

UE HAI704I

Architectures logicielles distribuées

Abdelhak-Djamel Seriai

seriai@lirmm.fr

- Le contenu de cette UE
 - Introduction aux architectures logicielles distribuées
 - Les architectures à objets distribués : RMI
 - Introduction aux architectures logicielles distribuées basées sur les web services
 - Les architectures logicielles distribuées basées sur les web services SOAP
 - Les architectures logicielles distribuées basées sur les web services REST
 - Les architectures logicielles distribuées basées sur les web services GRPC
 - Les architectures logicielles distribuées basées sur les Microservices
- Intervenants
 - Abdelhak-Djamel Seriai (seriai@lirmm.fr)
 - Bachar Rima (bachar.rima@lirmm.fr)
- MCC
 - CCI : 5 TP + 1 QCM

Cours 1 : Introduction aux Architectures Logicielles Distribuées (Réparties)

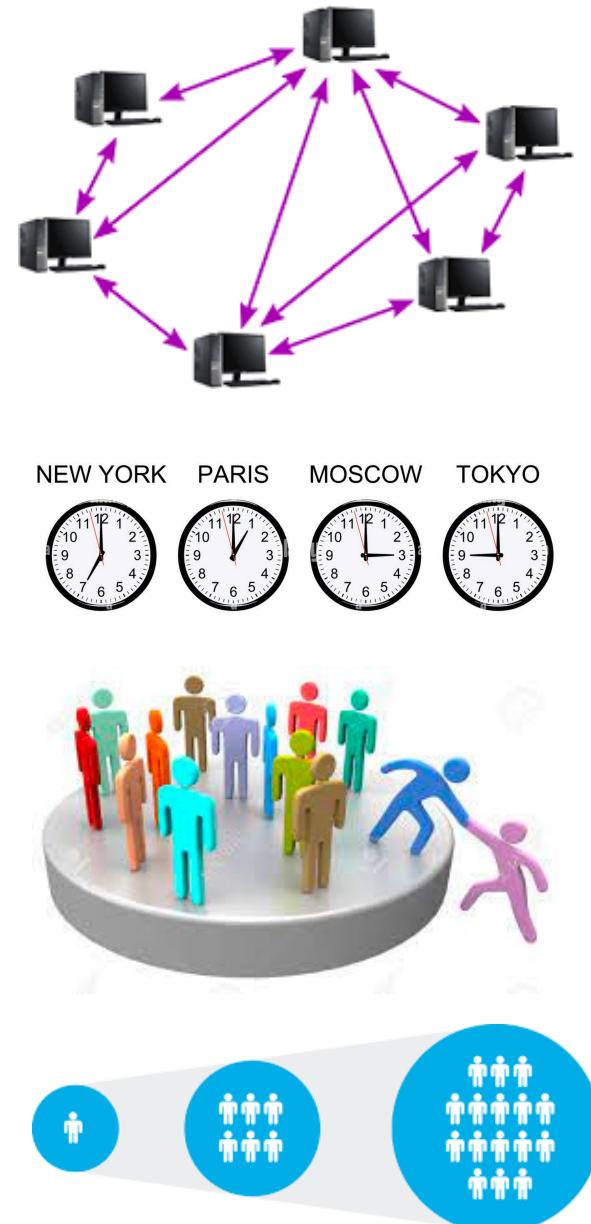
Définition 1

Un système distribué est un système formé de composants logiciels localisés sur des ordinateurs en réseau qui communiquent et coordonnent leurs actions uniquement par l'envoi de messages, le tout en vue de répondre à un besoin donné.

Définition 2

Ensemble composé d'éléments reliés par un système de communication ; les éléments ont des fonctions de traitement (processeurs), de stockage (mémoire), de relation avec le monde extérieur (capteurs, actionneurs).

- **Intégration et Communication** : Présenter le système comme un tout.
 - Propriété d'asynchronisme du système de communication : pas d'horloge commune
 - Conséquence : difficulté pour détecter les défaillances.
- **Hétérogénéités**
 - **Hétérogénéité matérielle et logicielle**
 - Gérer la diversité des matériels et logiciels en interaction.
 - **Hétérogénéité des données**
 - Echanger des données entre applications distribuées.
- **Ouverture & Passage à l'échelle (scalability)**
 - Découvrir et intégrer d'autres composants/parties.
 - Conserver l'efficacité une fois étendue (cf. internet).
 - Conséquence : difficulté pour définir un état global.
 - Difficulté pour administrer le système.
- **Gestion des défaillances**
 - Détection, masquage, tolérance, disponibilité...
- **Sécurité**
 - Confidentialité et intégrité



Définition

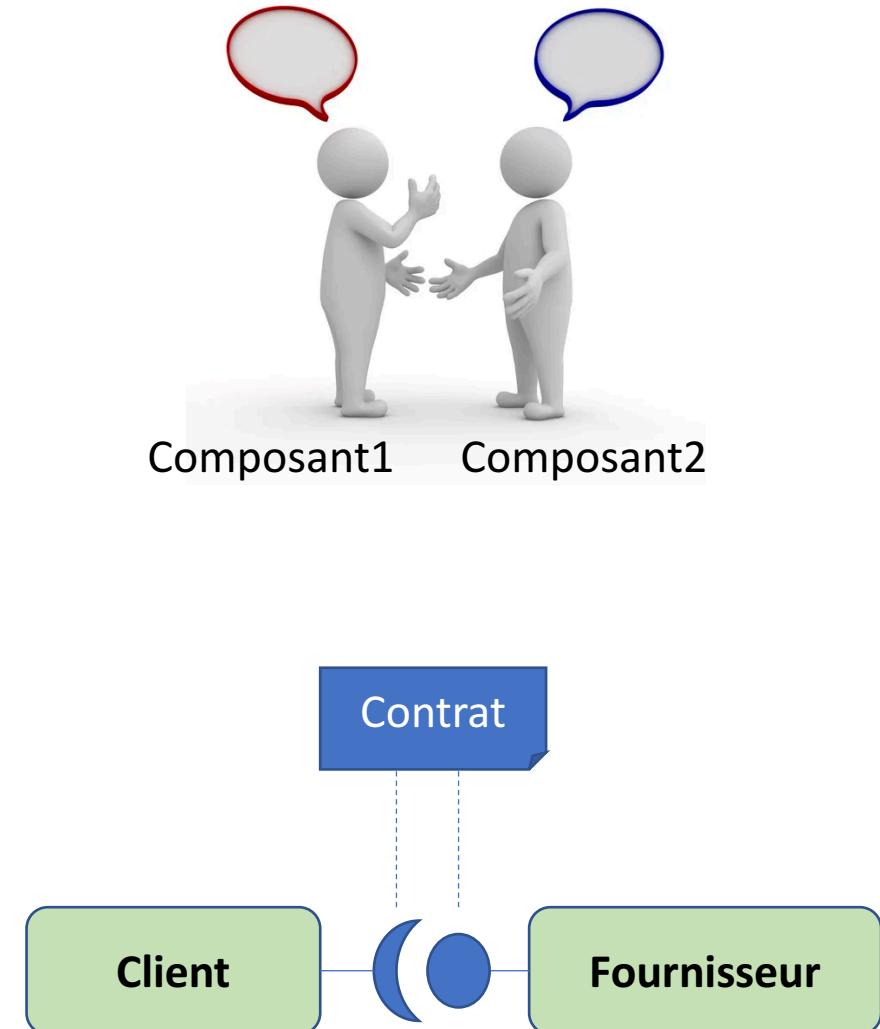
Une architecture est un ensemble de composants logiciels qui interagissent. Chaque composant offre un service.

Définition

Un service est un comportement défini par contrat, qui peut être implémenté et fourni par un composant pour être utilisé par un autre composant, sur la base exclusive du contrat. Un service est accessible via une ou plusieurs interfaces.

Définition

Une interface décrit l'interaction entre client et fournisseur du service.

