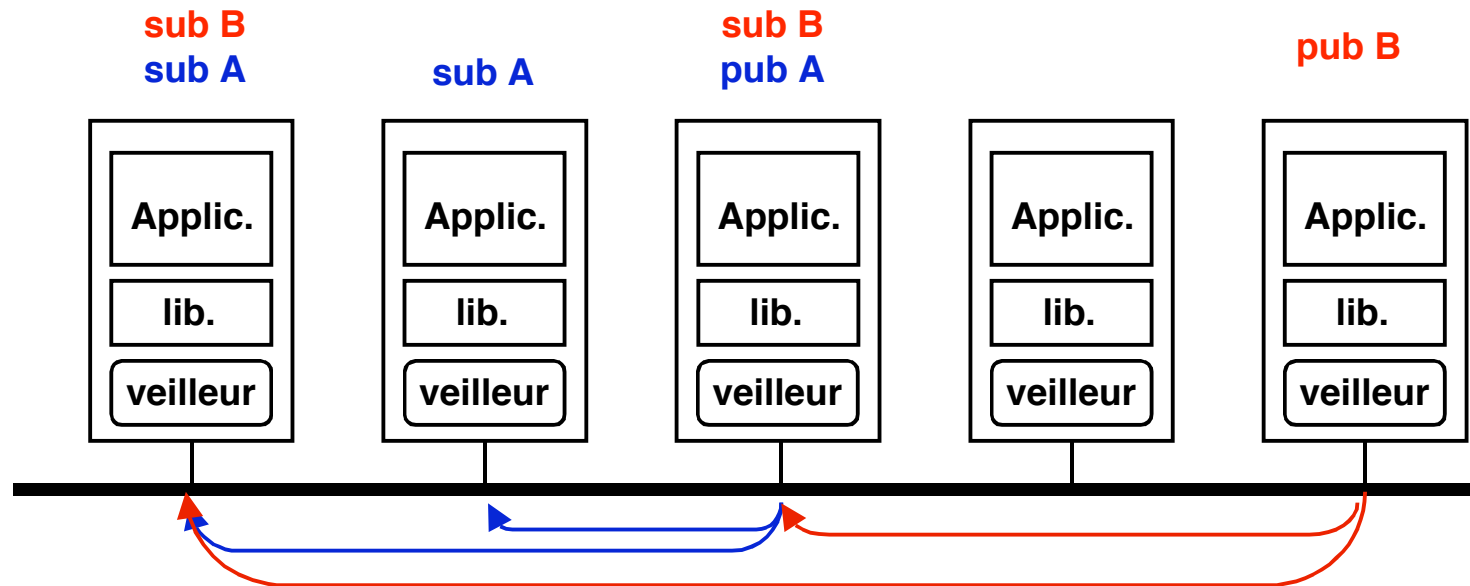
■ Réalisation

- ◆ Connexion des sites par réseau à diffusion (mais point à point possible)
- ◆ Sur un réseau local : chaque site a un veilleur (*daemon*) qui sert d'interface pour l'envoi et la réception des messages
- ◆ Sur une interconnexion de réseaux locaux : chaque réseau a un routeur qui lui permet de communiquer avec les autres réseaux. Le veilleur implanté sur le routeur sert d'interface de communication. On construit ainsi un réseau virtuel (*overlay network*) au niveau de l'application

Source : Tanenbaum & van Steen, Distributed Systems, Prentice Hall, 2002

TIB/Rendez-vous : structure de communication (1)

Réseau local



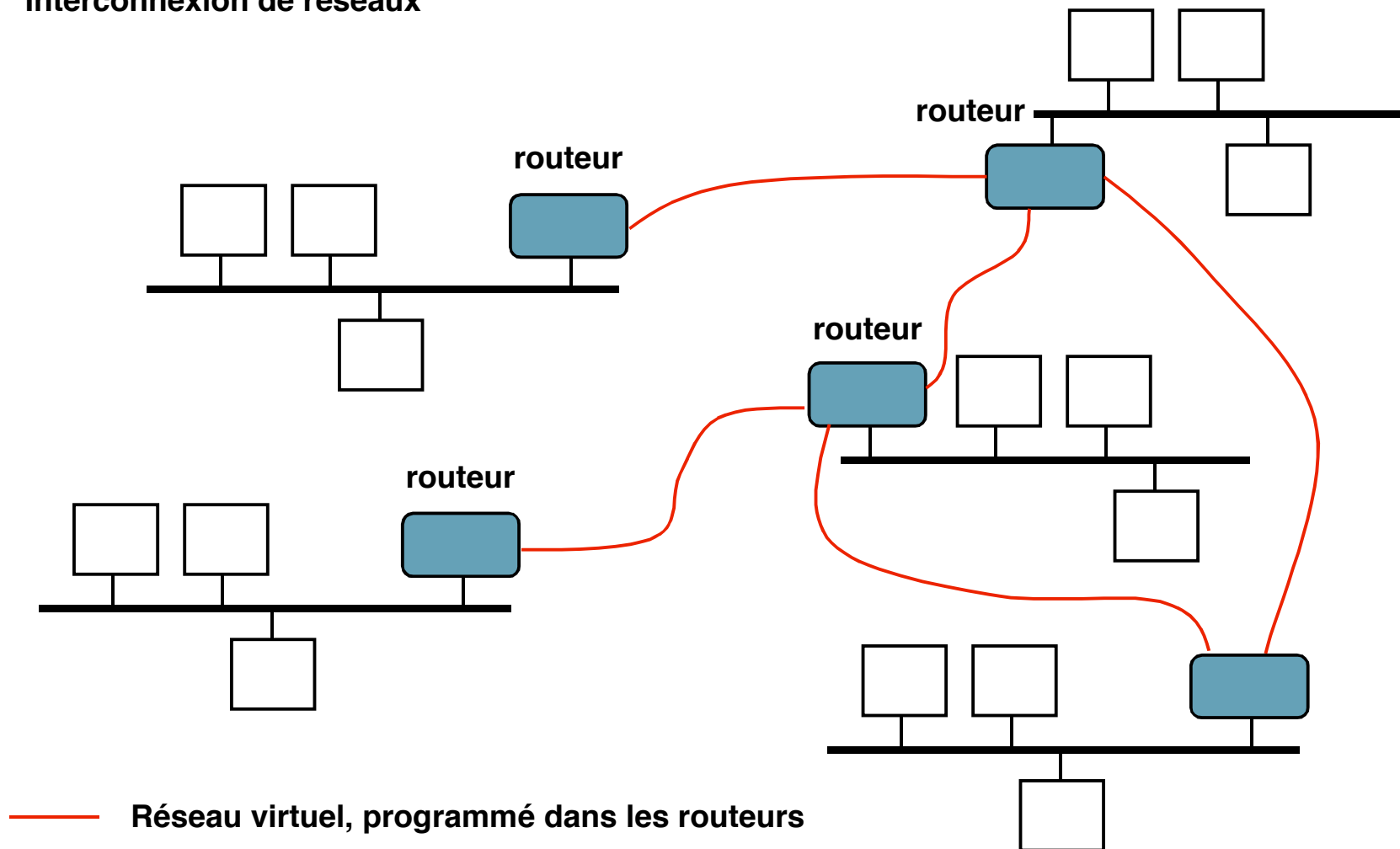
Un veilleur sur un site S entretient une table qui donne la liste des sujets auxquels a souscrit l'application sur S

Un site qui publie sur un sujet X passe le message à son veilleur qui le diffuse à tous les autres veilleurs

Chaque veilleur filtre les messages arrivants en fonction des sujets localement suscrits (le contenu de sa table)

