Réseaux middlewares et serveurs d'application

Partie 1 - Middlewares

Jules Chevalier
jules.chevalier@telecom-st-etienne.fr

Université Jean Monnet - Télécom Saint Etienne

novembre 2014

Introduction

Middleware (intergiciel)

Logiciel assurant la communications entre différentes applications

Plusieurs modèles

- Client-Serveur (RPC, RMI, Corba, Servlet...)
- Communication par messages (MOM, file de messages)
- Communication par évènements (publish/subscribe, push/pull, JMS)
- Modèle à composant (Bean, EJB)
- Modèle à agents mobiles (Agglet, Voyager)
- Modèle à mémoire "virtuelle" partagée (espaces de tuples-JavaSpaces, objets dupliqués)

Introduction

Objectifs

- Utiliser une technologie objet
- Invoquer des méthodes sur des objets distants comme s'ils étaient locaux

```
objetDistant.methode();
```

- Utiliser un objet distant sans savoir où il se trouve mais en le localisant dynamiquement

```
objetDistant=serviceDeNoms.recherche("monObjet");
```

- Récupérer le résultat d'un appel distant sous la forme d'un objet distant

```
distantObject=objetDistant.methode();
```

Introduction

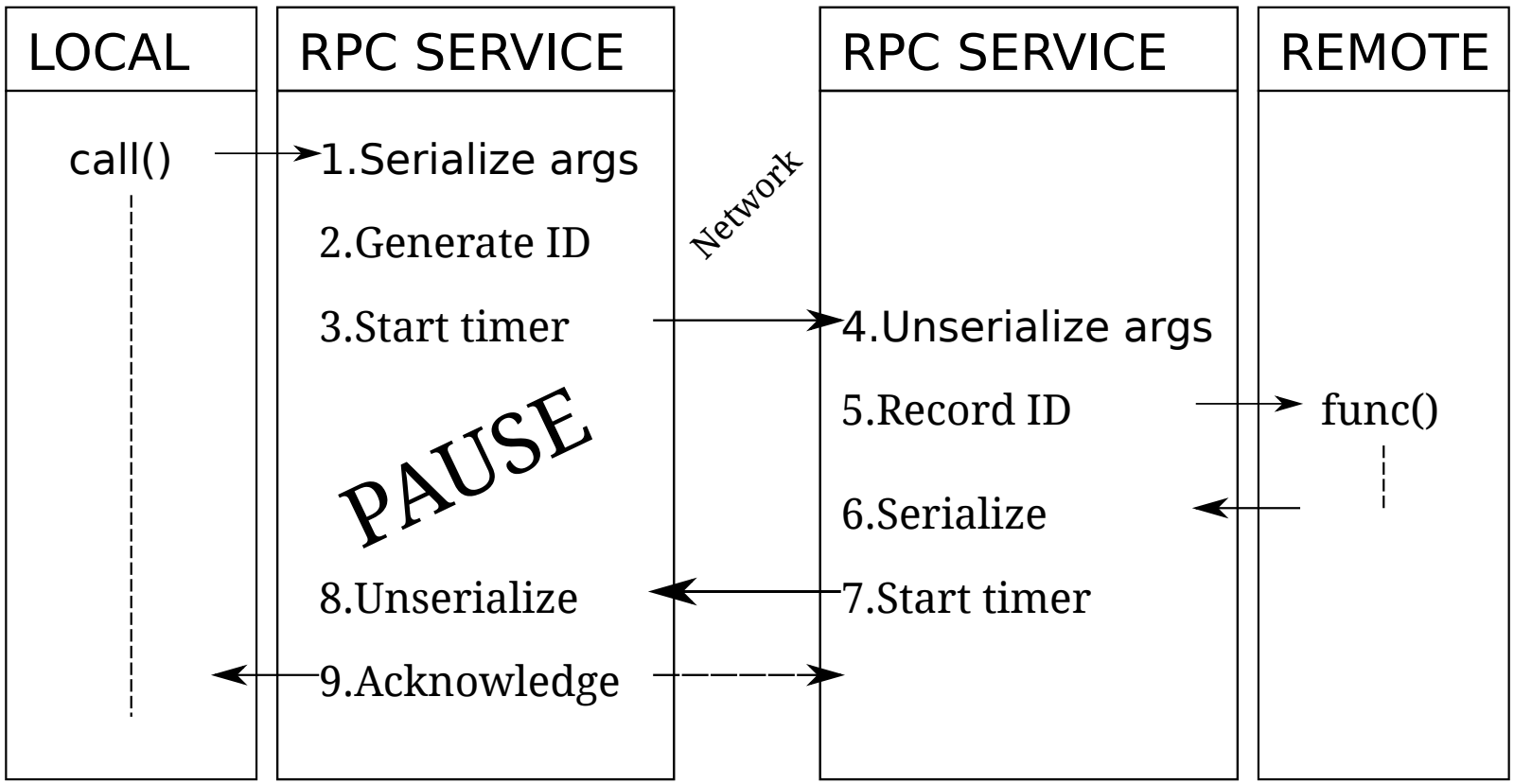
Types de Middlewares

- RPC : Remote Call Procedure
- OOM : Object Oriented Middlware
- MOM : Message Oriented Middleware
- SOA : Service Oriented Architecture