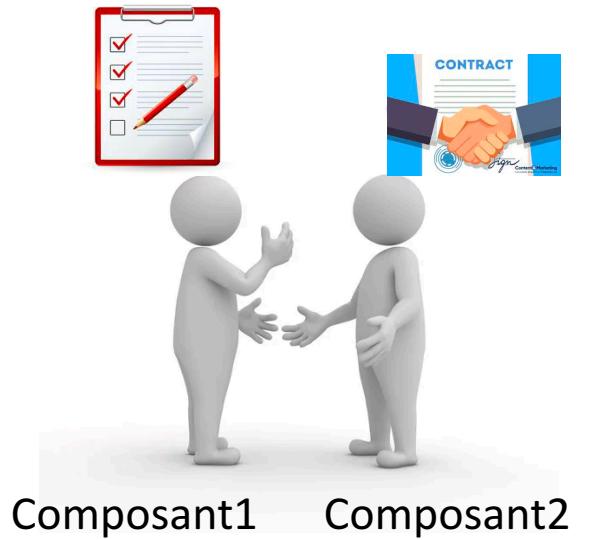


- **Deux éléments sont nécessaires pour définir une interface :**

- La liste des opérations et structures de données qui concourent à la réalisation du service
- Le contrat de service entre le client et le fournisseur : spécifie la compatibilité (conformité) entre ces deux interfaces
 - Conséquence : client ou fournisseur peuvent être remplacés du moment que le composant remplaçant respecte le contrat (est conforme)

- **La fourniture d'un service met en jeu deux interfaces :**

- Interface requise (côte client)
- Interface fournie (côte fournisseur)



• Mise en place des interfaces

• La liste des opérations

- Interface Definition Language (IDL) : S'appuie sur un langage existant
 - IDL CORBA sur C++
 - Voir <http://corba.dev/elopez.com/presentation/exemple/>
 - Java et C# définissent leur propre IDL



Quelles opérations fournies/requises ?



Mais, exprimée dans quel langage ?

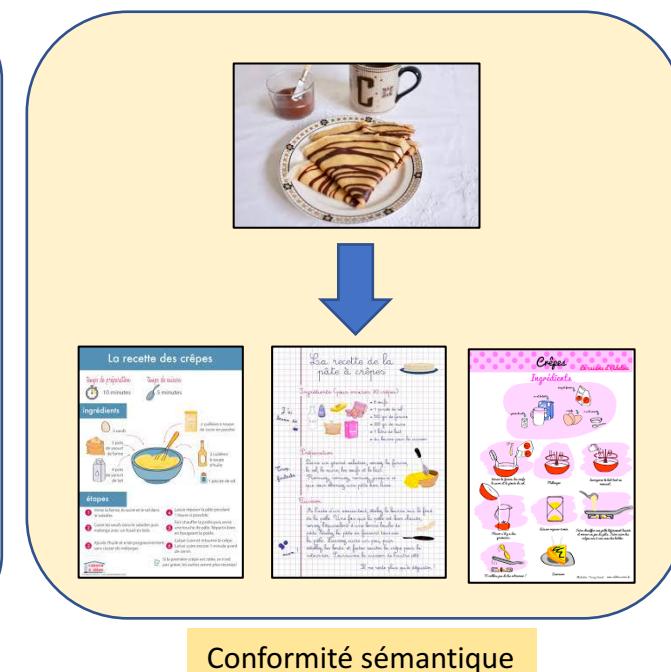
• Le contrat : Plusieurs niveaux de contrats :

- **Conformité syntaxique** : spécification de types.
- **Conformité sémantique** : spécification du comportement (1 méthode), utilisation des assertions.
- **Pour la synchronisation** : Sur les interactions entre méthodes.
- **QoS** : Sur les aspects non fonctionnels (performances, etc.).



Le français en 3 mots		
Québec	France	Suisse
Allô!	Salut!	Adieu!
Bienvenu	De rien	Service!
Bonjour!	Au revoir!	Tchüss!
Gomme	Chewing-gum	Chiclette*
Suce	Tétine	Lolette
En spécial	En promotion	En action
Scott towel	Sopalin	Papier-Ménage
Yapaslefeuaulac.ch		

Conformité syntaxique



Conformité sémantique



Synchronisation :
Quand faire une opération ?



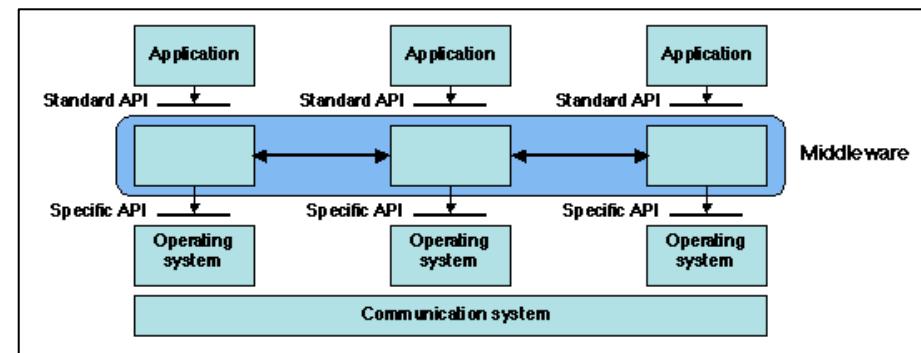
Qualité de service : QoS

• Motivations

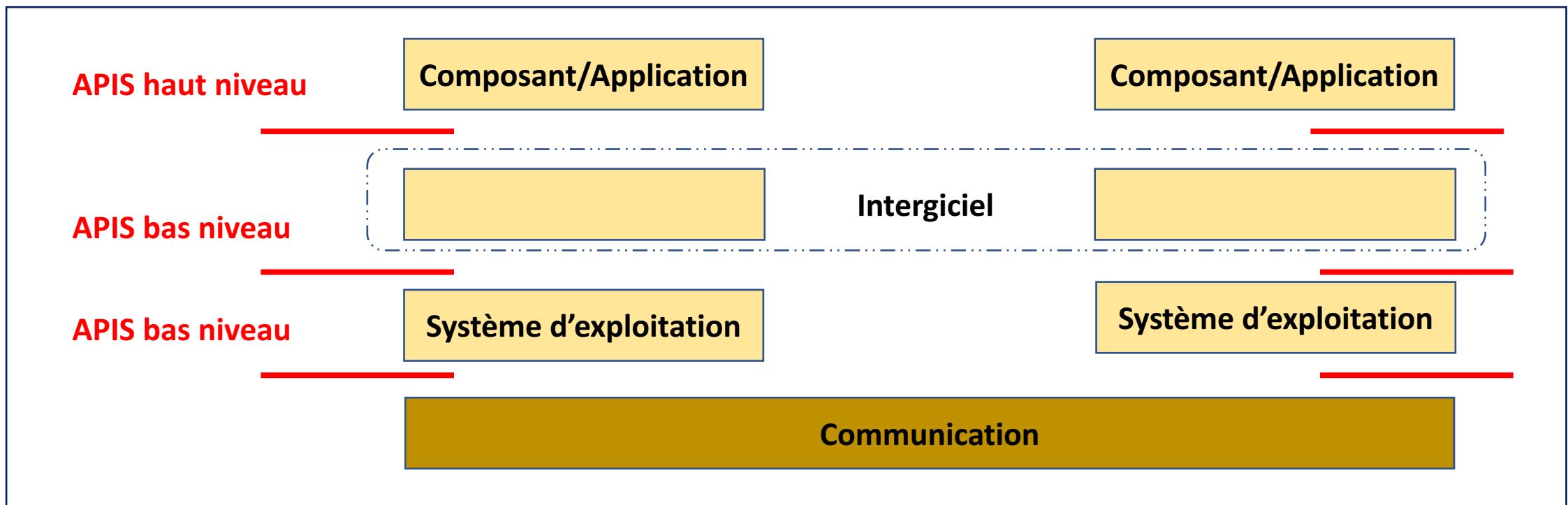
- Difficulté de s'appuyer sur les services de base de l'OS et réseau pour réaliser la communication entre composants (services) répartis.
 - Hétérogénéité.
 - Complexité des mécanismes (bas niveau).
 - Nécessité de gérer (et de masquer, au moins partiellement) la répartition.
- Solution : un nouveau composant – l'intergiciel.

• Définition

- Une couche de logiciel intermédiaire (repartie) entre les niveaux bas (systèmes et communication) et le niveau haut (applications).
- Joue un rôle analogue à celui d'un "super-système d'exploitation" pour un système repartì



- **Fournir une interface ou une API de haut niveau aux applications pour :**
 - Masquer l'hétérogénéité des systèmes matériels et logiciels sous-jacents.
 - Rendre la répartition aussi invisible ("transparente") que possible.
 - Fournir des services repartis d'usage courant.