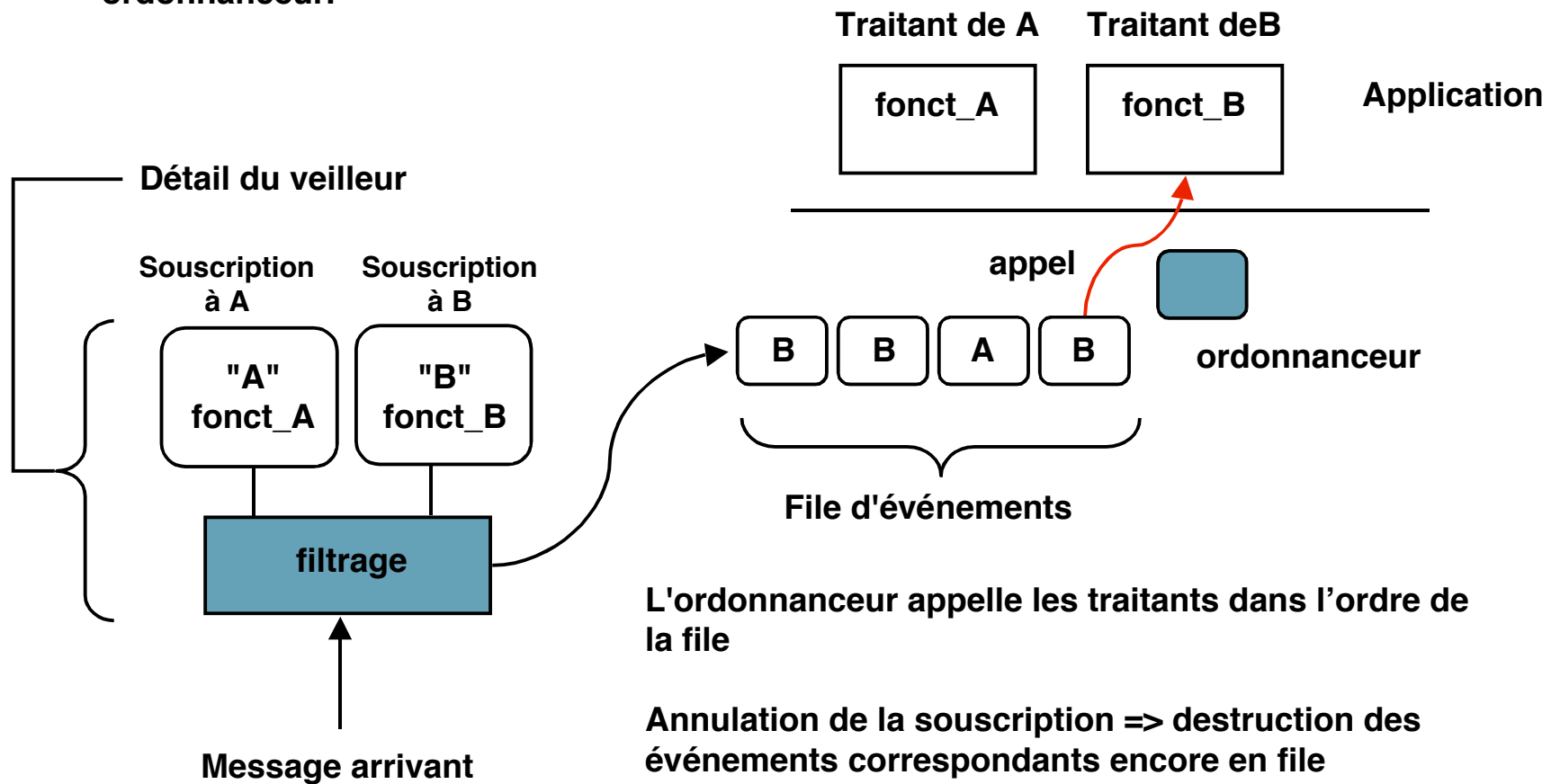
TIB/Rendez-vous : structure de communication (2)

Interconnexion de réseaux



TIB/Rendezvous : traitement des réceptions

Principe : chaque réception de message correspondant à la souscription à un sujet S déclenche l'appel (asynchrone) d'une fonction de l'application (*callback*) chargée de traiter S. L'asynchronisme est traité au moyen d'une file d'événements et d'un ordonnanceur.



Conclusion sur messages et événements (1)

■ Domaines d'application

◆ Événements

- ❖ génie logiciel (coopération entre outils de développement (SoftBench, ToolTalk, DecFuse, ...))
- ❖ *workflow* (applications à flots de données) KoalaBus, ...
- ❖ diffusion de logiciels et d'informations sur le web (iBus, CastaNet, Ambrosia, SmartSockets, ...)

◆ Messages

- ❖ Accès distant à des serveurs
- ❖ Intégration d'applications, notamment hétérogènes et de grande taille
- ❖ Applications financières (bourse, enchères)
- ❖ Administration, surveillance, applications configurables

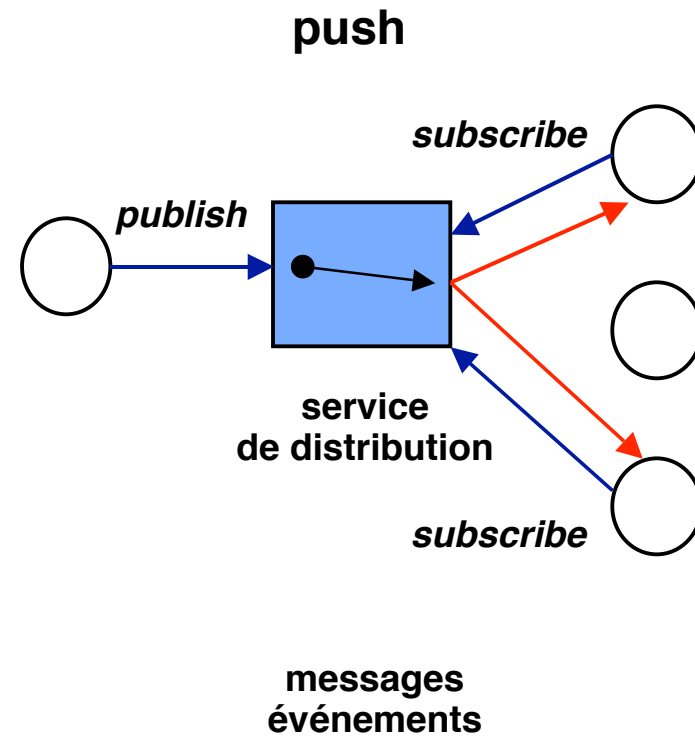
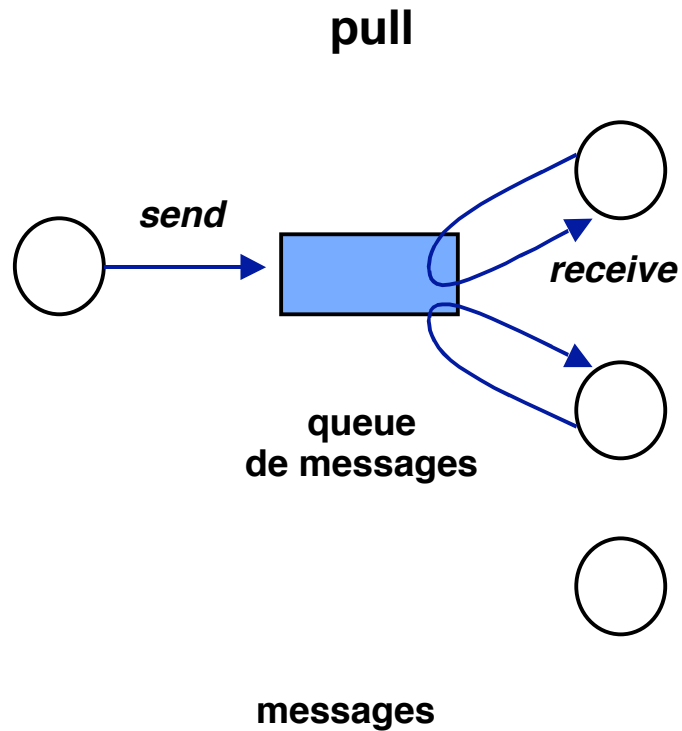
Conclusion sur messages et événements (2)