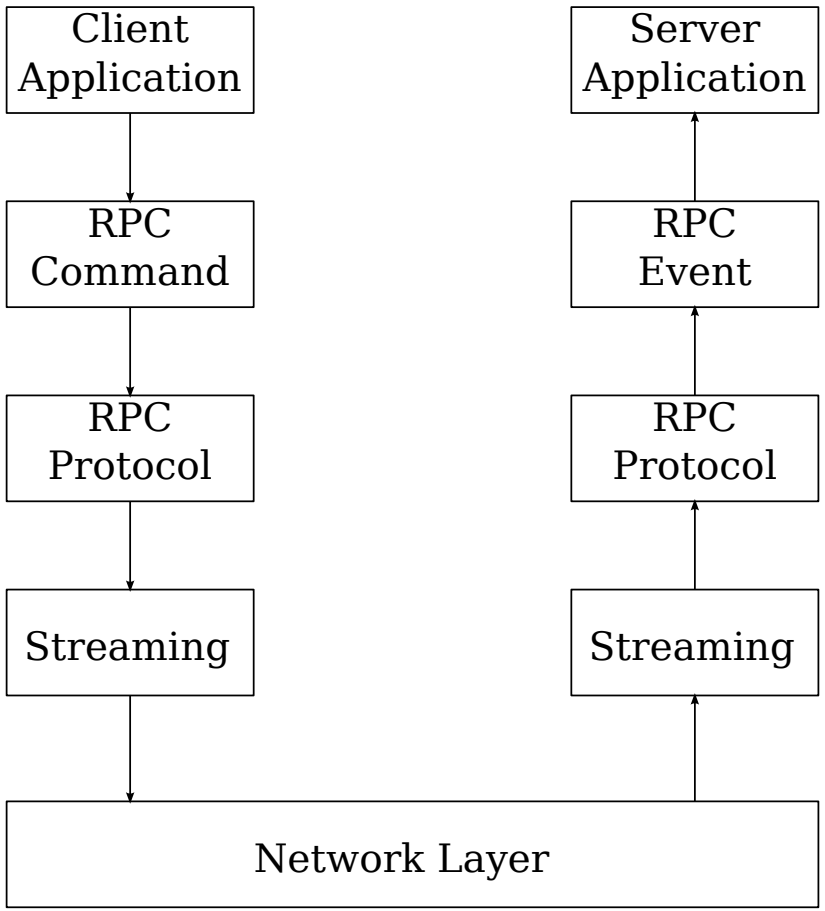
Plan

Plan

Remote Procedure Call



Remote Procedure Call



Remote Procedure Call

Mode synchronisé

Client et Serveur doivent être connectés

Pas de sécurité

Appel de n'importe quelle fonction possible

Pas d'orientation objet

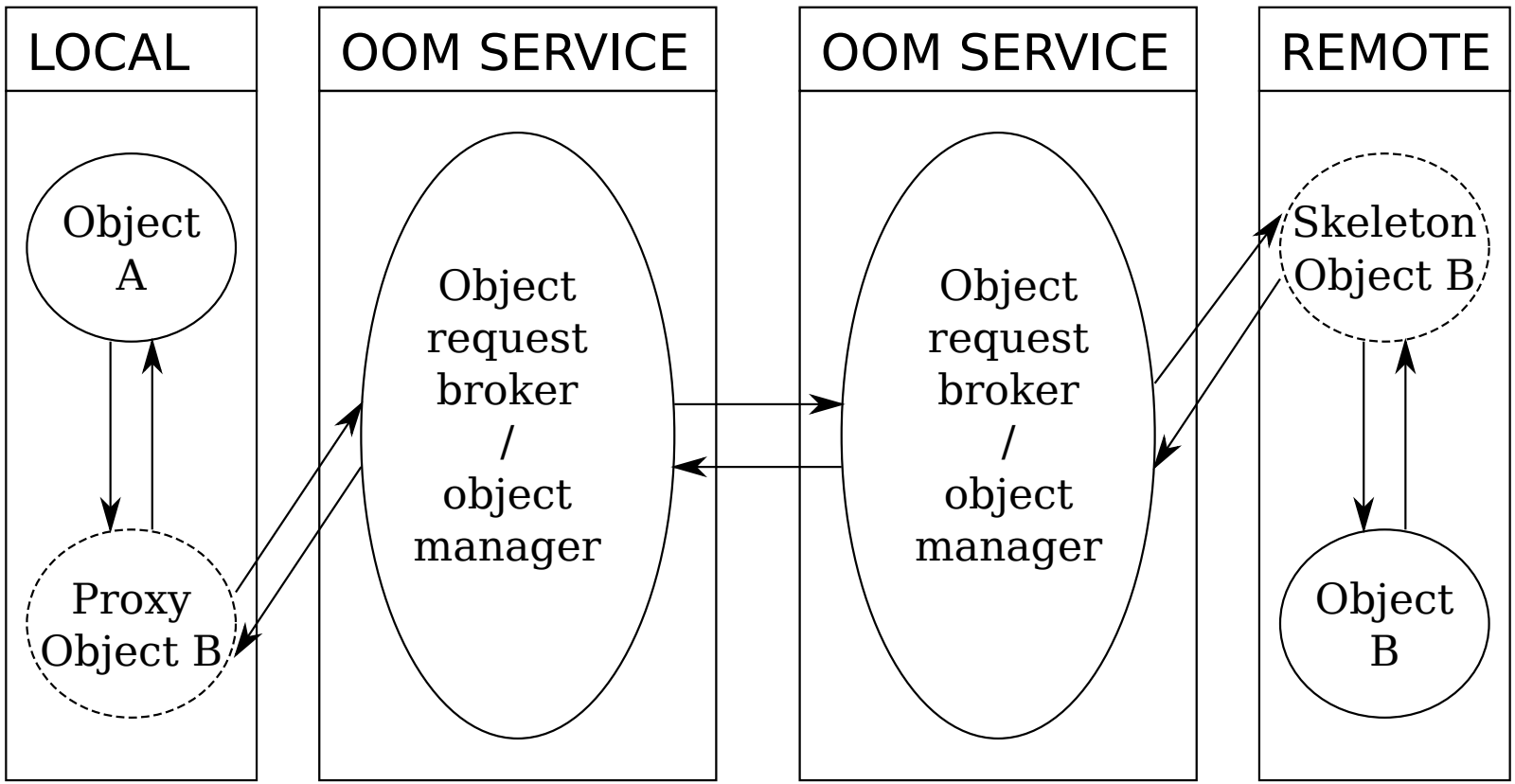
Appel de fonctions au travers d'un serveur

Pas de suivi

Aucune vérification des transmissions, aucune gestion des erreurs

Plan

Object Oriented Middleware



Object Oriented Middleware

Orientation objet

On appelle directement des méthodes sur des objets plutôt que des fonctions

Utilisation d'un annuaire d'objets

Il suffit de connaître le nom d'un objet pour le localiser

Mode synchronisé

Client et serveur doivent être connectés

Plan

Remote Methode Invocation

Implémentation de OOM en Java

- API intégrée au JDK (depuis 1.1)
- Permet d'appeler des méthodes d'objets situés sur d'autres machines
- Utilise des sockets pour communiquer, avec le protocole RMP (Reliable Multicast Protocol)
- Sécurité garantie grâce au RMISecurityManager et au Distributed Garbage Collector

Remote Methode Invocation

Principe

- Une interface permet de connaître les méthodes de l'objet distant
- Utilisation transparente des méthodes et objets distants
  - Passage en paramètres ou en retour des objets dans les fonctions
  - Appels directs de méthodes
  - ...
- Cast des références d'objets distants en n'importe quelle interface implémentée par l'objet

Remote Methode Invocation

Interface = Contrat

- L'interface est un contrat entre le serveur et le client
- Le serveur héberge un objet qui implémente l'interface
- Le client appelle les méthodes de l'interface
- L'interface donne la signature des méthodes
  nom+types des arguments+retour+exceptions
- L'interface RMI est une interface java qui dérive de java.rmi.Remote

Remote Methode Invocation

Passage de paramètres

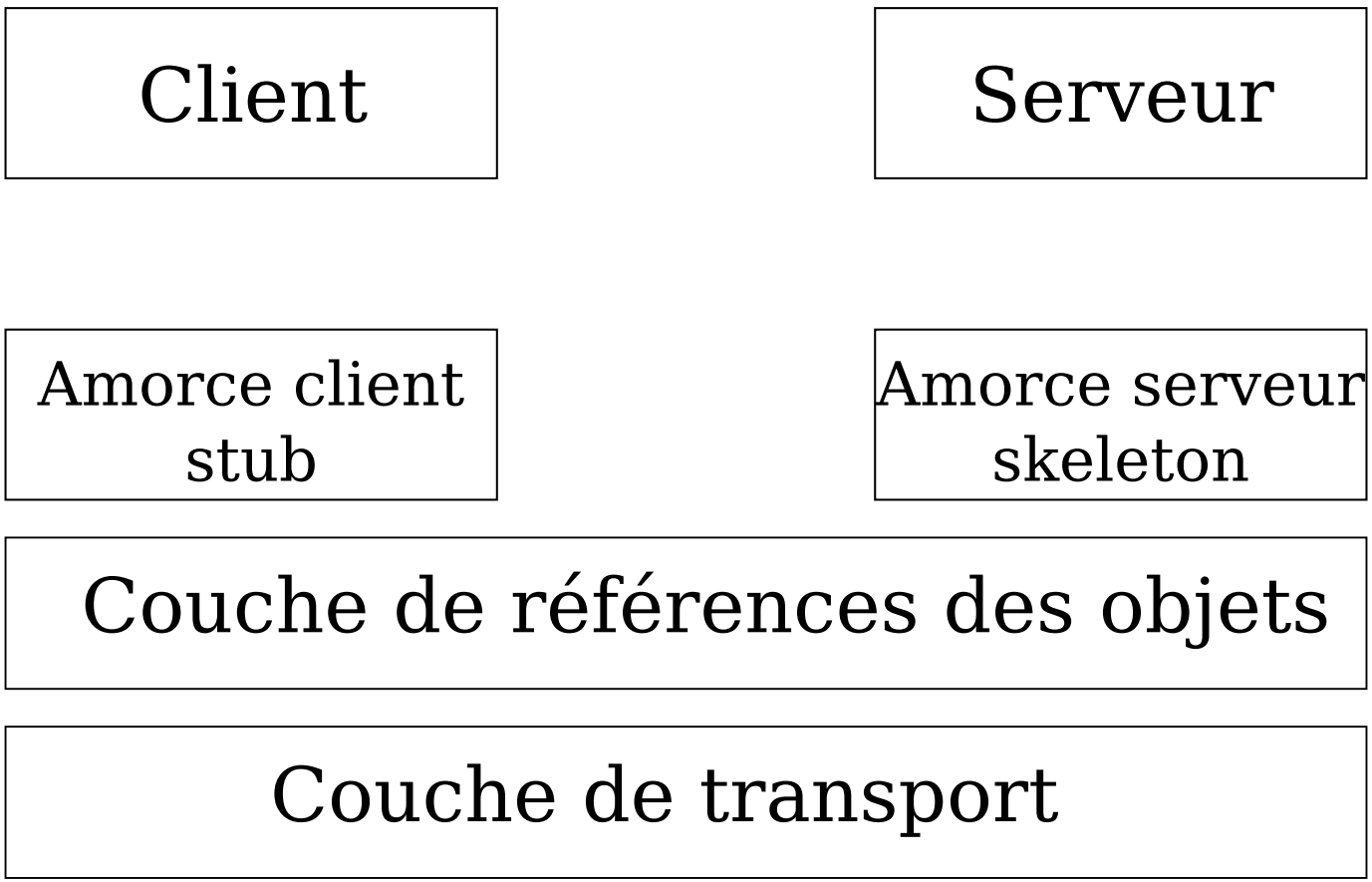
- Passage par copie de tous les arguments, qui sont sérialisés
- Les arguments doivent supporter la sérialisation (Java.io.Serializable)

Remote Methode Invocation

RemoteException

- L'interface RMI doit traiter l'exception RemoteException.
- Elle est levée si l'objet n'existe pas ou si la connexion est interrompue.

Architecture



Architecture

Les Amorces

- Générées automatiquement
- Intermédiaires pour le transport des appels distants
  - Appel sur la couche réseau
  - (Dé)sérialisation des paramètres

Architecture

Marshalling

- C'est l'opération de sérialisation/désérialisation, c'est-à-dire la transformation des objets en flux de données transmissibles sur le réseau
- Pour cette opération, les objets doivent implémenter Java.io.Serializable
- Java.io.Externalizable permet de définir sa propre méthode de sérialisation

Architecture

Amorce client stub

- Donne une vue des méthodes de l'objet distant
- Transmet l'invocation distante à la couche des références distantes
- Sérialise des arguments (Marshalling)
- Désérialise des valeurs de retour (Demarshalling)

Architecture

Amorce serveur skeleton

- Désérialise les arguments
- Appelle la méthode de l'objet sur le serveur
- Sérialise la valeur de retour

Architecture

Couche de références distantes

- Transforme la référence locale de l'objet en référence distante
- Utilise le service rmiregister
  - Service d'annuaire pour tous les objets distants enregistrés
  - Un seul service par JVM

Architecture

Couche transport

- Connecte deux espaces d'adressages, deux JVM
- Suit les connexions en cours
- Écoute et répond aux invocations
- Transporte les invocations

Développement RMI

Processus de développement

- Définition de l'interface RMI pour l'objet distant
- Création et compilation de la classe implémentant cette interface
- Création et compilation de l'application serveur RMI
- Démarrage de rmiregister et lancement de l'application serveur RMI
- Création, compilation et lancement de l'application cliente utilisant le stub et donc les références aux objets distants

Architecture

Remarque : Compilateur RMIC

- Depuis Java SE 6, les armoces sont compilées automatiquement.