- Plusieurs critères de classification :

- Unité de répartition :

- Objets repartis
 - Composants

- Degré de l'hétérogénéité supportée

- OS
 - Langage
 - Application

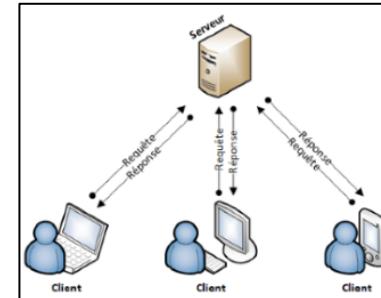
- Style architectural

- Nature du flux de contrôle

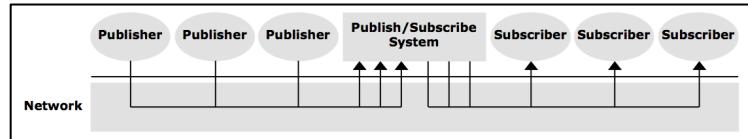
- Synchrone (client-serveur)
 - Asynchrone (événements)
 - Mixte

- Structure statique ou dynamique

- Mobilité des éléments
 - Reconfiguration



Synchrone



Asynchrone

L'intergiciel (middleware): Classification

• Classes d'intergiciel

• Unité de répartition

- Objets repartis
 - Java RMI, CORBA, DCOM, .NET

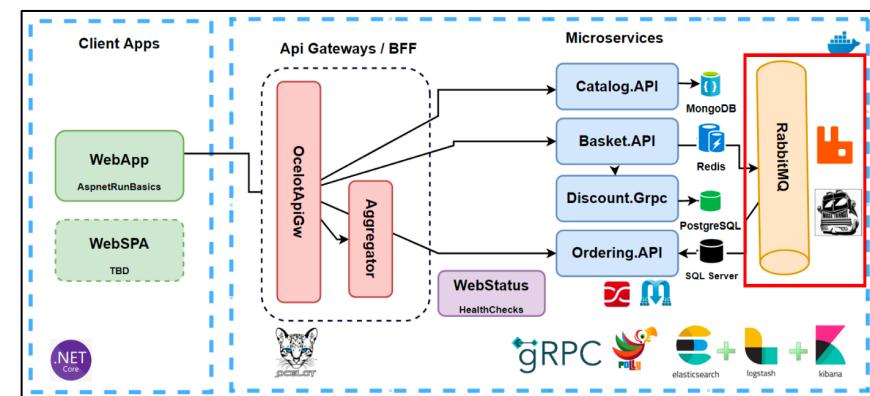
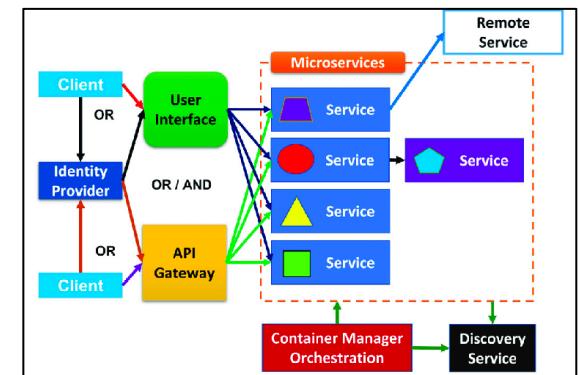
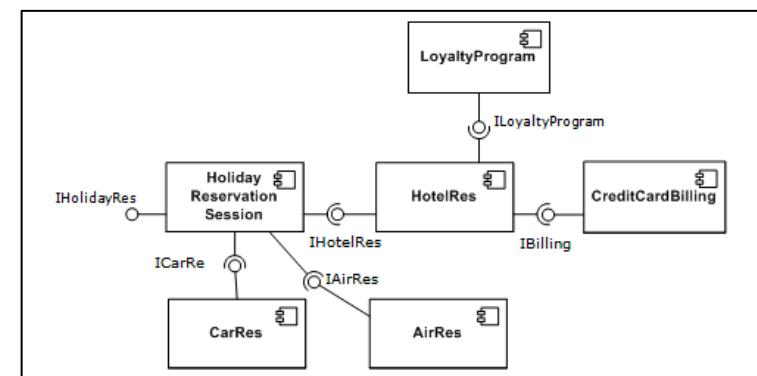
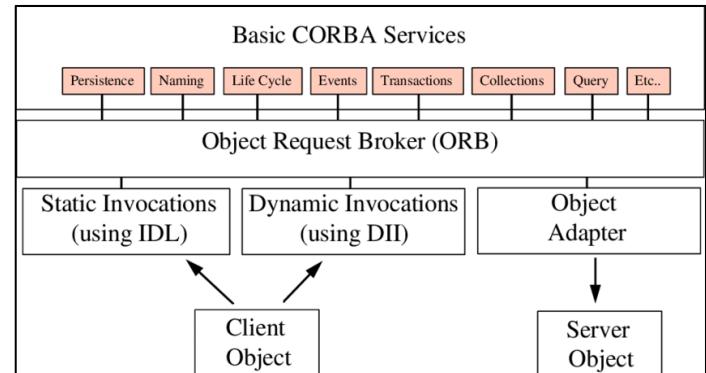
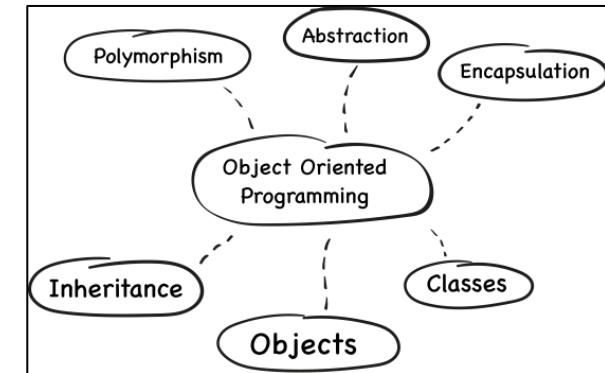
- Composants répartis
 - Java Beans, Enterprise Java Beans, CCM, MicroService/Conteneur

- Message-Oriented Middleware (MOM)

- Message Queues, Publish-Subscribe

- Intégration d'applications

- Web Services SOAP/REST



- **Classes d'intergiciel**

- **Degré d'hétérogénéité supportée**

- Hétérogénéité totale
 - OMG – CORBA, web services SOAP/REST/GRPC
- Hétérogénéité des langages
 - MicroSoft – DCOM, .NET Remoting
- Hétérogénéité des OS
 - JAVA RMI

• Styles architecturaux

- Les styles architecturaux spécifient, à grands traits, **comment organiser le code de l'application**.
- C'est le plus **haut niveau de granularité** et il spécifie les couches, les modules de haut niveau de l'application et comment ces modules et couches interagissent les uns avec les autres et les relations entre eux.
- **Exemples de styles architecturaux :**
 - Component-based
 - Monolithic application
 - Layered
 - Pipes and filters
 - Event-driven
 - Publish-subscribe
 - Plug-ins
 - Client-server
 - Service-oriented
 - Etc.