■ État du marché

- ◆ Secteur en développement
 - ❖ large domaine d'application
 - ❖ besoins d'intégration grande échelle
- ◆ Infrastructures propriétaires, mais début de normalisation
 - ❖ autour de Java, interface JMS
 - ❖ autour de CORBA : *Corba Event Service*
 - ❖ efforts de l'ECMA
- ◆ Outils de développement encore sommaires

Rappel sur bus à messages

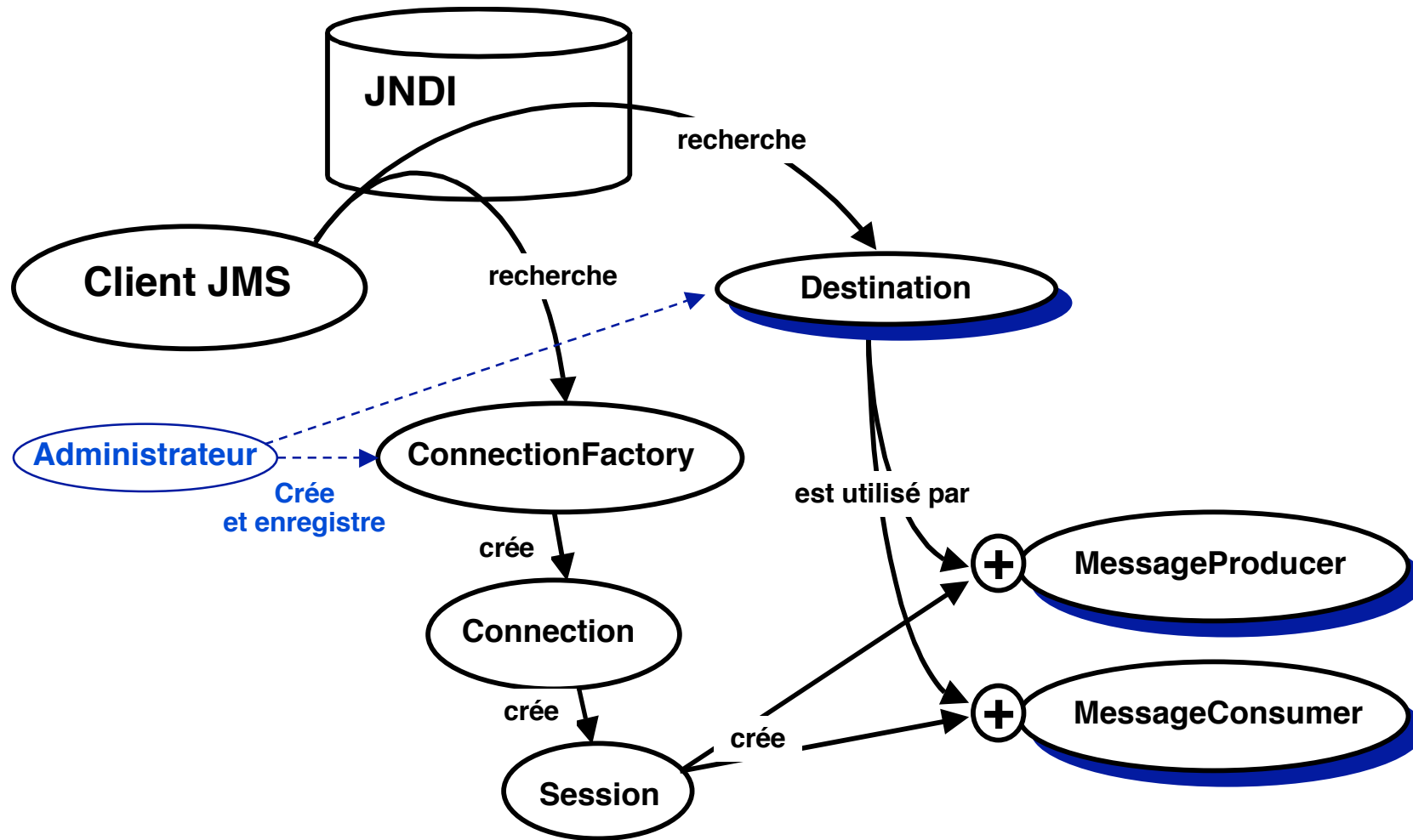
■ Deux modes de fonctionnement



Trois exemples de mise en œuvre des messages et événements

- **JMS (*Java Message Service*)**
 - ◆ Interface Java standard pour communication par message
 - ◆ Ne définit pas d'implémentation
 - ❖ Peut servir à interfacier un système existant
 - ▲ IBM MQ Series, Novell, Oracle, Sybase, Tibco
 - ❖ Des implémentations spécifiques peuvent être réalisées
 - ❖ Supporte point à point et *publish/subscribe*
- **Event Service de CORBA**
 - ◆ L'un des *Object Services* définis par l'OMG
 - ◆ Définit un ensemble de classes permettant de réaliser un service d'événements
 - ◆ Supporte les modes *push* et *pull*
- **Jini (Sun)**
 - ◆ Service de coordination à base d'objets partagés (Java)
 - ◆ Utilisé pour des environnements dynamiquement variables (découverte de ressources)

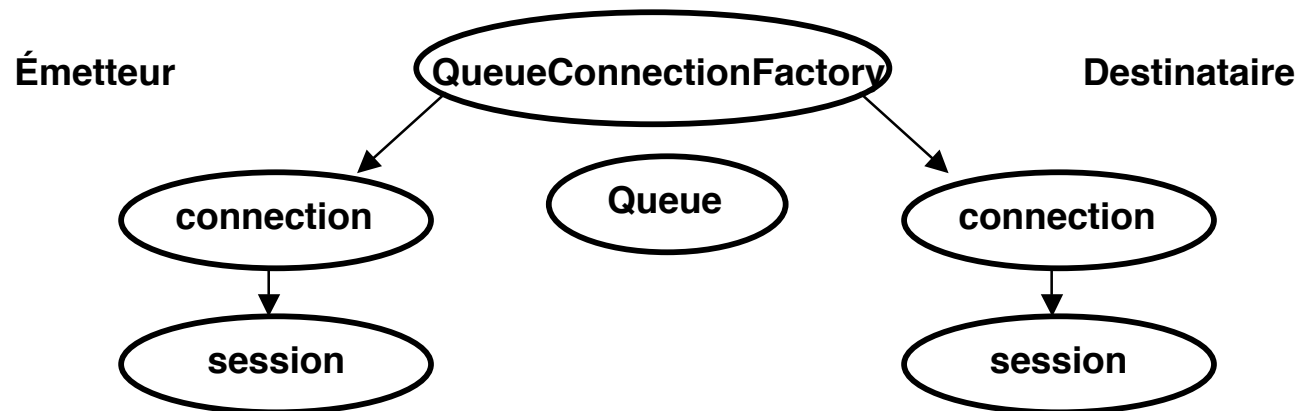
Architecture générale de JMS



JMS “Point-to-Point”

Opérations préalables :

création de *Queue* et *QueueConnectionFactory* par l'administrateur
enregistrement dans *messaging* sous JNDI