Exemple : ECHO distribué

Implémentation

- Echo : interface de description de l'objet distant
- EchoImpl : Implémentation de l'interface Echo
- EchoAppliServer : application serveur RMI
- EchoClient : Application cliente utilisant l'objet distant

Exemple : ECHO distribué

RemoteEcho.java

```
1   import java.rmi.Remote;
2   import java.rmi.RemoteException;
3   public interface RemoteEcho extends Remote {
4       String Hello() throws RemoteException;
5       String Echo(String s) throws RemoteException;
6   }
```

Exemple : ECHO distribué

RemoteEchoImpl.java

```
1   import java.rmi.RemoteException;
2   import java.rmi.server.UnicastRemoteObject;
3   public class RemoteEchoImpl extends UnicastRemoteObject implements RemoteEcho{
4       private static final long serialVersionUID = -7278596802637355415L;
5       protected RemoteEchoImpl() throws RemoteException {
6           super();
7       }
8       public String Hello() throws RemoteException {
9           return "Hello world";
10      }
11      public String Echo(String s) throws RemoteException {
12          return "Echo : "+s+" (from MyComputer)";
13      }
14  }
```

Exemple : ECHO distribué

Server.java

```
1   import java.rmi.AccessException;
2   import java.rmi.RemoteException;
3   import java.rmi.registry.LocateRegistry;
4   import java.rmi.registry.Registry;
5   public class Server {
6       public static void main(String[] args){
7           try {
8               Registry registry = LocateRegistry.createRegistry(18500);
9               RemoteEchoImpl echo;
10              echo = new RemoteEchoImpl();
11              String rebindName = "echo";
12              registry.rebind(rebindName, echo);
13              System.out.println("Object Echo ok");
14          } catch (AccessException e) {
15              e.printStackTrace();
16          } catch (RemoteException e) {
17              e.printStackTrace();
18          }
19      }
20  }
```

Exemple : ECHO distribué

Commentaires

- On utilise RMISecurityManager, un SecurityManager spécifique
- On enregistre l'objet dans le rmiregistry :

  ```
  Naming.rebind("monObjet",objet);
  ```

- L'accès se fait via l'URL

  ```
  rmi://nomServeurRMI:port/objet
  ```

- La récupération se fait avec

  ```
  Naming.lookup(url);
  ```

Exemple : ECHO distribué

Avant Java 6

- Le compilateur d'amorces rmic crée les classes stub et skeleton :
- rmic EchoImpl crée EchoImpl_stub.class et EchoImpl_skel.class

À partir Java 6

- Les amorces sont compilées et intégrées automatiquement. Leur utilisation est devenue totalement transparente.

Exemple : ECHO distribué

Client.java

```
1   import java.rmi.NotBoundException;
2   import java.rmi.RemoteException;
3   import java.rmi.registry.LocateRegistry;
4   import java.rmi.registry.Registry;
5   public class Client {
6       public static void main(String[] args) {
7           String serverAdress = "161.3.199.192";
8           try {
9               Registry registry = LocateRegistry.getRegistry(serverAdress,18500);
10              RemoteEcho echo = (RemoteEcho)registry.lookup("echo");
11              System.out.println("Connection OK to "+serverAdress);
12              String s = echo.Hello()+"\n"+echo.Echo("All your base are belong to us");
13              System.out.println(s);
14          } catch (RemoteException e) {
15              e.printStackTrace();
16          } catch (NotBoundException e) {
17              e.printStackTrace();
18          }
19      }
20  }
```

Exemple : ECHO distribué

Commentaires

- On manipule l'objet distant comme un objet local
- On récupère la référence de l'objet grâce au service d'annuaire
- Le client utilise l'objet distant au travers de ses amorces
- A l'exécution, le client télécharge dynamiquement l'amorce cliente depuis le serveur (byte code)

Exemple : ECHO distribué

Sécurité

- Le chargement dynamique de données (objets ou références) pose un problème : On charge du bytecode inconnu
- On utilise donc un securityManager spécifique : RMISecurityManager
  - Vérifie la définition des classes et autorise seulement le passage des arguments et des valeurs des méthodes distantes

Object Oriented Middleware

Avantages

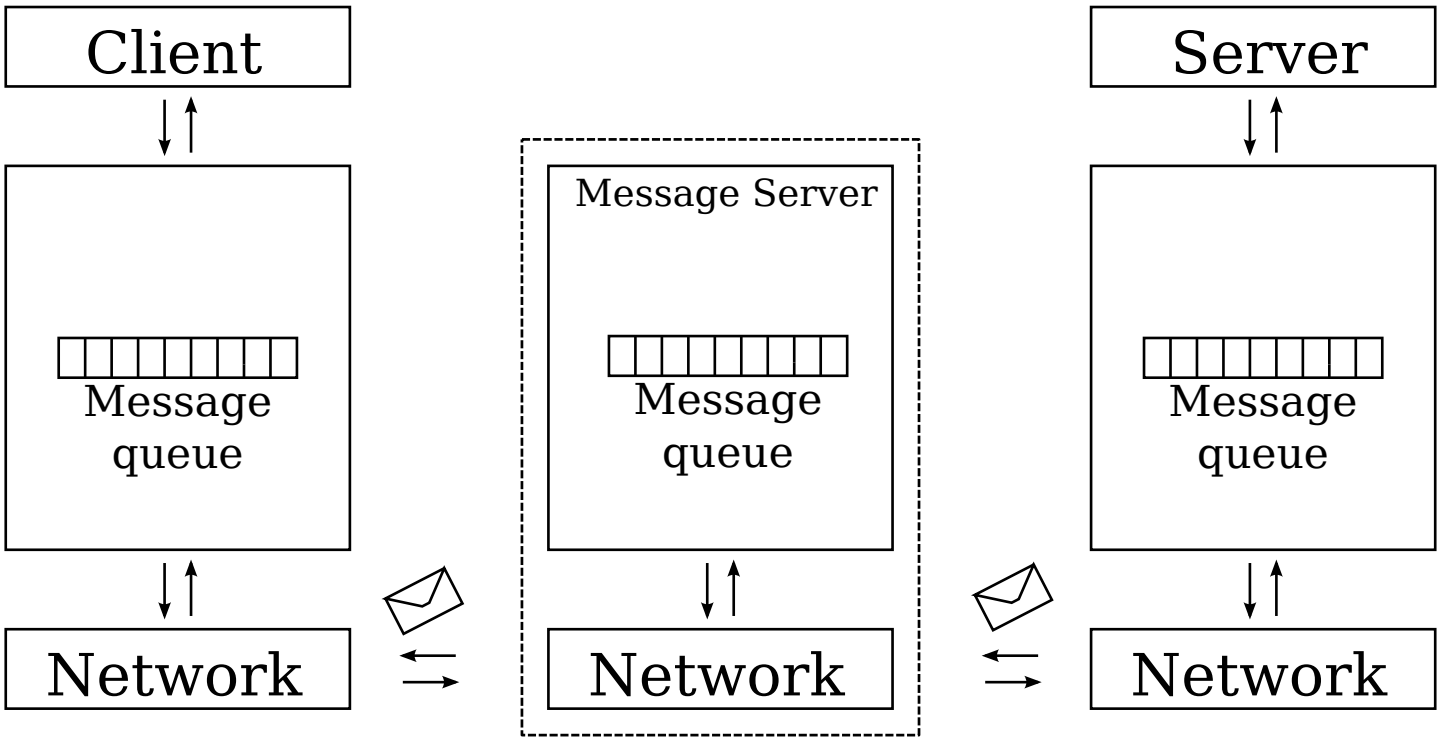
- Orientation objet
- Utilisation transparente (une fois initialisée)

Inconvénients

- Synchronisation requête/réponse
- Localisation des objets contenus dans la référence

Plan

Message Oriented Middleware



Message Oriented Middleware

Fonctionnement

- Communication par messages inter-applicatifs
- Utilisation de files de messages
- Garantie de livraison des messages, ordonnés et persistants
- Services intermédiaires proposés par le serveur de messages
  - Filtrage, transformation, logging, rejeu des messages, etc

Faible couplage client/serveur

- Interaction asynchrone entre le client et le serveur
- Découplage serveur/client possible grâce à un serveur de messagerie