- **Exemples de styles architecturaux distribués**

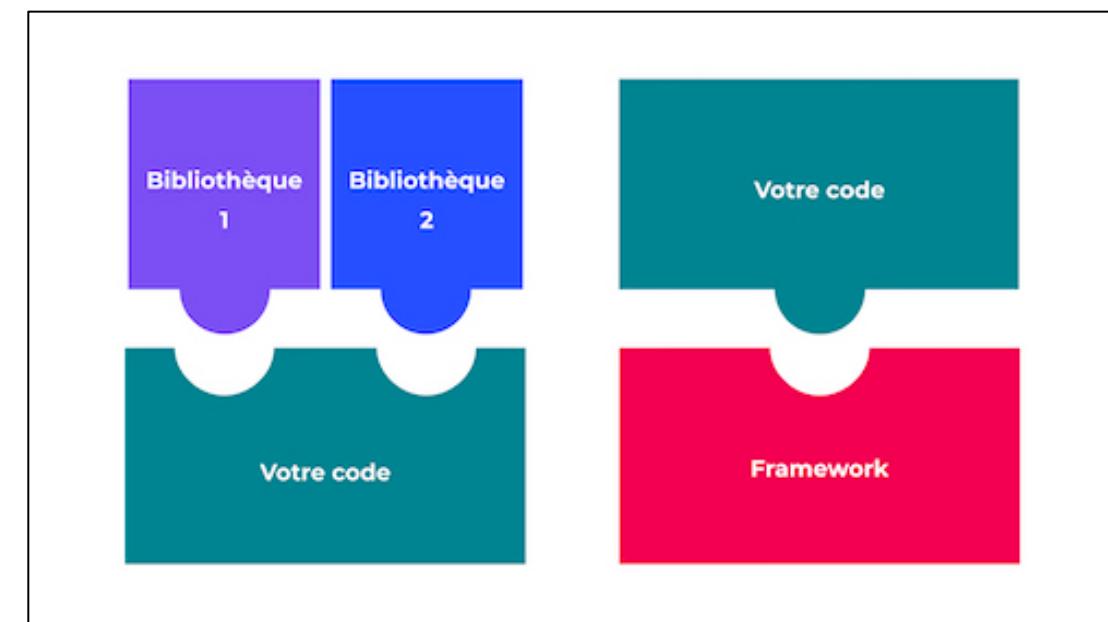
- Object request broker (Objets répartis)
- Client-server (n-tiers)
- Representational state transfer
- Service-oriented (SOA)
- Cloud computing/Micro-service
- Shared nothing architecture
- Space-based architecture
- Peer-to-peer
- Etc.

- **Les intergiciels sont souvent concrétisés sous forme de Frameworks, exemples :**

- Framework RMI
- Framework Corba
- Framework .NET/Web services/Remoting/
- etc.

- **Définition**

- Un canevas (Framework) est un « squelette » de programme qui peut être réutilisé (et adapté) pour une famille d'applications.
- Il met en œuvre des modèles, pas toujours explicites, d'architecture, de comportement, de structure.
- Dans les langages à objets, un canevas comprend:
 - Un ensemble de classes (souvent abstraites) devant être adaptées (par ex. par surcharge) à des environnements et contraintes spécifiques.
 - Un ensemble de règles d'usage pour ces classes.
- Les canevas réutilisent du code

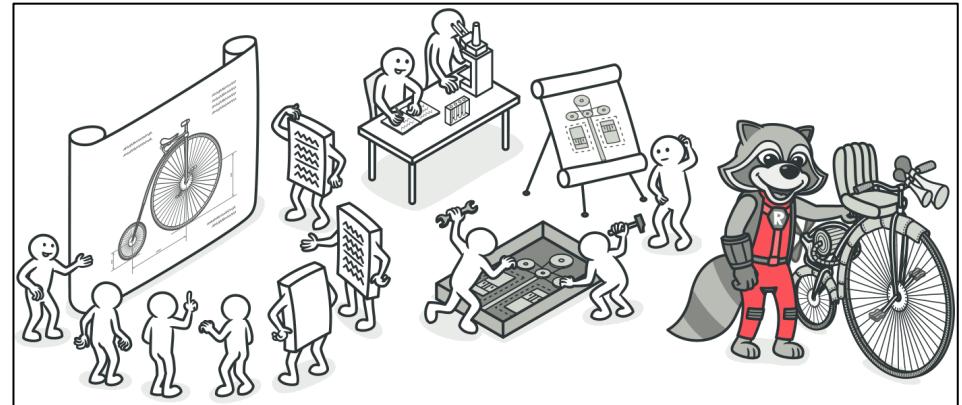


- **Patrons et canevas**

- Les patrons réutilisent un schéma de conception.
- Un canevas implémente en général plusieurs patrons

- **Quelques bases de construction des frameworks d'intergiciels:**

- **Proxy** : Patron de conception « représentant pour accès à distance »
- **Wrapper (Adapter)** : Patron de conception « transformation d'interface »
- **Factory** : Patron de conception « création d'objet »
- **Interceptor** : Patron d'architecture « adaptation de service »
- **Observer** : Patron de base pour l'asynchronisme



THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

- **Proxy (Mandataire)**

- **Contexte**

- Environnement reparti (pas d'espace unique d'adressage).
 - Applications constituées d'un ensemble d'objets repartis.
 - Un client accède à des services fournis par un objet pouvant être distant (le "servant").

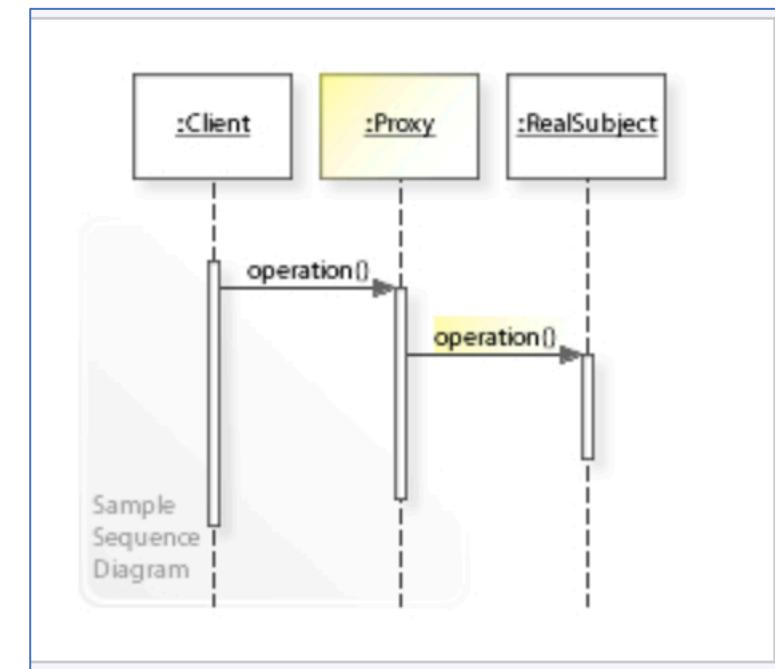
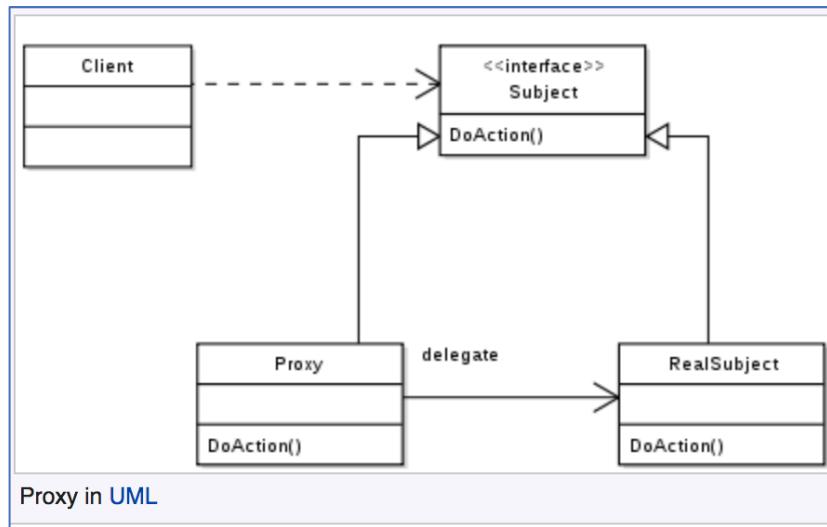
- **Objectif :**

- Définir un mécanisme d'accès qui évite au client :
 - Le codage "en dur" de l'emplacement du servant dans son code.
 - Une connaissance détaillée des protocoles de communication.

• Proxy (Mandataire)

• Solutions

- Utiliser un représentant local du servant sur le site client (isole le client du servant et du système de communication).
- Garder la même interface pour le représentant et le servant.
- Définir une structure uniforme de représentant pour faciliter sa génération automatique.