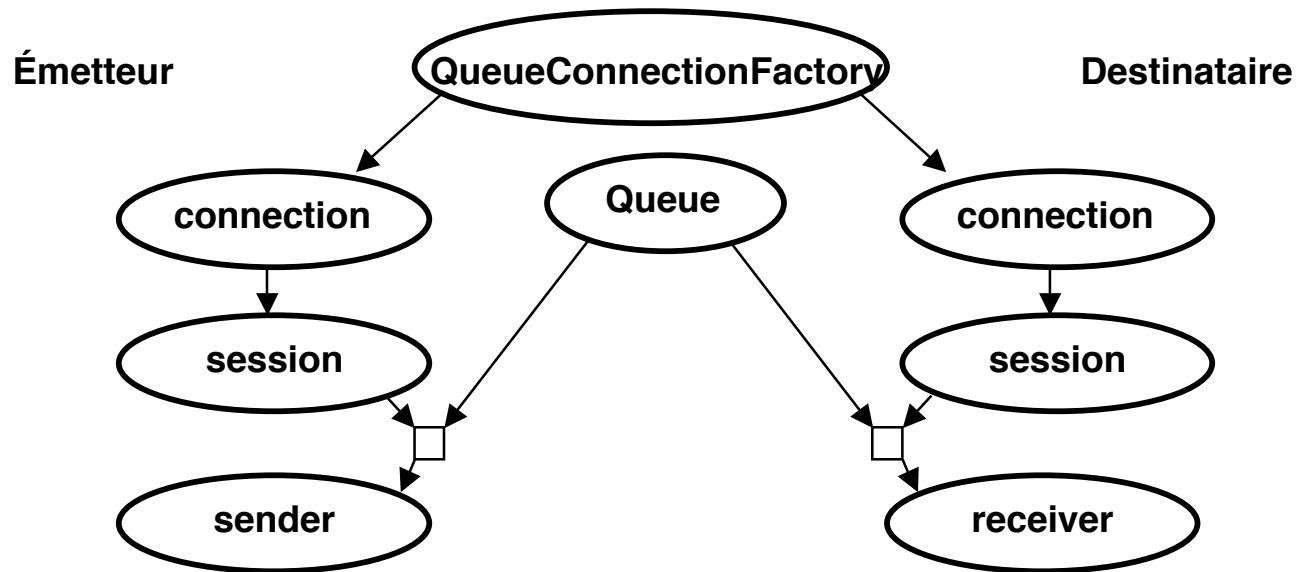
```
QueueConnectionFactory connectionFactory =  
    (QueueConnectionFactory) messaging.lookup ("nom_fabrique");
```

```
Queue queue =  
    (Queue) messaging.lookup ("nom_queue.");
```

```
QueueConnection connection = connectionFactory.createQueueConnection ();
```

```
QueueSession session = connection.createQueueSession ();
```


JMS “Point-to-Point”



Chez l'émetteur

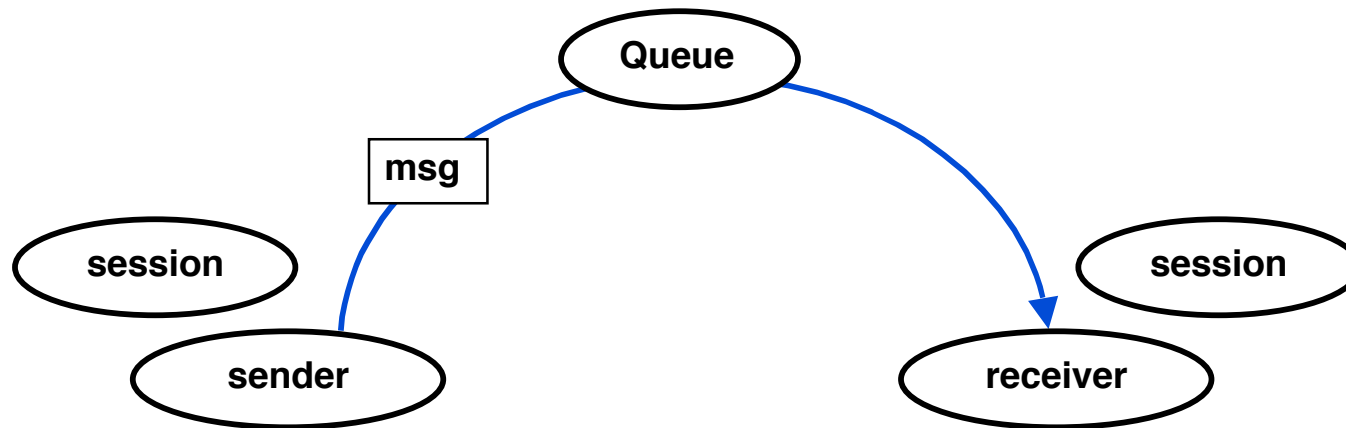
```
QueueSender sender =  
    session.CreateSender (queue) ;
```

Chez le destinataire

```
String selector =  
    new String ("(name = 'Bull') or (name = 'IBM')");  
QueueReceiver receiver =  
    session.CreateReceiver (queue, selector) ;
```

Réception sélective des messages dont l'attribut *name* contient une chaîne donnée

JMS “Point-to-Point”



Chez l'émetteur

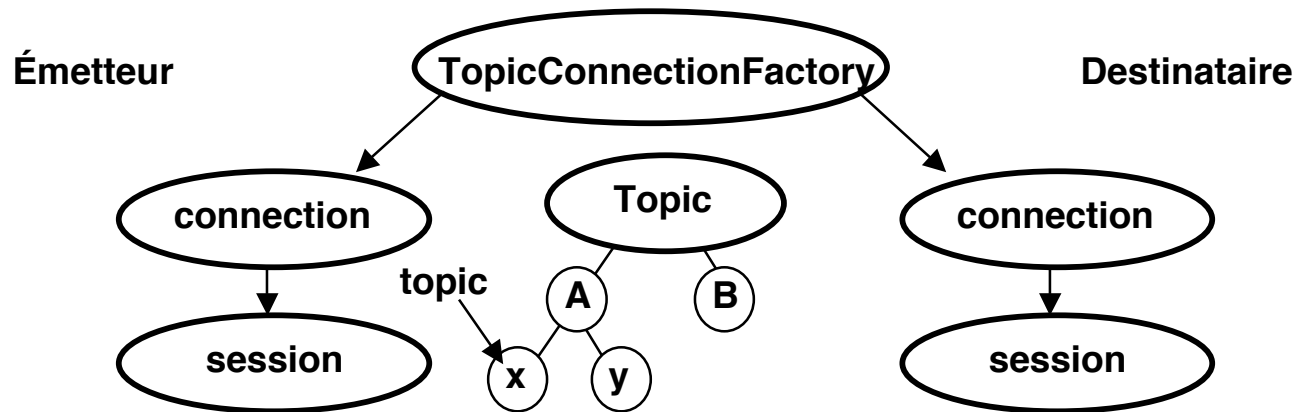
```
TextMessage msg =  
    session.createTextMessage ;  
msg.setText ("le texte du message") ;  
Sender.send (msg) ;
```

Chez le destinataire

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

JMS “Publish-Subscribe”

Opérations préalables :
création de *Topic* et *TopicConnectionFactory* par l'administrateur
enregistrement dans *messaging* sous JNDI



```
TopicConnectionFactory connectionFactory =  
    (TopicConnectionFactory) messaging.lookup ("nom_fabrique") ;
```