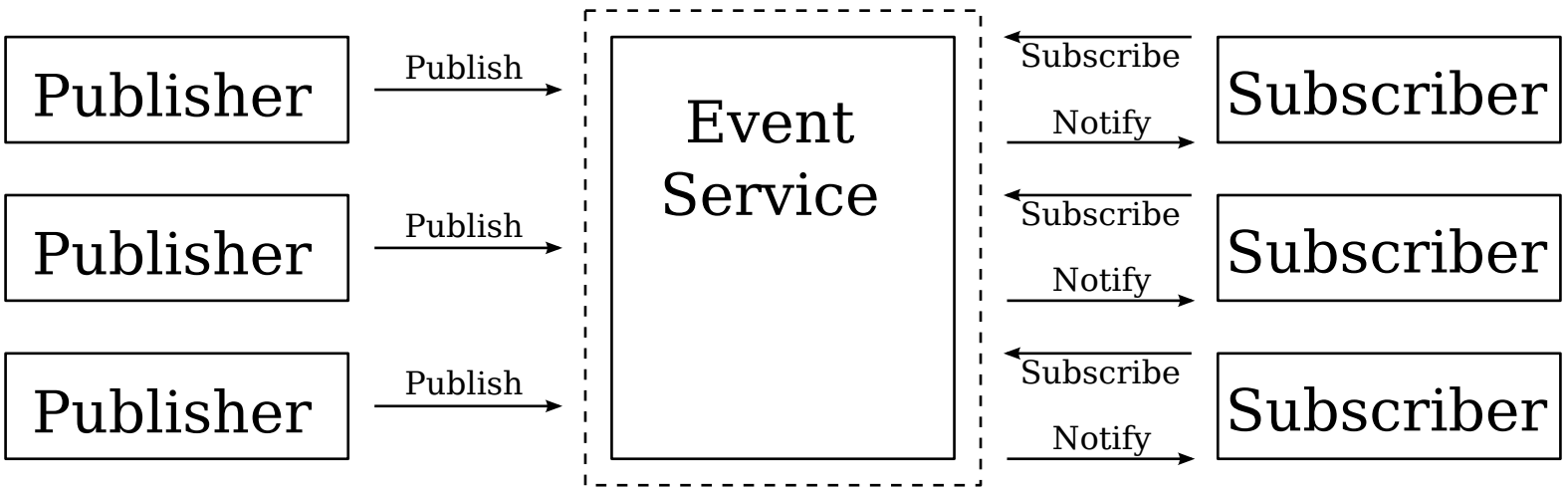
Plan

Event Based Middleware

Fonctionnement

- Extension généralisée de MOM/JMS
- Les publishers publient des events (messages)
- Les subscribers s'abonent à des topics (rubriques)
- Les eventservices informent les subscribers de la disponibilité d'un nouvel event
- Les event peuvent contenir quasiment n'importe quoi

Event Based Middleware



Event Based Middleware

Topic Based

- Les publishers publient des events dans des topics
- Les subscribers s'abonnent à des topics

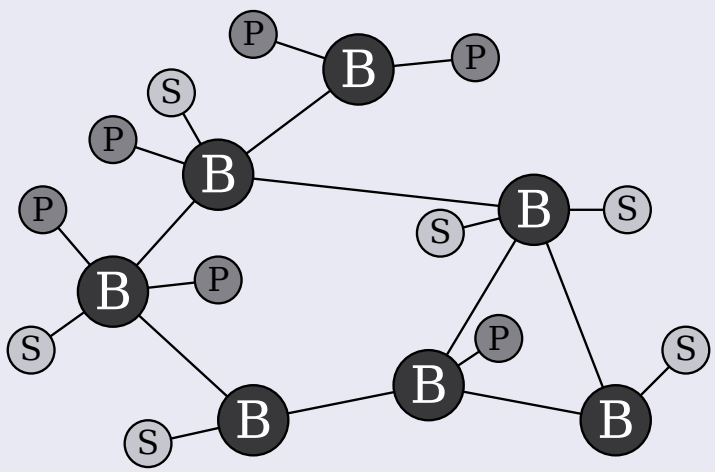
Content Based

- Les publishers publient des events
- Les subscribers utilisent un filtre pour recevoir les events qui les intéressent

Exemple

Hermes : Implémentation Open Source

- Implémentation distribuée d'un service d'évènements Publishers Ⓟ/Subscribers Ⓢ
- Utilise un réseau de Messages Brokers Ⓑ
- Routage des events en fonction du contenu



Composite Event Detection

Modèle plus "évolué" de publish/subscribe

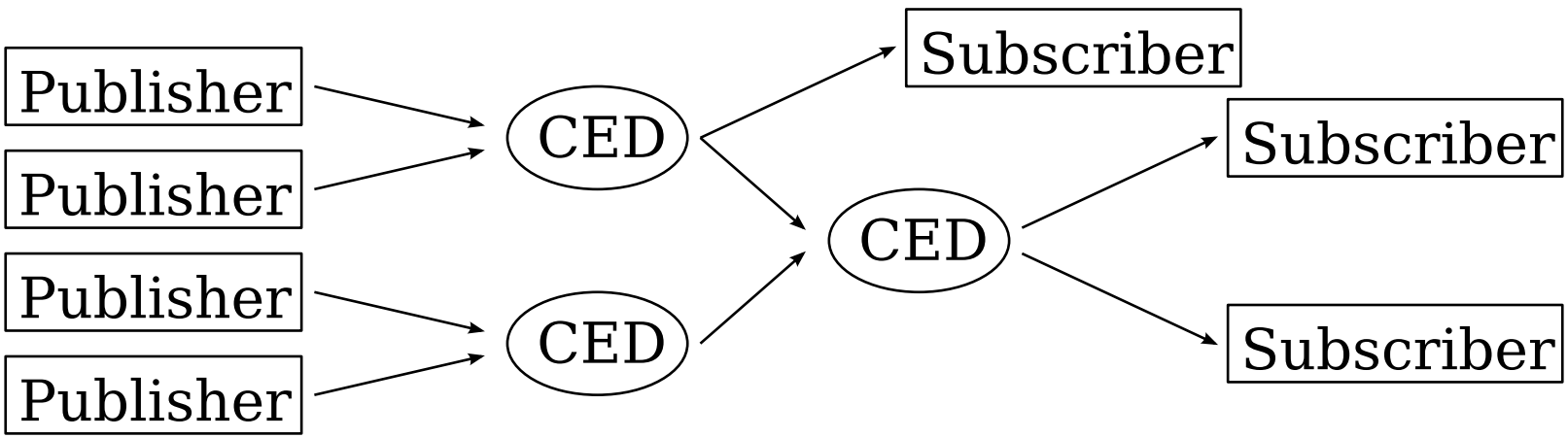
Sans les CED

- Potentiellement des milliers de types de messages
- Pas de possibilité d'inscription à un modèle de message
- Par exemple "Problème d'impression" au lieu de "Plus de papier dans le bac du bas de l'imprimante couleur"

Avec les CED

- Les subscribers s'abonnent à des events primitifs
- Les publishers publient des events composites

Composite Event Detection



Plan

Java Message Service

API intégrée à J2EE

Envoi et réception des messages

- Synchrones : Applications clientes, EJBs, Composants web, ...
- Asynchrones : Applications clientes, beans orientés messages

Mode de communication

- Point à Point : Un producteur envoi un message à un consommateur
- Publish/Subscribe : Des publishers publient, des subscribers souscrivent

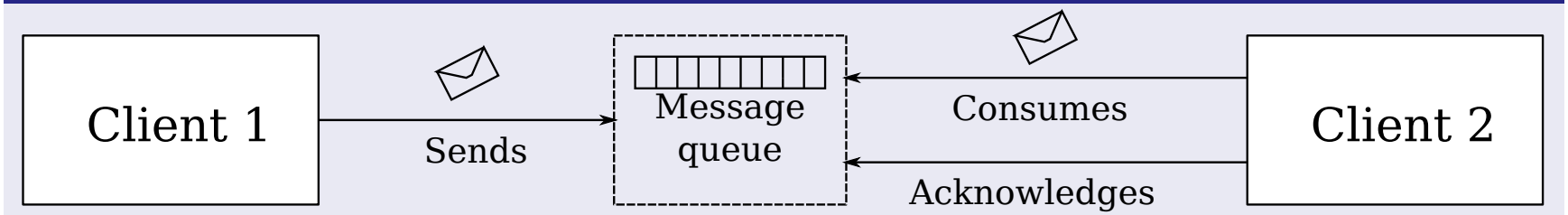
Java Message Service

Architecture

- JMS Provider : Système de gestion de messages qui implémente les interfaces JMS
- Clients JMS : Produisent et consomment les messages
- Messages : Objets qui communiquent des infos entre les clients