- **Wrapper/Adapdateur (ou Adapter)**

- **Contexte :**

- Des clients demandent des services ; des servants fournissent des services ; les services sont dénis par des interfaces

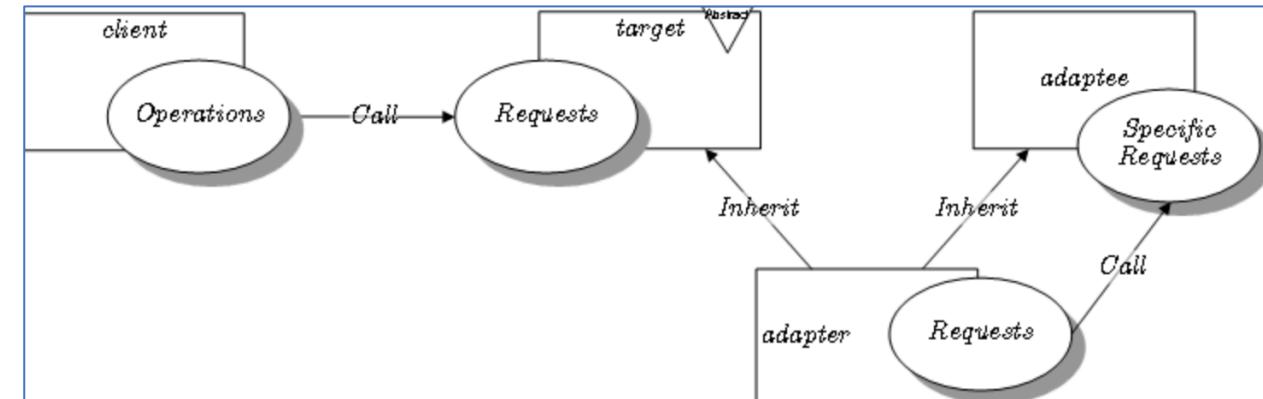
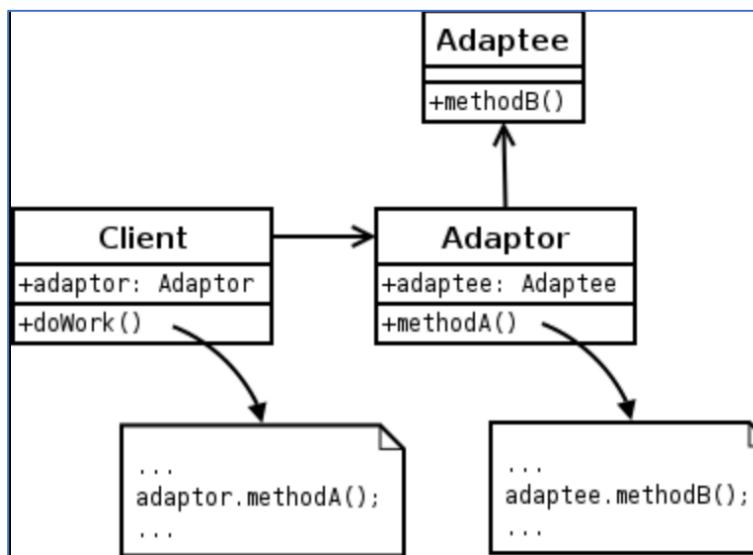
- **Objectif :**

- Réutiliser un servant existant en modifiant son interface et/ou certaines de ses fonctions pour satisfaire les besoins d'un client (ou d'une classe de clients).
- Propriétés souhaitables : doit être efficace ; doit être adaptable car les besoins peuvent changer de façon imprévisible ; doit être réutilisable (générique)

• Wrapper/adaptateur (ou Adapter)

• Solution :

- Le Wrapper/Adaptateur isole le servant en interceptant les appels de méthodes vers l'interface de celui-ci. Chaque appel est précédé par un prologue et suivi par un épilogue dans le Wrapper.
- Les paramètres et résultats peuvent être convertis



- **Factory (Fabrique)**

- **Contexte:**

- Application = ensemble d'objets en environnement reparti.
- Environnement reparti : pas d'espace d'adressage unique.

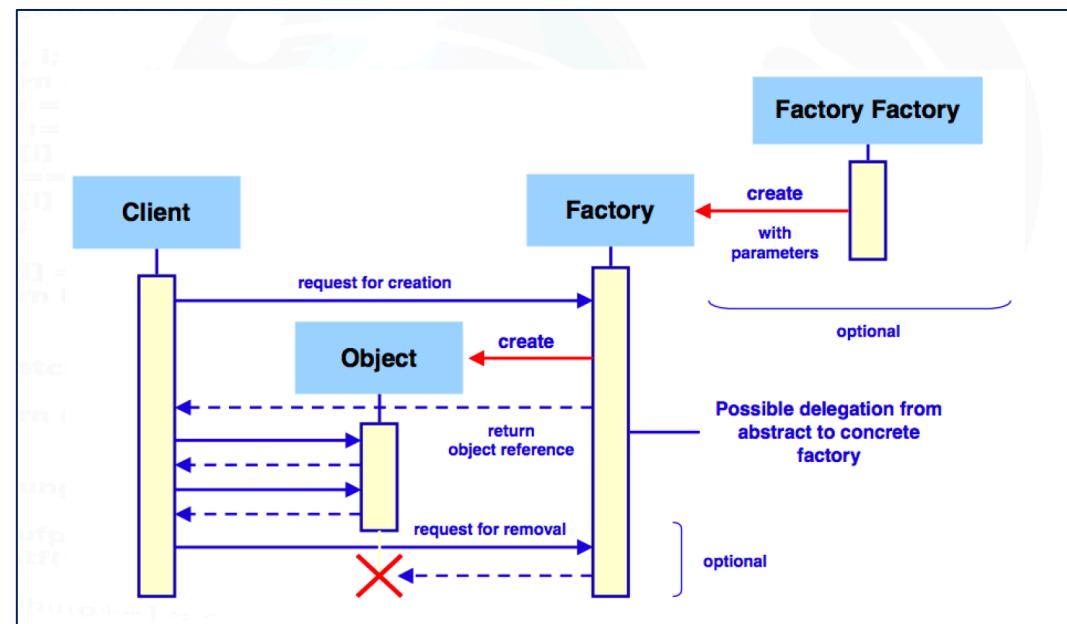
- **Objectif :**

- Créer dynamiquement des instances multiples d'une classe d'objets.
- Propriétés souhaitables :
 - Les instances doivent être paramétrables.
 - L'évolution doit être facile : pas de décisions "en dur".

• Factory (Fabrique)

• Solution :

- AbstractFactory : définit une interface et une organisation génériques pour la création d'objets ; la création effective est déléguée a des fabriques concrètes qui implémentent les méthodes de création.
- AbstractFactory peut être implémentée par Factory Methods : méthode de création redéfinie dans une sous-classe



• Intercepteur (Interceptor)

- Contexte :

- Fourniture de services (cadre général).
- Client-serveur, pair à pair, hiérarchique.
- Uni- ou bidirectionnel, synchrone ou asynchrone.

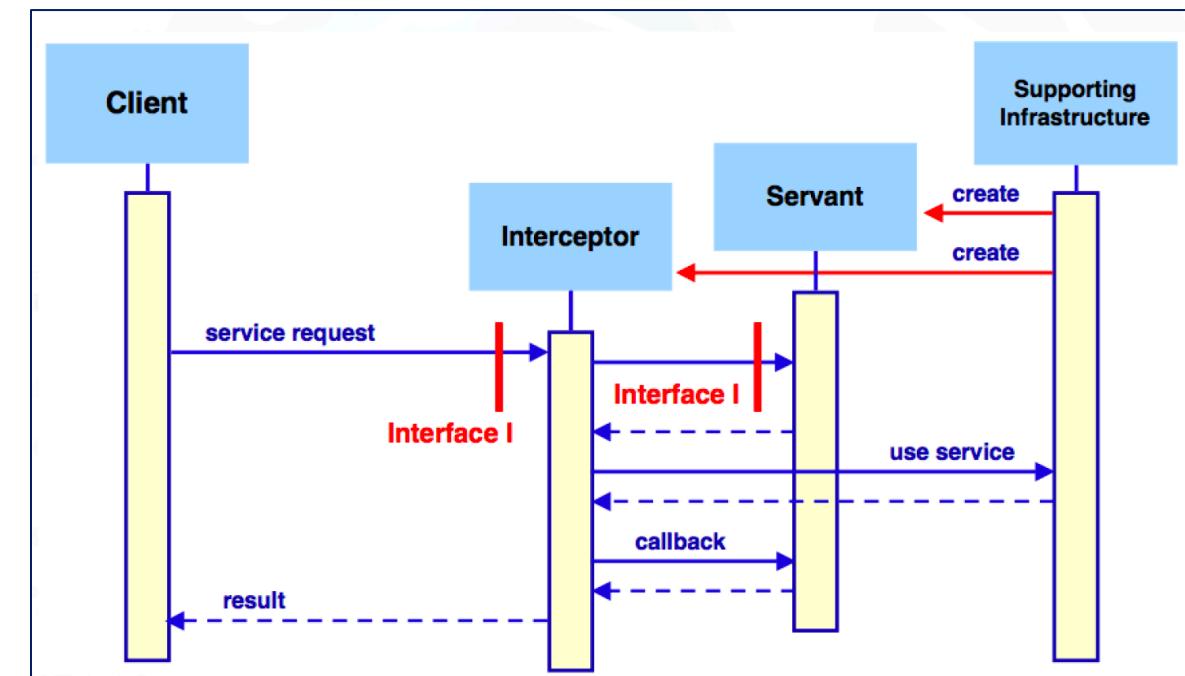
- Objectif :