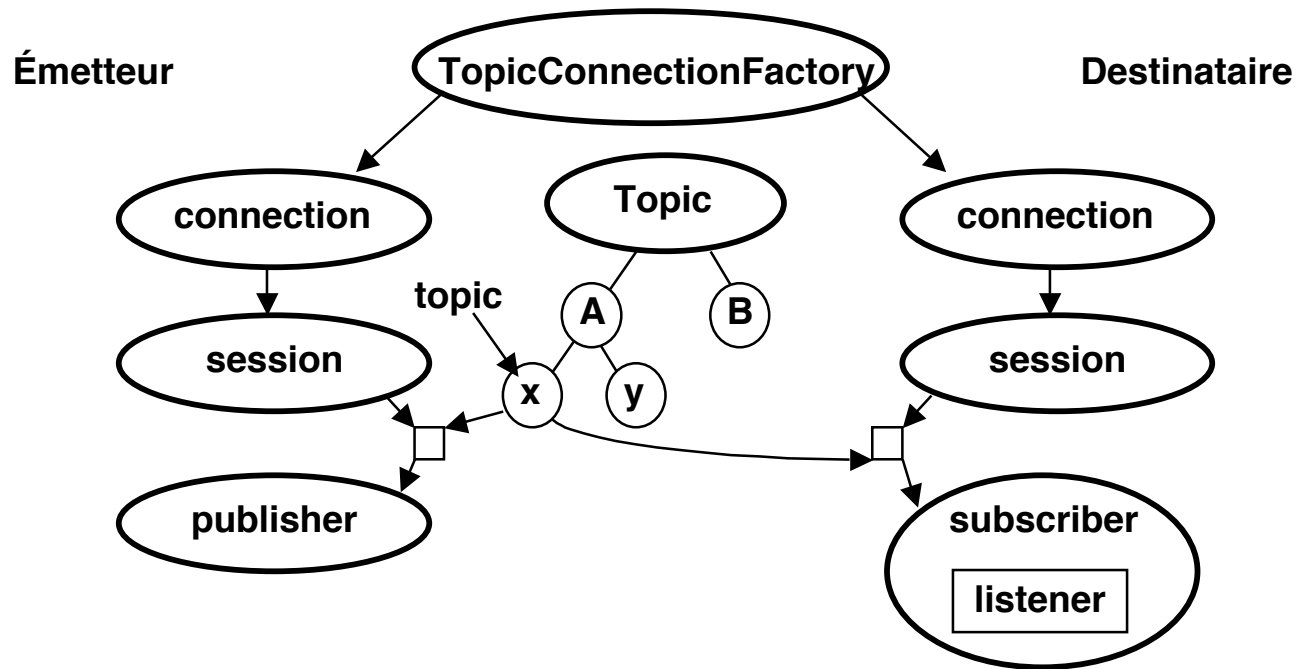
```
Topic topic =  
    (Topic) messaging.lookup ("/A/x") ;
```

```
TopicConnection connection = connectionFactory.createTopicConnection () ;
```

```
TopicSession session = connection.createTopicSession (false, Session.CLIENT_ACKNOWLEDGE)
```

pas en mode transaction acquitté par client

JMS “Publish-Subscribe”



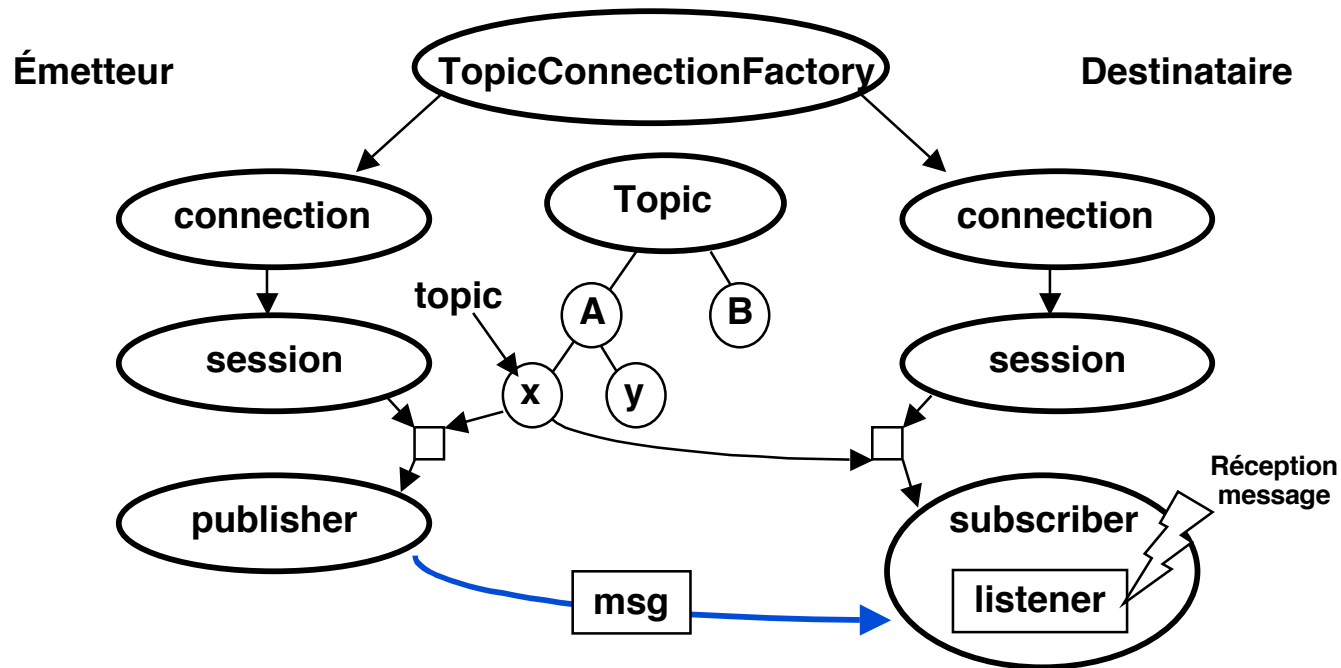
Chez l'émetteur

**TopicPublisher publisher =
session.CreatePublisher (topic) ;**

Chez le destinataire

**TopicSubscriber subscriber =
session.CreateSubscriber (topic) ;
Subscriber.setMessageListener(listener)**

JMS “Publish-Subscribe”



Chez l'émetteur

```
publisher.publish (msg) ;
```

Chez le destinataire : programme du *listener*

```
public class listener implements javax.jms.MessageListener {  
    void onMessage (Message msg) throws JMSEException {  
        // déballer et traiter le message  
        ...  
    }  
}
```

Service d'événements CORBA

- Définit un ensemble de classes permettant la communication par événements
- Deux modes : *push* et *pull*

push (initiative émetteur, ou producteur d'événements)



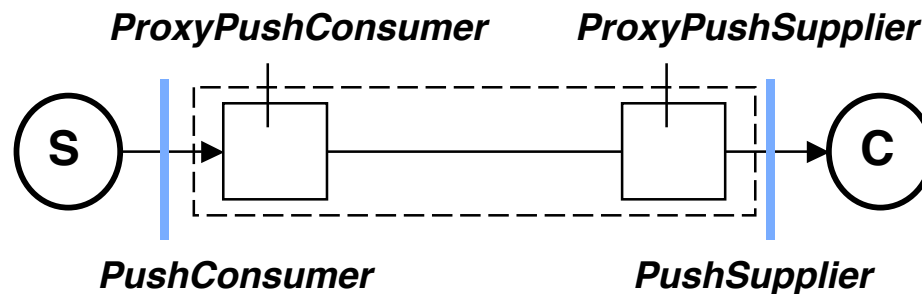
pull (initiative récepteur, ou consommateur d'événements)