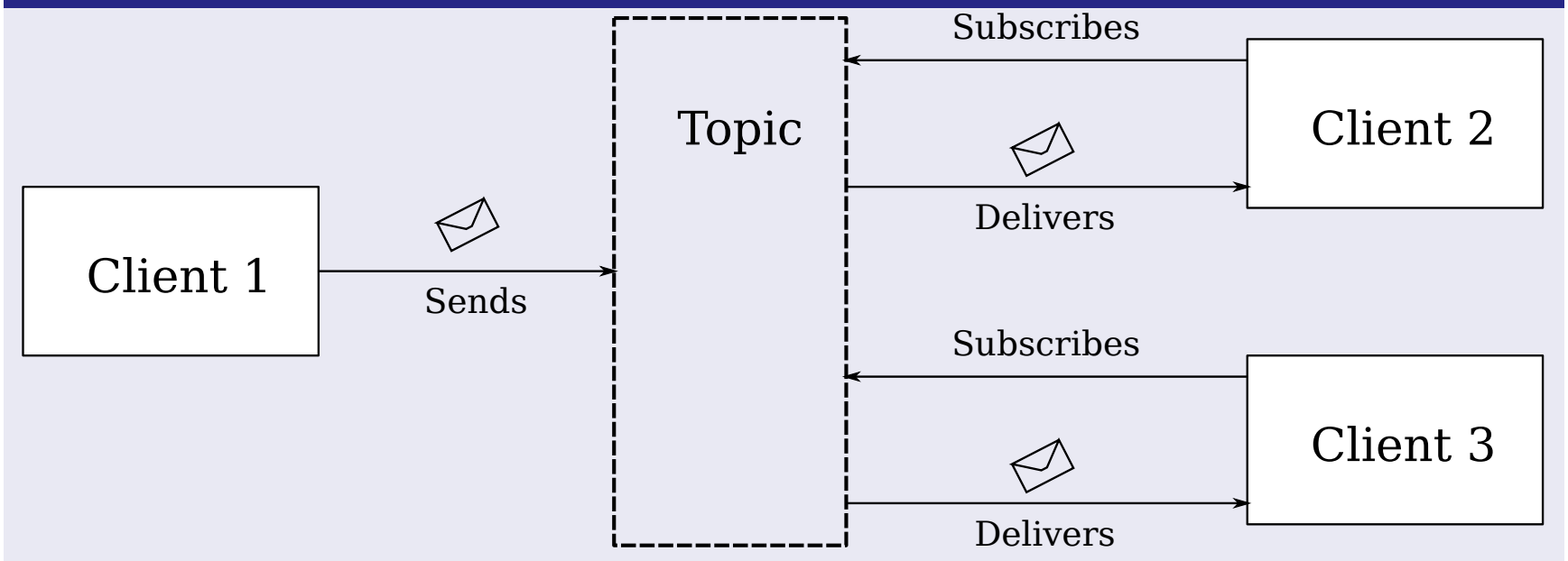
Java Message Service

JMS Point à point



Java Message Service

JMS Publish/Subscribe



Java Message Service

Consommation des messages

- Synchrone
  - Méthode receive() bloquante jusqu'à ce qu'un message soit délivré (ou que le timer arrive à échéance)
- Asynchrone
  - Le client s'enregistre auprès d'un écouteur de messages (listener)
  - Si un message est posté, la méthode onMessage du listener est invoquée

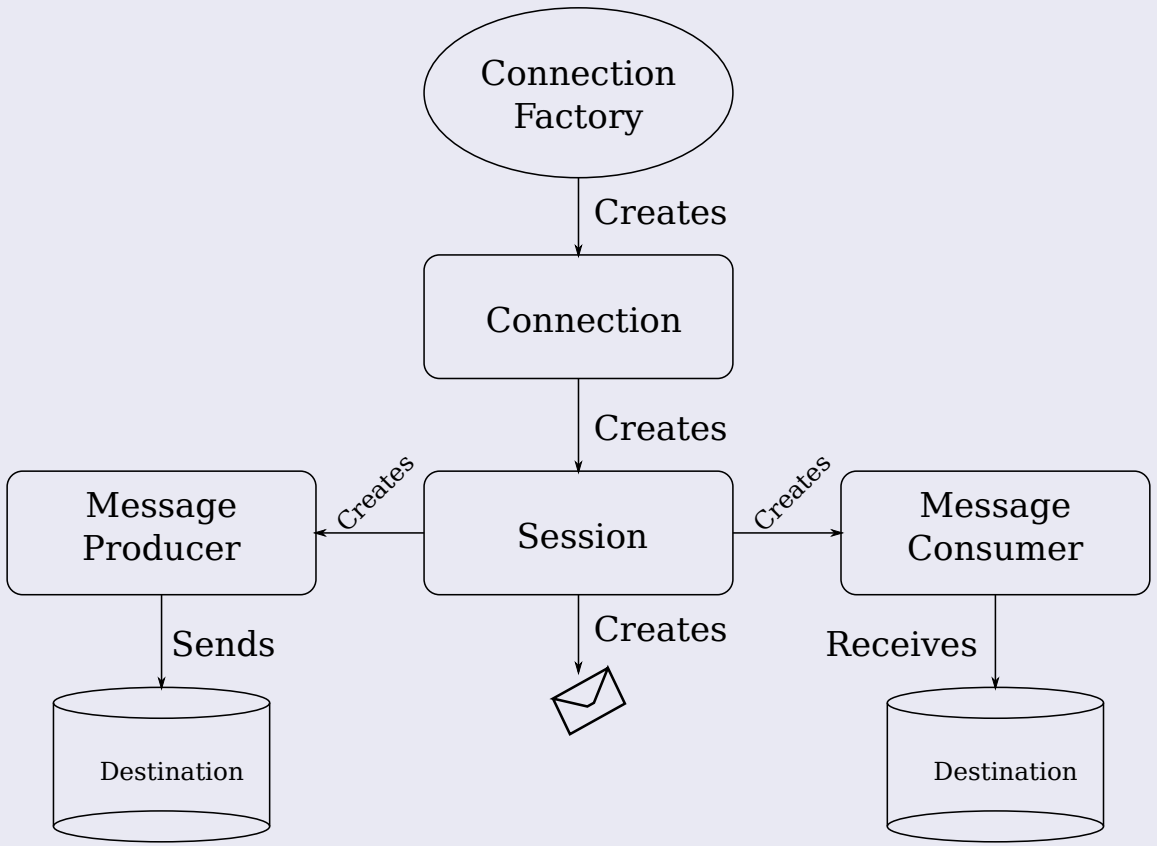
Java Message Service

Modèle de programmation

- Création d'une connexion au fournisseur JMS
- Création d'une session avec cette connexion (permet de gérer le commit/rollback)
- Création du client, en fonction de client et d'approche :
  - Point à Point - Consommateur
  - Point à Point - Producteur
  - Publish/Subscribe - Subscriber
  - Publish/Subscribe - Publisher
- Création du message du type de son choix
- Envoi/Réception du message

Java Message Service

Modèle de programmation



Java Message Service

Connexion factories

```
1  Context ctx = new InitialContext();
2  QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory)ctx.
       lookup("QueueConnectionFactory");
3  TopicConnectionFactory topicConnectionFactory = (TopicConnectionFactory)ctx.
       lookup("TopicConnectionFactory");
```

Java Message Service

Initialisation du client

```
1   //Le client doit rechercher la destination JMS via JNDI
2   //Represente le theme de la ''discussion''
3   Topic myTopic = (Topic) ctx.lookup("MyTopic");
4   Queue myQueue = (Queue) ctx.lookup("MyQueue");
5   //Creation de la connexion avec le JMS provider
6   QueueConnection queueConnection =queueConnectionFactory.createQueueConnection
        ();
7   TopicConnection topicConnection =topicConnectionFactory.createTopicConnection();
8   //Fermeture sessions, producteurs et consommateurs
9   queueConnection.close();
10  topicConnection.close();
```

Java Message Service

Production des messages

```
1  QueueSender queueSender = queueSession.createSender(myQueue);
2  TopicPublisher topicPublisher =topicSession.createPublisher(myTopic);
3  //L'envoi de message est realise par :
4  queueSender.send(message);
5  topicPublisher.publish(message);
```

Java Message Service

Consommation synchrone des messages

```
1  QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);
2  TopicSubscriber topicSubscriber = topicSession.createSubscriber(myTopic);
3  //La reception de message debute :
4  queueConnection.start();
5  Message m = queueReceiver.receive();
6  topicConnection.start();
7  Message m = topicSubscriber.receive(1000); // time out d'une seconde
8  //La methode close() permet de rende inactif un consommateur
```

Java Message Service

Consommation asynchrone des messages

```
1    TopicListener topicListener = new TopicListener();
2    topicSubscriber.setMessageListener(topicListener);
3    //La reception de message debute :
4    topicConnection.start();
```

Java Message Service

Consommation synchrone des messages

```
1  QueueListener queueListener = new QueueListener();
2  queueSubscriber.setMessageListener(queueListener);
3  //La reception de message debute :
4  queueConnection.start();
```

Java Message Service

Messages JMS

- Entête (générée automatiquement)
- Des propriétés optionnelles (filtrage de messages sur les queues)
- Corps du message parmi les cinq suivants :
  - **TextMessage** : String
  - **MapMessage** : Couple nom/valeur
  - **BytesMessage** : Suite d'octets
  - **StreamMessage** : suite de valeurs primitives remplies et lues de façon séquentielle
  - **ObjectMessage** : objet java (sérializable)
  - **Message** : void

Java Message Service

Exemple : Envoi et réception d'un **TextMessage**

```
1   TextMessage message = queueSession.createTextMessage();
2   message.setText(msg_text); // msg_text est une chaine (String)
3   queueSender.send(message);
4   //Consommation :
5   Message m = queueReceiver.receive();
6   if (m instanceof TextMessage) {
7   TextMessage message = (TextMessage) m;
8   System.out.println("Reading message: " + message.getText());
9   } else {
10  // Gestion d'erreur
11  }
```

Plan

Message Passing Interface

Présentation

- Norme créée entre 1993 et 1994
- Défini une bibliothèque de fonctions C, C++ et Fortran
- Permet d'utiliser des machines distantes ou multicoeur par passage de messages
- Devenu un standard de communication pour programmes parallèles
- En 1997, MPI-2 apporte de puissantes fonctionnalités