- Transformer le service par différents moyens :
 - Interposer une nouvelle couche de traitement (cf. Wrapper).
 - Ajouter de nouvelles fonctions.
 - Changer (conditionnellement) la destination de l'appel ".
 - Les programmes client et serveur ne doivent pas être modifiés.
 - Les services peuvent être ajoutés ou supprimés dynamiquement.

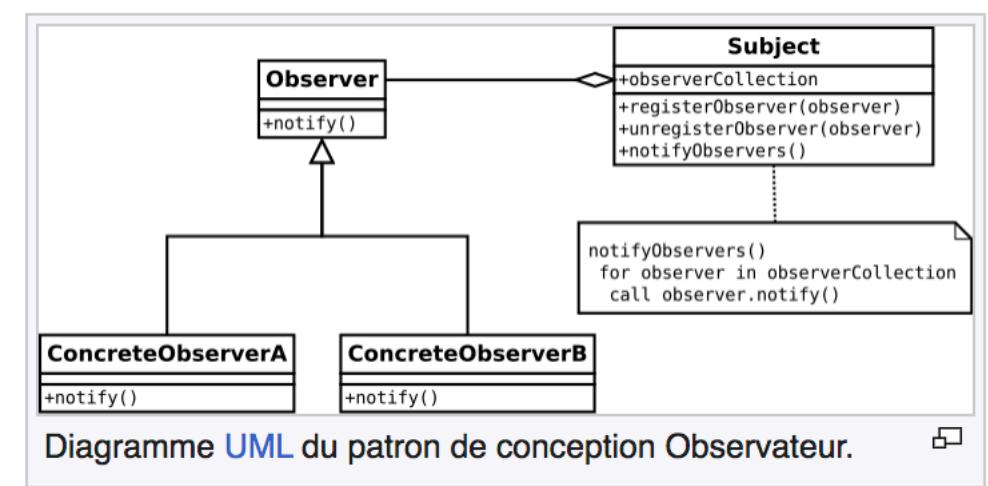
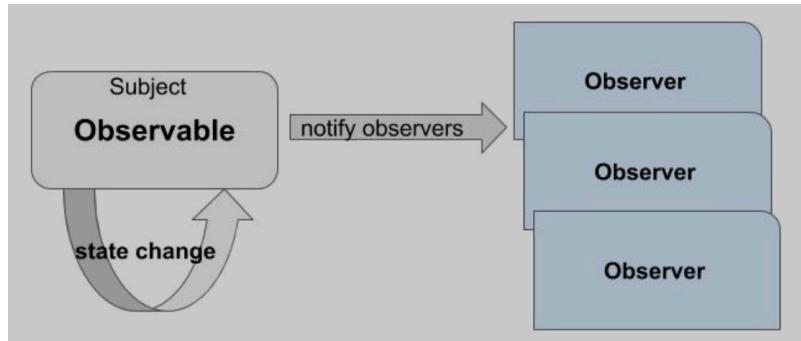
- **Intercepteur (Interceptor)**

- **Solution :**

- Créer des objets d'interposition
 - Ces objets interceptent les appels et insèrent des traitements spécifiques.
 - Peuvent rediriger l'appel vers une cible différente.
 - Peuvent utiliser des appels en retour.



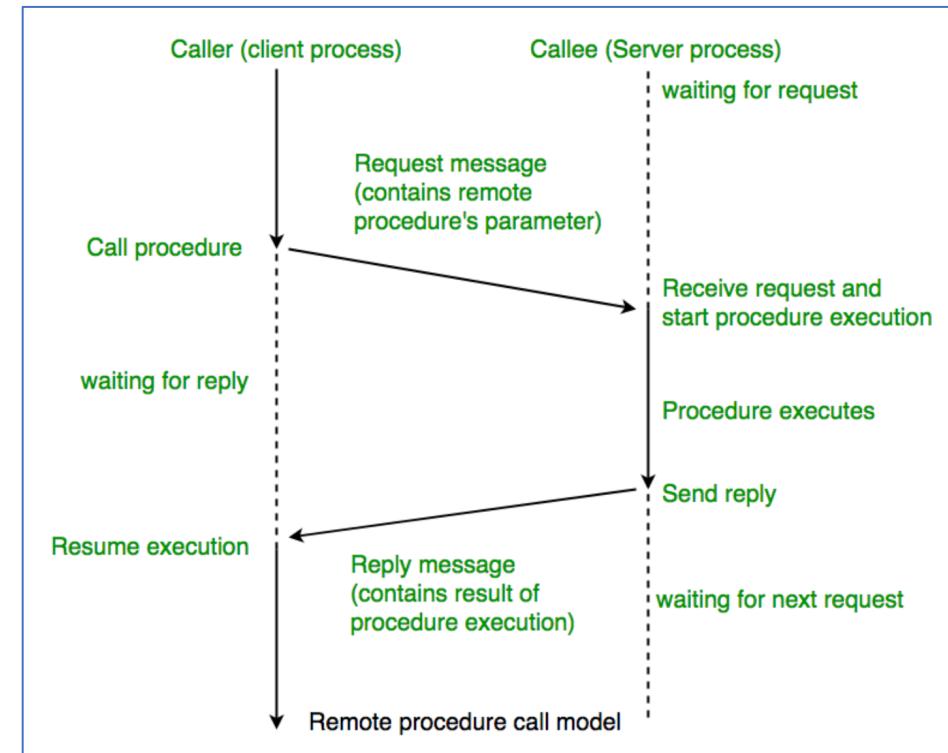
- Il est utilisé pour envoyer un signal à des modules qui jouent le rôle d'observateurs.
- En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les observables).
- Les notions d'observateur et d'observable permettent de limiter le couplage entre les modules aux seuls phénomènes à observer.
- Le patron permet aussi une gestion simplifiée d'observateurs multiples sur un même objet observable.
- Il est recommandé dès qu'il est nécessaire de gérer des évènements, quand une classe déclenche l'exécution d'une ou plusieurs autres.