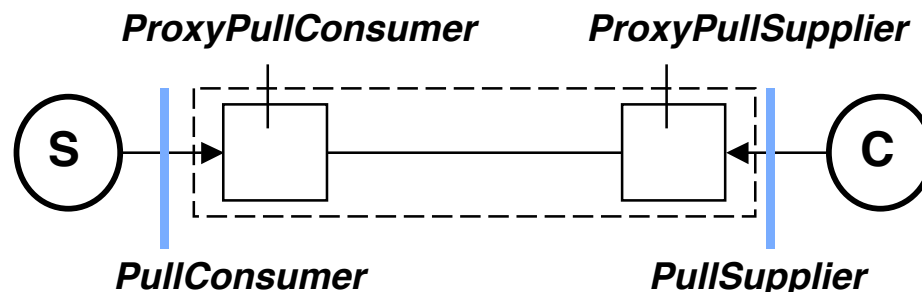
Événements CORBA : principes de base

■ Utilisation du principe du mandataire (*proxy*)

- ◆ Chaque correspondant s'adresse au *proxy* de l'autre (des autres)
- ◆ Le service d'événements fournit un mécanisme pour la création des *proxy*
Mode push



Mode pull

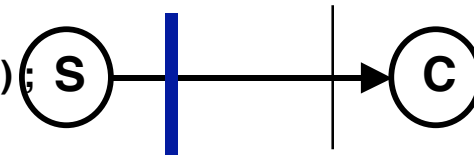


Le module *EventComm*

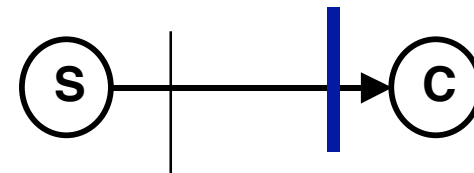
- ◆ Fournit les 4 interfaces de base du service d'événements

```
module CosEventComm {  
  exception Disconnected {} ;
```

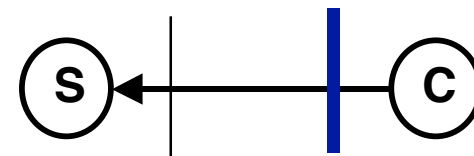
```
  interface PushConsumer {  
    void push (in any data) raises (Disconnected) ;  
    void disconnect_push_consumer () ;  
  } ;
```



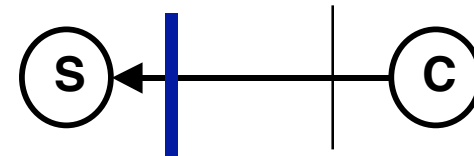
```
  interface PushSupplier {  
    void disconnect_push_supplier () ;  
  } ;
```



```
  interface PullSupplier {  
    any pull () raises (Disconnected) ;  
    any try_pull ()  
      raises (Disconnected) ;  
    void disconnect_pull_supplier () ;  
  } ;
```



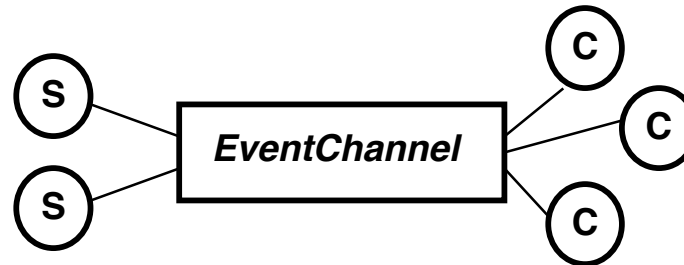
```
  interface PullConsumer {  
    void disconnect_pull_consumer () ;  
  } ;  
};
```



Gestion des participants : *EventChannel*

■ Fonctions

- ◆ Gestion de la communication entre producteurs multiples et consommateurs multiples
- ◆ Ces participants sont connectés à travers un canal d'événements (*EventChannel*)



- ◆ Fonctionne selon le principe de la fabrique (*factory*)
- ◆ Connexion d'un nouveau participant = fabrication et fourniture du *proxy* correspondant à sa fonction (S ou C, *push* ou *pull*)
- ◆ L'interface *EventChannel* fournit les primitives de gestion des canaux d'événements

Le module *EventChannelAdmin*

```
#include "EventComm.idl"
module CosEventChannelAdmin {
  exception AlreadyConnected {};
  exception TypeError {};

  interface ProxyPushConsumer ...
  ...
  interface ProxyPushSupplier ...
  ...
  interface ProxyPullConsumer ...
  ...
  interface ProxyPullSupplier ...
  ...

  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier ();
    ProxyPullSupplier obtain_pull_supplier (); };

  interface SupplierAdmin {
    ProxyPushconsumer obtain_push_consumer ();
    ProxyPullconsumer obtain_pull_consumer (); };

  interface EventChannel {
    ConsumerAdmin for_consumers (); // ajouter nouveau consommateur
    SupplierAdmin for_suppliers (); // ajouter nouveau producteur
    void destroy ; }; // détruire le canal
};
```

Les interfaces des *proxy*
(détails plus loin)

L'interface de gestion
des consommateurs

L'interface de gestion
des producteurs

L'interface de gestion
du canal d'événements

Interfaces des *proxy*

```
module CosEventChannelAdmin {
  exception AlreadyConnected {};
  exception TypeError {};

  interface ProxyPushConsumer : CosEventComm ::PushConsumer {
    void connect push_supplier
      (in CosEventComm::PushSupplier push_supplier)
      raises AlreadyConnected; }

  interface ProxyPushSupplier : CosEventComm ::PushSupplier {
    void connect push_consumer
      (in CosEventComm::PushConsumer push_consumer)
      raises AlreadyConnected, TypeError; }

  interface ProxyPullConsumer : CosEventComm ::PullConsumer {
    void connect pull_supplier
      (in CosEventComm::PullSupplier pull_supplier)
      raises AlreadyConnected, TypeError; }

  interface Proxy PullSupplier : CosEventComm ::PullSupplier {
    void connect pull_consumer
      (in CosEventComm::PullConsumer pull_consumer)
      raises AlreadyConnected; }

  ...
};
```

Étapes du développement d'une application

- ◆ Obtenir un canal d'événements (soit *channel*)
 - ❖ La gestion des canaux n'est pas du ressort du service d'événements
 - ❖ On utilise une fabrique de canaux et un serveur de noms
- ◆ Côté producteur (par ex. mode *push*) [très schématique]

```
supplier_admin = channel->for_suppliers() ;  
proxy_push_consumer = supplier_admin->obtain_push_consumer ;  
proxy_push_consumer->connect_push_supplier (...);
```

Initialiser une structure de “callback” (appel en retour)
pour activer le consommateur lors de l'arrivée du message

```
proxy_push_consumer->push (message) ;
```

- ◆ Côté consommateur

```
consumer_admin = channel->for_consumers() ;  
proxy_push_supplier = consumer_admin->obtain_push_supplier ;  
proxy_push_supplier->connect_push_consumer (...);
```

Lors du “callback” signalant l'arrivée d'un message, extraire et traiter le message reçu

Jini : un service de coordination à objets partagés

■ Motivations

- ◆ Développement d'applications réparties dans lesquelles l'environnement change dynamiquement par ajout ou suppression d'éléments (objets mobiles, appareils divers, etc.) et où l'application doit s'adapter dynamiquement à ces changements

■ Objectif

- ◆ Fournir un service assurant la coordination d'un ensemble de processus à travers un espace d'information partagé
- ◆ En particulier, permettre l'intégration et la découverte **dynamiques** de ressources dans un environnement réparti
- ◆ Implémentation : intégré à Java, mais les principes peuvent être réutilisés dans un environnement non Java
 - ❖ **Repose sur un modèle de coordination par espace d'objets partagés (JavaSpaces)**