Message Passing Interface

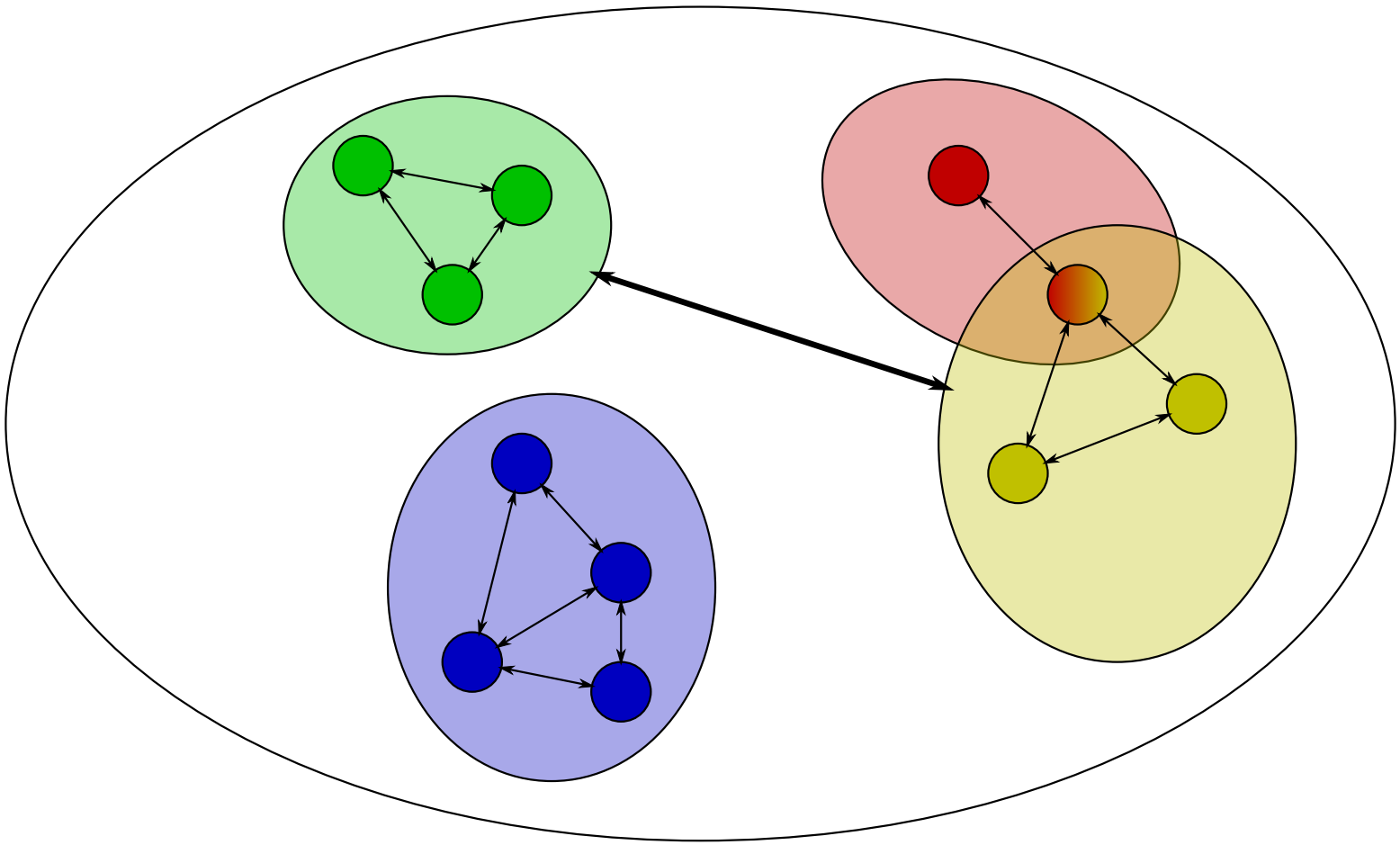
Portable

- Implanté sur la plupart des architectures mémoires
- Disponible sur de nombreux OS/Matériels
- Adapté :
  - Aux machines massivement parallèles à mémoires partagées
  - Aux clusters hétérogènes de machines

Performant

- Bas niveau
- Chaque implantation est optimisée pour le matériel
- Aussi performant sur machine parralèle que sur cluster

Message Passing Interface



Message Passing Interface

Communicateur

- Ensemble, domaine regroupant des processus en leur permettant de communiquer
- Les processus d'un même communicateur peuvent communiquer entre eux
- Ils ont un identifiant dans chaque communicateur
- Ils ont connaissance des autres processus du communicateur

Message Passing Interface

MPI_COMM_WORLD

- Communicateur contenant tout les processus

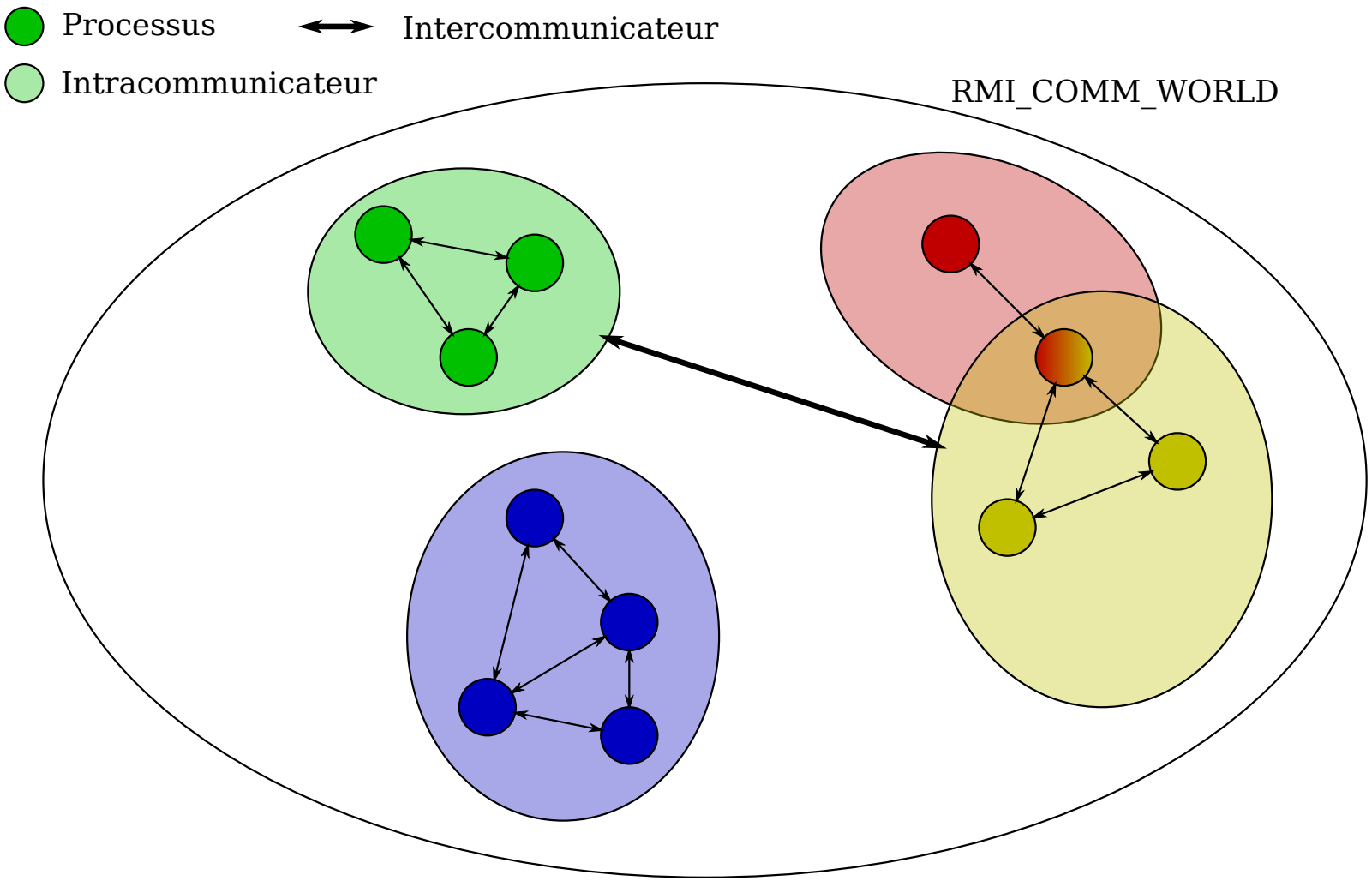
Intracommunicateurs

- Communicateurs *classiques*

Intercommunicateurs

- Communicateurs permettant la communication entre intracommunicateurs

Message Passing Interface



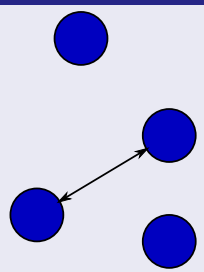
Message Passing Interface

Exemple C

```c
#include<mpi.h>
#define TAG 99

int main(int argc, char **argv){
char msg[20];
int my_rank;
MPI_Status status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

if (my_rank == 0) { /*-- process 0 --*/
strcpy(msg, "Hello world!\n");
MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, TAG, MPI_COMM_WORLD);
}
else {
MPI_Recv(msg, 20, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &status);
printf("Je recois : %s\n", msg);
}

MPI_Finalize();
return 0;
}
```

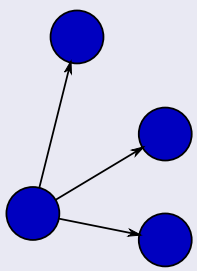
Message Passing Interface

Communication point à point



Permet à deux processus d'un même communicateur d'échanger des données, avec MPI_Send, MPI_Recv et MPI_Sendrecv

Communication collective



Permet à un processus d'envoyer des données à tous les autres processus des données, de découper un tableau entre tous les processus, ou d'effectuer une opération partagée entre tous les processus.