- **Remote Procedure Call (RPC)**

- **Appel de procédures à distance entre un client et un serveur**

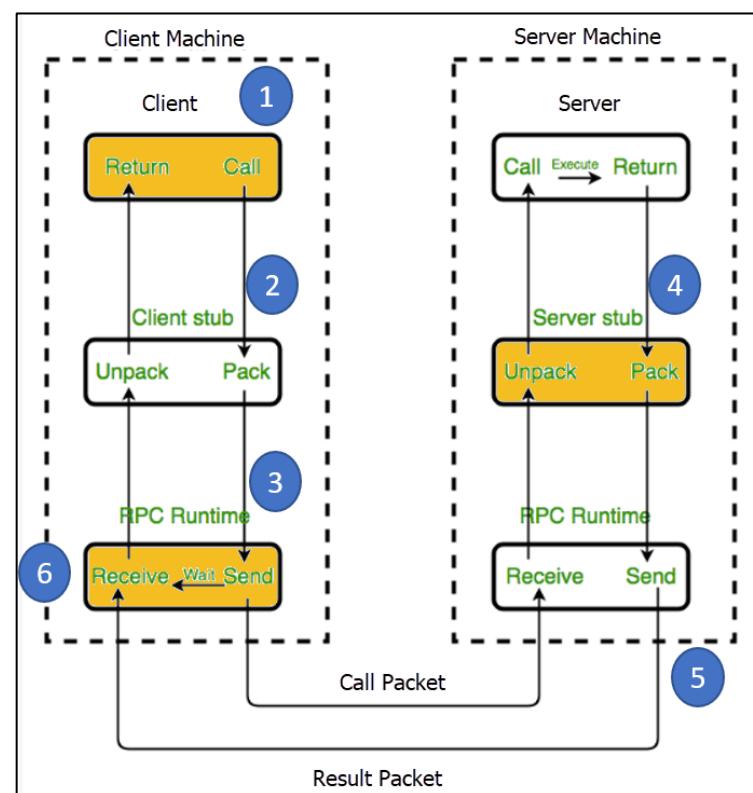
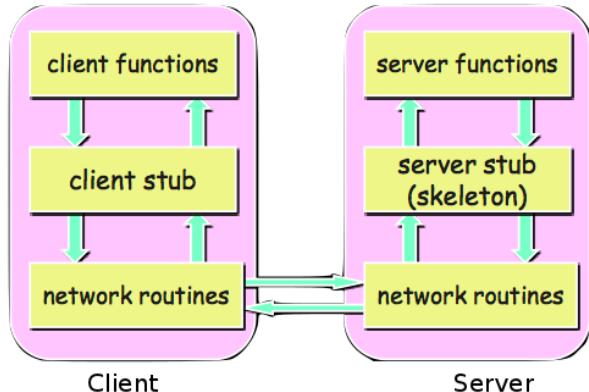
- Le client appelle une procédure que le serveur exécute, et en renvoie le résultat



- L'effet de l'appel doit être identique dans les deux situations.

Opérations RPC

- Le client appelle le stub (pousser les paramètres sur la pile).
- Le Stub (talon) package les paramètres en un message réseau.
- Message réseau envoyé au serveur.
- Recevoir le message : envoyer au talon.
- Dé-packer les paramètres, appeler la fonction du serveur.
- Retour de la fonction serveur.
- Packager la valeur retourner et envoie le message.
- Transférer le message sur le réseau.
- Recevoir le message : directement sur le talon.
- Dé-package le message reçu, retourner la valeur au client.



• Avantages du RPC

- Interface d'appel de procédure.
- Facile à utiliser.
- L'écriture d'applications est simplifiée.
 - RPC masque tout le code réseau dans des fonctions de stub.
 - Les programmeurs d'applications n'ont pas à se soucier des détails.
 - Sockets, numéros de port, ordre des octets.
- Fiable.
- Etc.

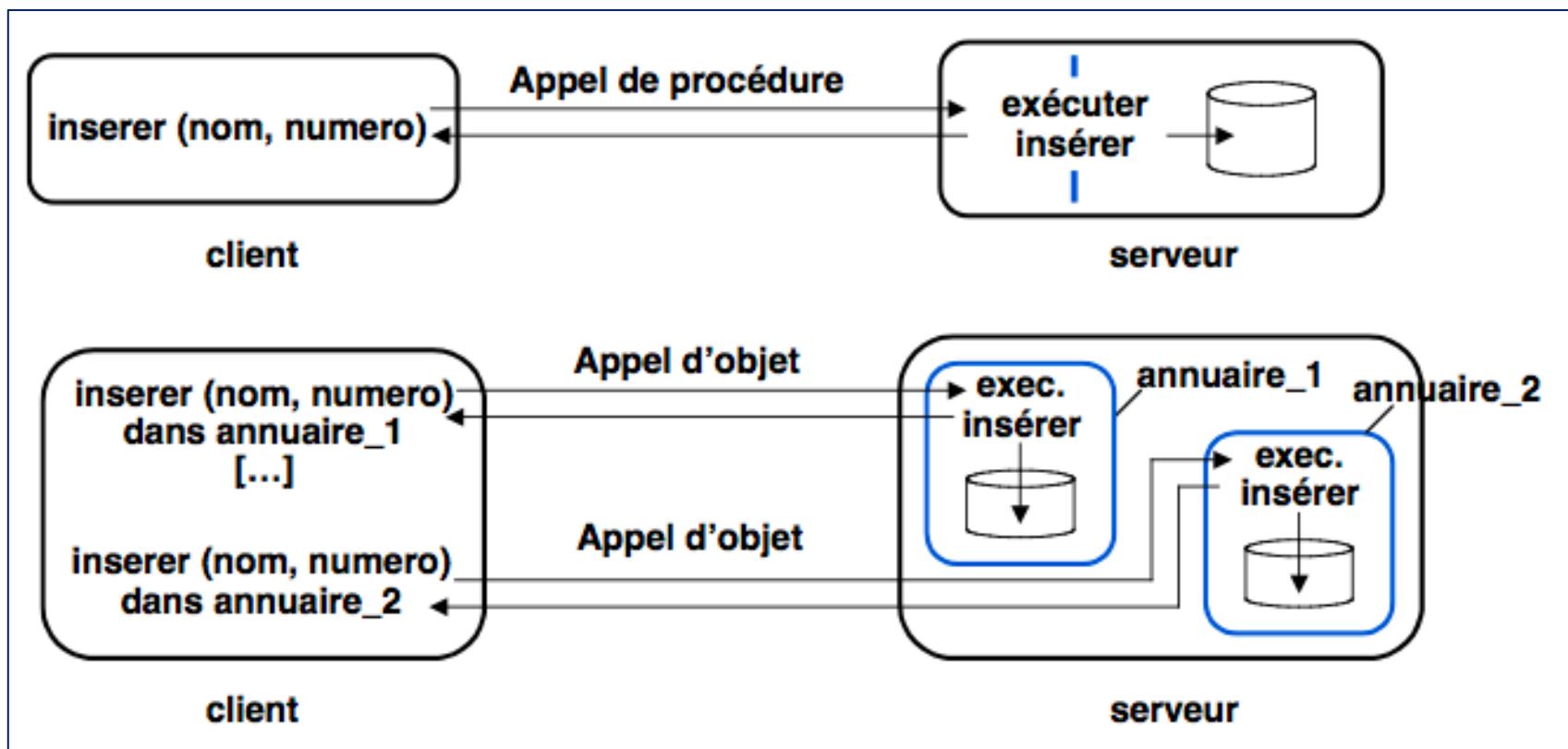
- **Intérêt des objets pour la construction d'applications reparties**
 - **Encapsulation :**
 - L'interface (méthodes + attributs) est la seule voie d'accès à l'état interne, non directement accessible
 - **Classes et instances :**
 - Mécanismes de génération d'exemplaires conformes à un même modèle
 - **Héritage :**
 - Mécanisme de spécialisation : facilite la récupération et la réutilisation de l'existant
 - **Polymorphisme :**
 - Mise en œuvre diverses des fonctions d'une interface
 - Remplacement d'un objet par un autre si interfaces "compatibles"
 - Facilite l'évolution et l'adaptation des applications