JavaSpaces : coordination par objets partagés

■ Espace de tuples

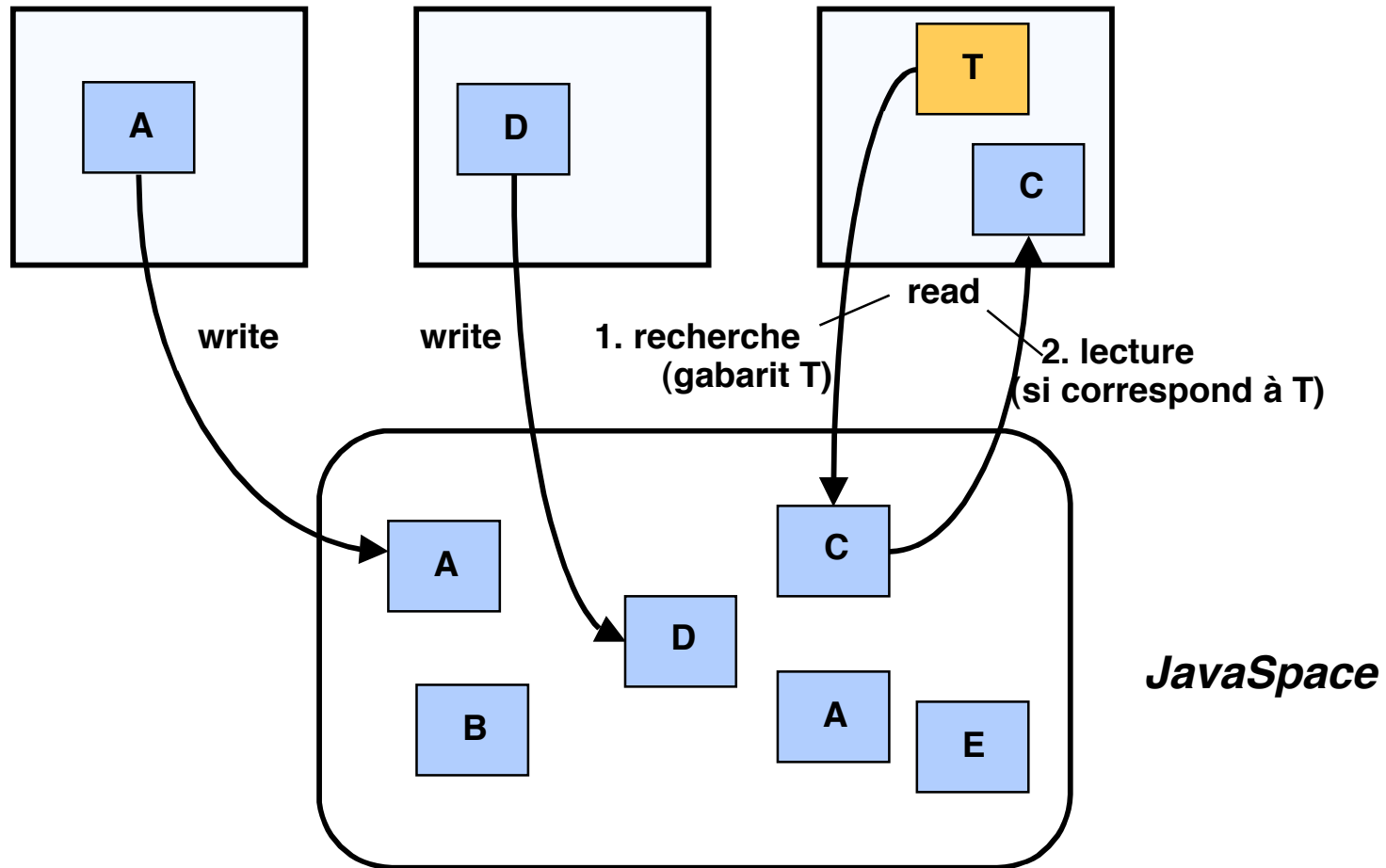
- ◆ Modèle inspiré de Linda
- ◆ Un *JavaSpace* = un espace partagé contenant des **tuples** (ensembles de références typées à des objets Java)
- ◆ Représentation d'un tuple : ensemble de références sérialisées

■ Opérations sur les tuples

- ◆ Écrire (*write*) : insérer un tuple dans un JavaSpace
- ◆ Lire : rechercher dans un JavaSpace un tuple filtré par un gabarit (*template*)
- ◆ Gabarit : modèle pour un tuple, contient des références typées, peut contenir des champs NULL (qui sont conformes à toute référence)
 - ❖ **Opération bloquante pendant la recherche**
 - ❖ **Deux variantes**
 - ▲ *read* : le tuple lu reste dans le JavaSpace
 - ▲ *take* : le tuple lu est extrait du JavaSpace

Opérations sur un *JavaSpace*

processus



Architecture de Jini

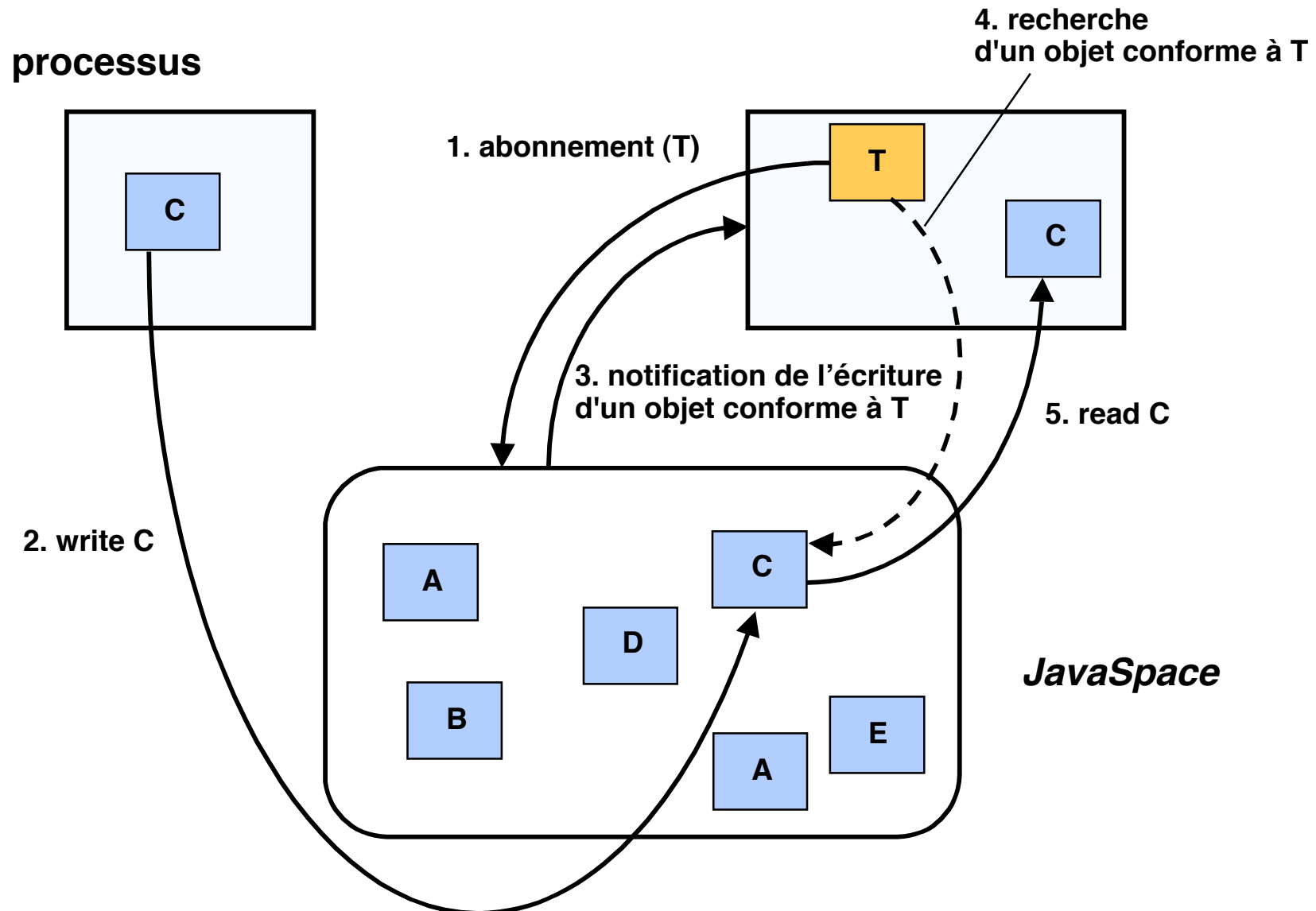
■ 3 niveaux

- ◆ Bas : infrastructure de communication (Java RMI), localisation de services
- ◆ Moyen : service d'événements ; notifications, réalisation de transactions, autres
- ◆ Haut : clients et services (sécurité, transactions, etc.)