Message Passing Interface

Implémentations

- Classiquement, les implémentations de MPI sont en C, C++ et Fortran. Il existe aussi des implémentations Python, OCaml et Java
- MPICH 2 est une implémentation libre de MPI-2

Message Passing Interface

Avantages

- Bas niveau
- Implémentions dans plusieurs langages
- Fonctionne avec les machines distantes ou multicoeurs
- Clusterisation de processus
- Communication point à point ou broadcast

Inconvénients

- Mode synchrone uniquement
- Pas d'orientation objet dans les implémentations classiques
- Pas de serveur de messages

Plan

Web Services

Principe

- Entre MOM et OOM
- Communication par messages XML véhiculés par le protocole SOAP (Simple Object Access Protocol)
- Description des services par WSDL (Web Services Description Language)
- Découverte des services par UDDI (Universal Description Discovery and Integration)
  - Contient les différents WSDL (=langages) des différents web services

Web Services

Intérêts

- Indépendant du langage
- Standard ouvert du W3C

Plan

Service Oriented Architecture

Objectifs

- Gagner un niveau en orientant l'architecture vers les services
- Décomposer les fonctionnalités en services indépendants
- Décrire le schéma d'interaction entre ces services

Plan

Entreprise Service Bus

Objectifs

- Permettre la communications entre des applications qui ne sont pas forcément prévues pour fonctionner ensemble
- Permettre de communiquer par messages, services ou évènements

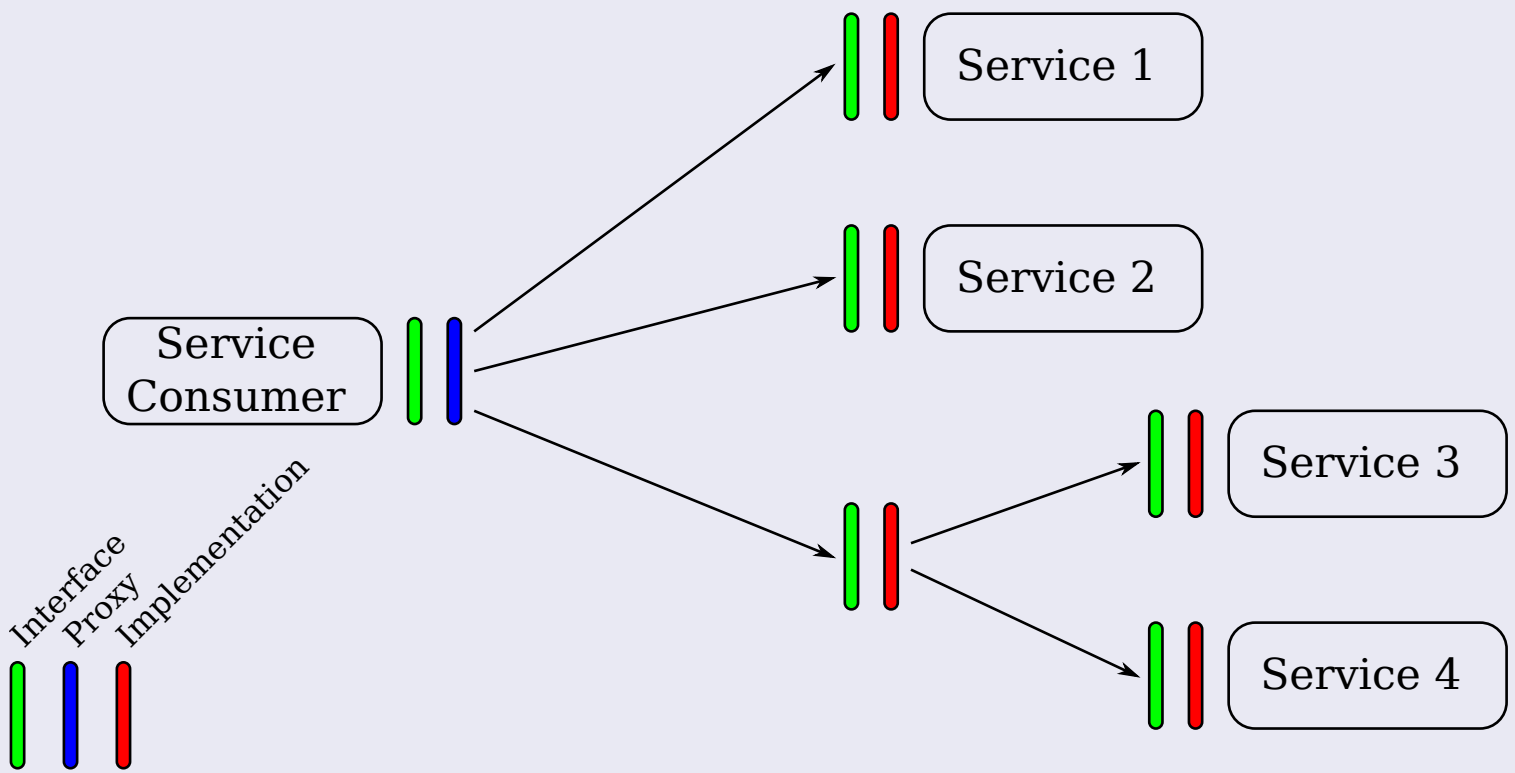
Entreprise Service Bus

Caractéristiques

- Utilise des standards comme XML, JMS ou encore les Web Services
- Les services interagissent en échangeant des données sur un BUS
- Arbitre les requêtes et les réponses
- Utilise les protocoles du Web : SOAP/HTML, SOAP/JMS
- Conversion à la volée pour les différents protocoles