• Problèmes de l'exécution répartie des objets

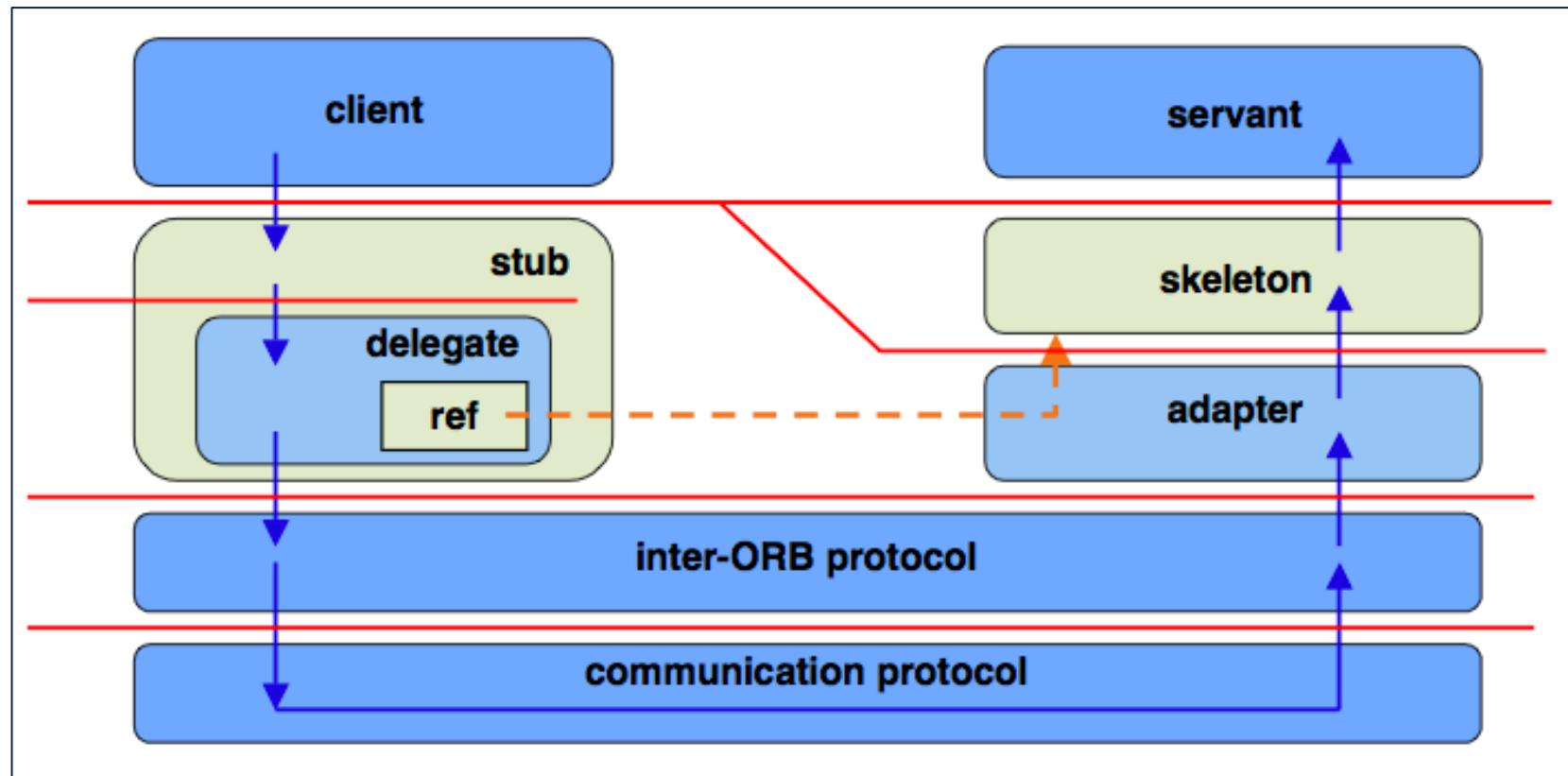
- Désignation et localisation des objets
 - Identification/référence
- Cycle de vie
 - Création, conservation, destruction des objets
- Liaison
 - Etablissement de la chaîne d'accès
- Schéma d'interaction/Communication
 - Synchronisation
- Mise en œuvre (réalisation, services).

- Application distribuée = ensemble d'objets repartis sur un réseau, communiquant entre eux (un objet est déployé intégralement sur un site)
- **Exemples**
 - Java Remote Method Invocation (RMI) : appel d'objets distants en Java - Sun
 - Common Object Request Broker Architecture (CORBA) : support pour l'exécution d'objets repartis hétérogènes OMG
 - DCOM, COM+, .Net Remoting : Distributed Common Object Model – Microsoft
- Schéma commun de solution : ORB (Object Request Broker - Courtier de demande d'objet)
 - Modèle de base : client-serveur.
 - Identification et localisation des objets.
 - Liaison entre objets clients et serveurs.
 - Exécution des appels de méthode à distance.
 - Gestion du cycle de vie des objets (création, activation, ...).
 - Services divers (sécurité, transactions, etc.).

- **Appel de procédure vs appel de méthode sur un objet**
 - Exemple : insérer une entrée dans un annuaire

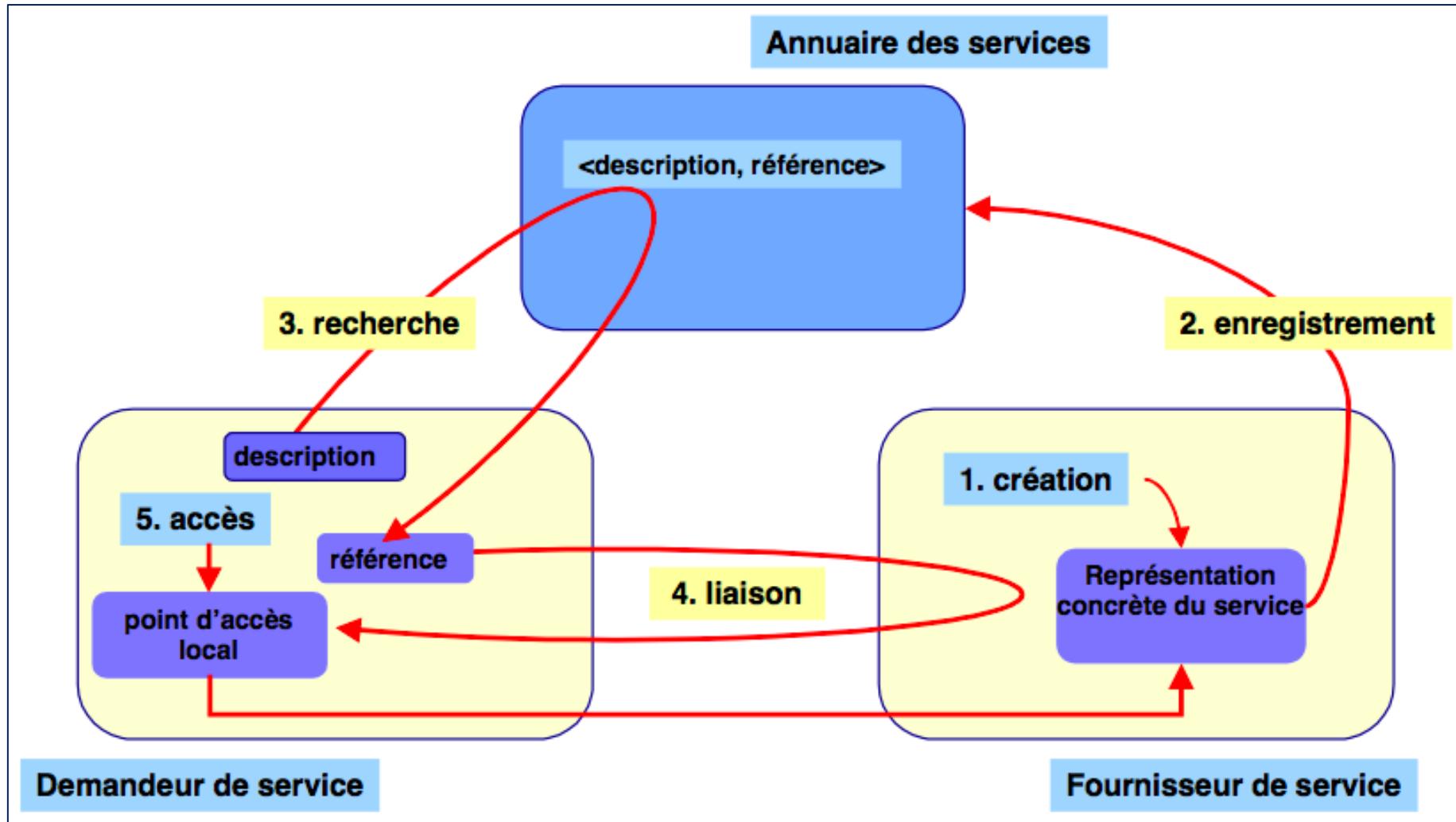


- Structure d'un appel distant



Intergiciel pour objets repartis

• Accès à un service



- Livre collectif : Intergiciel et Construction d'Applications Réparties.
- Cours : Environnement de distribution et architecture repartie
- DISTRIBUTED SYSTEMS, Concepts and Design. George Coulouris & al. Addison Wesley. 2012.
- Distributed Systems: Principles and Paradigms. Maarten van Steen & al. Pearson edition, 2006.
- Distributed Systems, [R15A0520], LECTURE NOTES, MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
- Introduction to Distributed Systems. SABU M. THAMPI