■ Service d'événements (analogue à *publish-subscribe*)

- ◆ Abonnement : le processus client qui s'abonne indique un objet veilleur (*listener*) qui doit être appelé (via RMI) si l'événement arrive
- ◆ L'abonnement a toujours une durée limitée (doit être renouvelé au delà)
- ◆ Cas particulier d'événement : écriture d'un tuple, correspondant à un gabarit spécifié, dans un JavaSpace

Événements



Problèmes d'implémentation

■ Implémentation répartie d'un *JavaSpace*

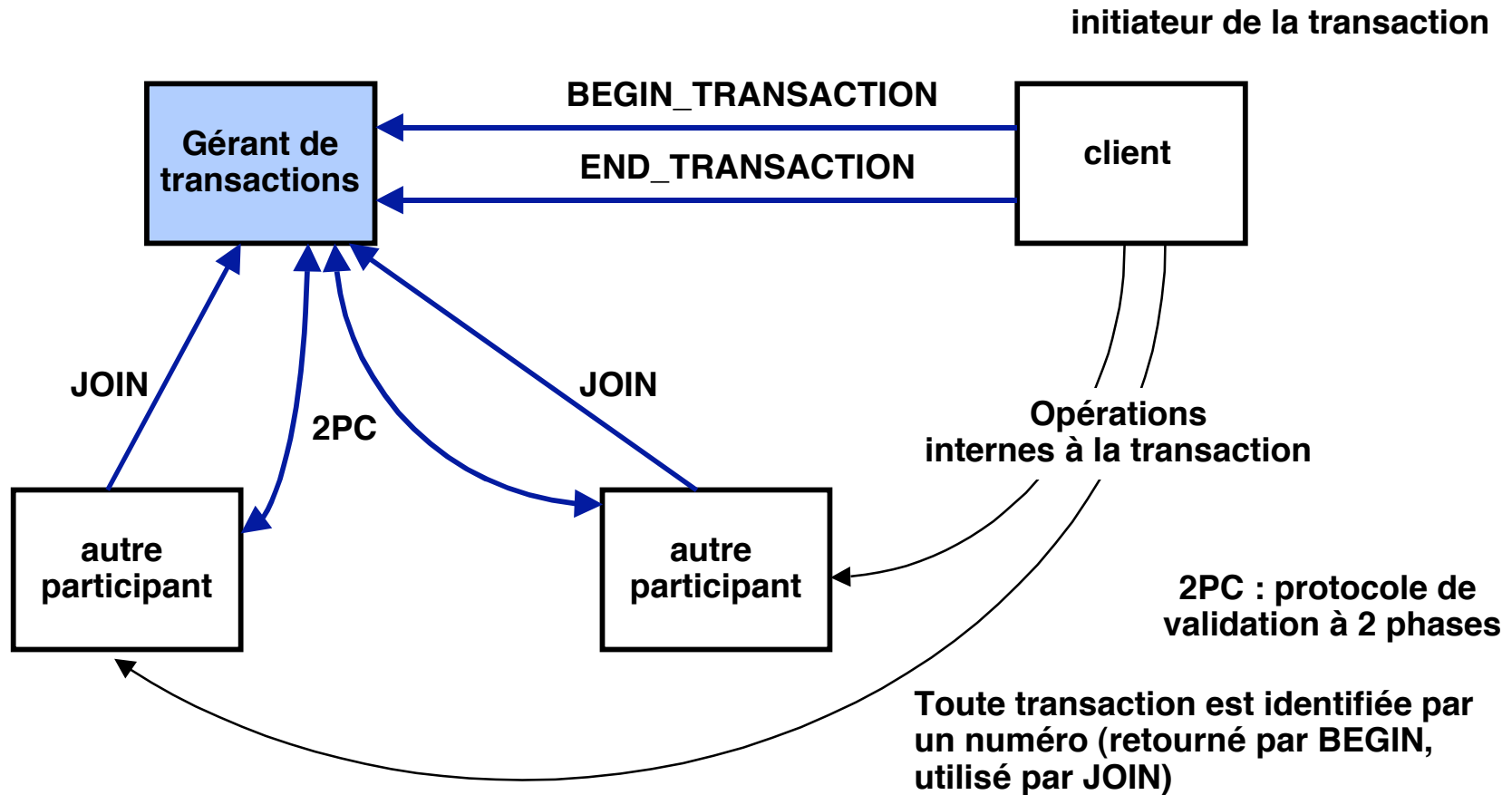
- ◆ Le *JavaSpace* est dupliqué sur toutes les machines
 - ❖ Accès efficace (local)
 - ❖ Mais il faut garder la cohérence, cher si modifications fréquentes
 - ❖ Nécessite service de diffusion (fiable) - cf plus loin dans le cours, tolérance aux fautes
- ◆ Le *JavaSpace* n'est pas dupliqué
 - ❖ Accès distant nécessaire hors du site de résidence
 - ❖ Gestion simple, mais problèmes d'efficacité
- ◆ Solutions mixtes : duplication partielle

Localisation de services

- **Une application Jini peut utiliser divers services**
- **Les services doivent être localisés**
 - ◆ La localisation utilise un principe analogue aux JavaSpaces, mais plus efficace : recherche par gabarit
 - ◆ Chaque service s'enregistre avec des couples (attribut, valeur)
 - ◆ La recherche se fait sur un ou plusieurs attributs (cf pages jaunes)
 - ◆ Plusieurs services de localisation peuvent coexister
 - ◆ Les services de localisation sont eux-mêmes localisés
 - ❖ Au moyen d'une recherche par diffusion
 - ❖ En se signalant eux-mêmes
 - ❖ Noter qu'il n'y a pas d'adresse fixe pour un service primitif de localisation

Synchronisation

■ Outil de base : transactions



Références

Systèmes de communication par queues de messages

MQSeries d'IBM : <http://www.ibm.co/software/integration/wmq>

MessageQ de BEA : <http://www.bea.com/>

FalconMQ : <http://www.level8.com>

Bus logiciels

JMS : <http://www.java.com/jms>

iBus : <http://www.softwired-inc.com>

Tibco : http://www.tibco.com/software/enterprise_backbone

SonicMQ : <http://www.sonicsoftware.com/products/sonicmq>

Joram : <http://www.scalagent.com>

Service d'événements de CORBA (avec exemples d'utilisation)

D. C. Schmidt, S. Vinoski. OMG Event Object Service, SIGS, vol. 9, 2, Feb. 1997

<http://www.cs.wustl.edu/~schmidt/C++-report-col9.ps.gz>

D. C. Schmidt. On Overview of OMG CORBA Event Services (flips)

<http://www.cs.wustl.edu/~schmidt/coss4.ps.gz>

Jini

J. Waldo, The Jini Specifications, 2nd ed. Prentice Hall, 2000

Ressources sur Jini : <http://www.jini.org>