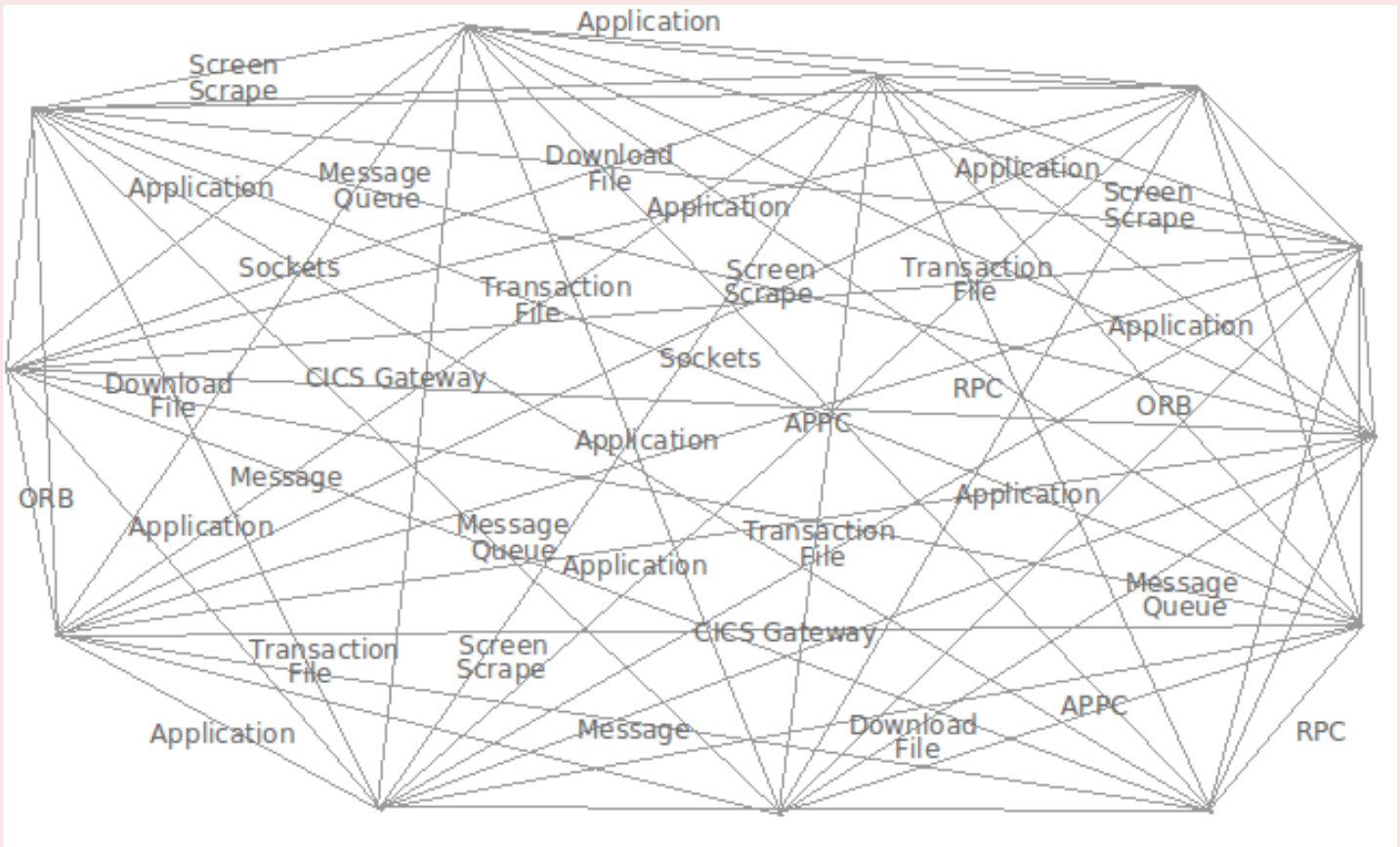
Entreprise Service Bus

Anatomie d'un service



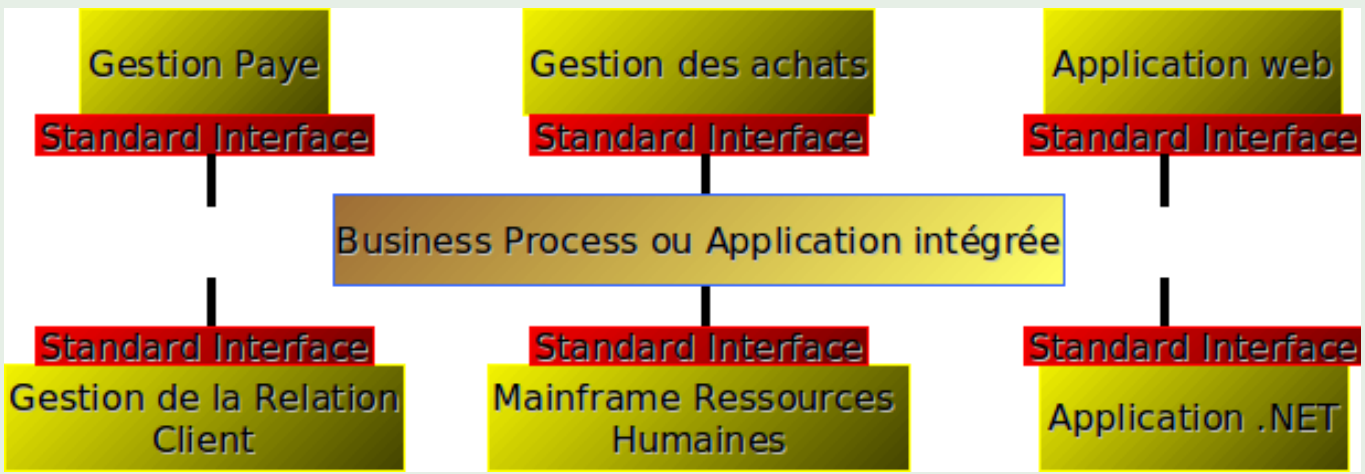
Entreprise Service Bus

Sans ESB



Entreprise Service Bus

Avec ESB



Plan

Java Business Integration

Norme éditée dans le cadre du Java Community Process

Objectifs

- Définir des services autorisant l'intégration de services applicatifs communiquant par web services et échanges de messages XML
- Faire travailler ensemble des services qui n'étaient pas conçus au départ pour le faire

Concrètement

- JBI est un conteneur de conteneur (EJB containers, Web Services, ...)

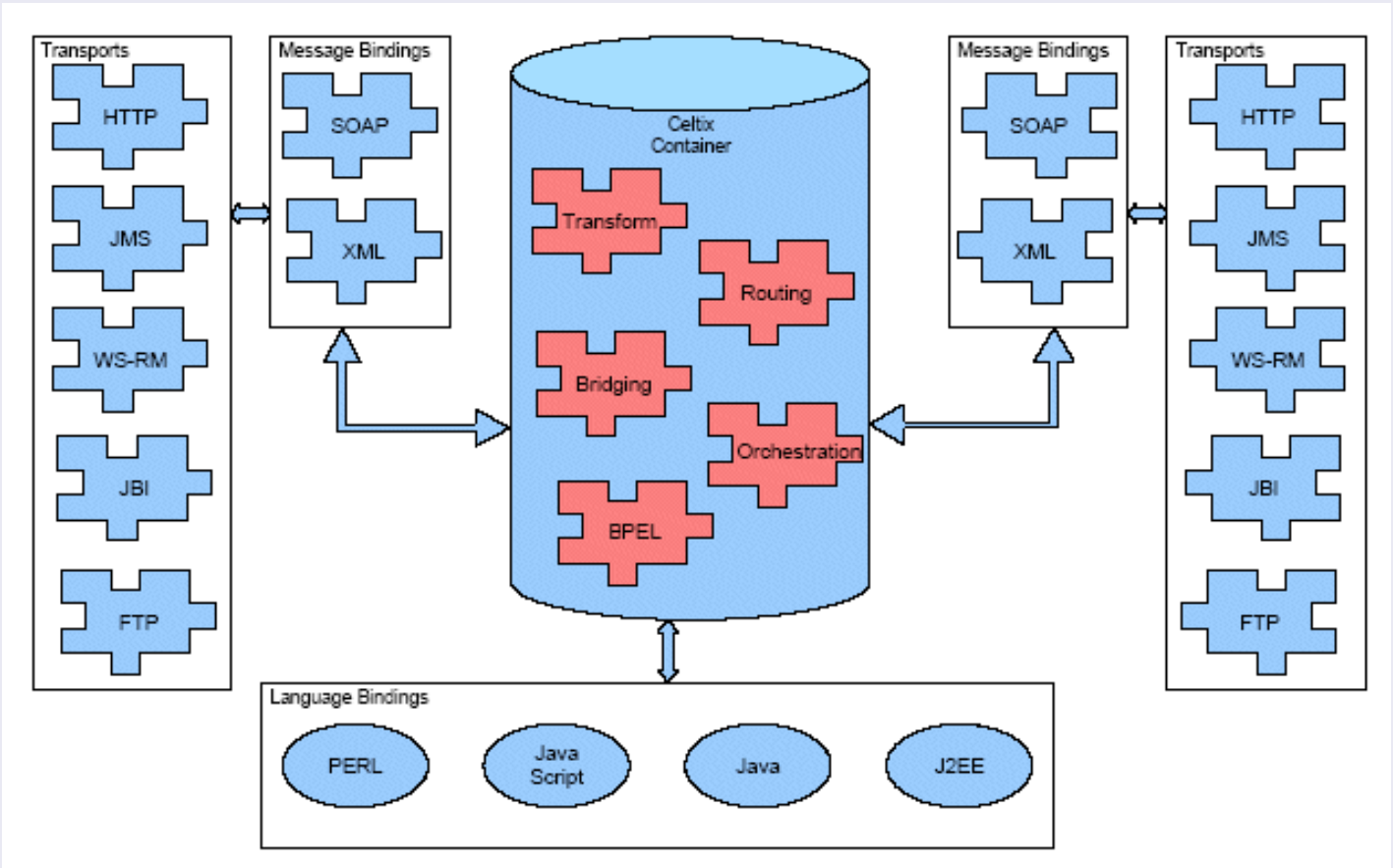
Java Business Integration

Intérêts

- Définition d'une nouvelle couche d'abstraction
  - Le container de container
  - Gage pour l'avenir : Standard extensible java container
- Une fois développé, peut être intégré partout
- Utilise un modèle de programmation simple basé sur :
  - Java, Web Services
  - XML
- Cache la tuyauterie d'intégration
  - Le développeur se consacre aux process métiers et aux échanges de données XML

Java Business Integration

Celtix : OpenSource JBI Object Web



Plan

Plan

Modèles à composants

Définition

Un composant est un objet, un module respectant un certain nombre de règles :

- Paramétrable : Un certain nombre de champs modifiables depuis des getters et des setters
- Sérialisable : Permet de conserver l'état du composant, offrant une persistance des données voire de l'application
- Réutilisable : Permet de limiter le nombre d'instantiations du composant

Modèles à composants

Définition

- Introspectif : Le composant doit être paramétrable dynamiquement, et permettre la découverte de son contenu (variables, méthodes) sans accès au code source
- Communiquant : Par exemple au travers d'écouteurs d'évennements

Modèles à composants

Exemple

- Les composants Swing en Java
- Les JavaBeans
- etc...

Modèles à composants

En Java

Un Bean doit :

- Être une classe publique
- Avoir un constructeur publique sans paramètres
- Implémenter l'interface **serializable**
- Définir des champs non publiques, accessibles par des getters et des setters, suivant les règles de nommage

Modèles à composants

Exemple

```
1  public class MonBean{
2
3  // Les champs de l'objet ne sont pas publics
4  private String propriete1;
5  private int propriete2;
6
7  //Constructeur publique sans parametres
8  public MonBean(){
9  this.propriete1="";
10 this.propriete2=0;
11 }
12
13 // Les proprietes de l'objet sont accessibles
14 // via des getters et setters publics
15 public String getPropriete1(){
16 return this.propriete1;
17 }
18
19 public int getPropriete2(){
20 return this.propriete2;
21 }
22
23 public void setPropriete1(String propriete1){
24 this.propriete1 = propriete1;
25 }
26
27 public void setPropriete2(int propriete2){
28 this.propriete2 = propriete2;
29 }
30 }
```

Bibliographie

- Cours de Middlewares, Jacques Fayolle, TSE
- http://goferproject.wordpress.com/2010/05/17/envoyer-et-recevoir-des-messages-en-java-avec-jms/
- Wikipédia
- Stackoverflow
- Oracle.com
- http://lionel.tricon.free.fr/Documentations/mpi/node16.html
- Le site du Zéro