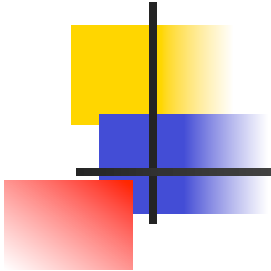


Des sockets aux objets répartis: BSD sockets, RMI, CORBA

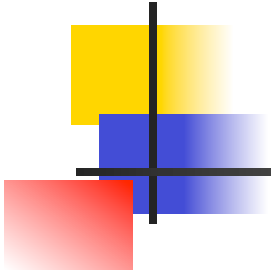
A. Diaconescu
B. Dupouy
L. Pautet



Plan du cours

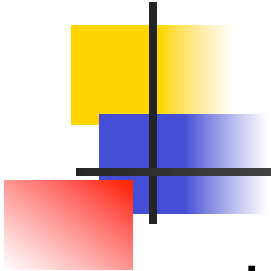
- BSD Sockets
- Concepts et définitions des intergiciels
- Java RMI
- CORBA

- UEs de spécialités approfondissant ces mécanismes:
 - INF341: Spécifications, modélisation et conception de systèmes logiciels
 - INF342: Systèmes Temps Réel
 - INF346: Systèmes Répartis



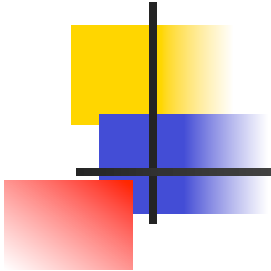
IP, UDP et TCP

- L' IETF a défini une série de protocoles : IPv4 et IPv6 pour l' adressage des machines; TCP et UDP échanger des messages au dessus de IP :
 - Ils sont une norme *de facto* pour la partie « utilisateur » du réseau, celle qui est directement accessible,
 - Les autres protocoles relèvent du cœur du réseau, ou des couches applicatives (e.g. telnet, DHCP, ...)
 - L' IETF ne définit qu' un format de messages et les automates pour établir des connexions, échanger des messages, gérer les erreurs, la fragmentation, ...



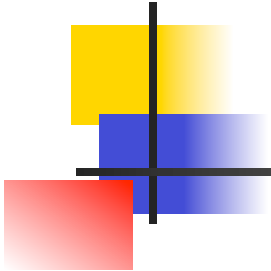
Rappels: modes d'adressage IPv4

- Les adresses sont sur 4 octets, en notation dite pointée :
 - 137.194.2.34, avec netid = 137.194, hostid = 2.34
- Deux éléments composent l'adresse : netid et hostid
 - Netid : identifiant du réseau
 - Hostid : identifiant de l'hôte sur le réseau
- Utilisé pour le routage des paquets
- IPv6 lève les limites d'adressage : les adresses passent de 4 à 6 octets, et ajoute des mécanismes de configuration



Rappels: modes d'adressage IPv4

- Les netid sont répartis en classes :
 - Classe A, netid codé sur 1 octet :
 - les adresses réseau vont de 1.0.0.0 à 126.0.0.0 (127 : réservé à localhost),
 - Classe B, sur 2 octets (les deux premiers bits sont à 1 et 0) :
 - les adresses réseau vont ainsi de 128.0.0.0 à 191.255.0.0
 - Classe C, sur 3 octets (les trois premiers bits sont à 1, 1 et 0) :
 - les adresses réseau vont de ainsi 192.0.0.0 à 223.255.255.0
 - Classe D (multicast), netid sur un octet, de 224 à 239

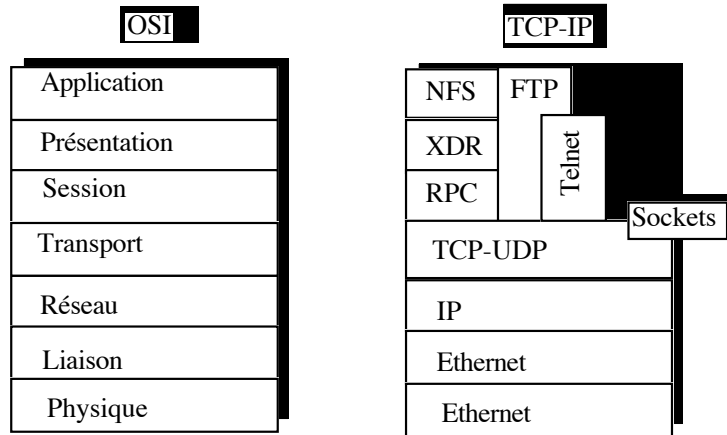


Rappels: modes d'adressage IPv4

- Adresses spéciales
 - 127.0.0.1 : « localhost », bouclage local
 - 0.0.0.0 : adresse de destination invalide
 - 255.255.255.255 : adresse de diffusion

- Adresses réservées pour les réseaux locaux (non accessibles depuis l'extérieur) :
 - Classe A : netid 10, hostid de 0.0.1 à 255.255.254
 - Classe B : netid 172.16 à 172.31, hostid de 0.1 à 255.254
 - Classe C : netid 192.168.0 à 192.167.255, hostid de 1 à 254

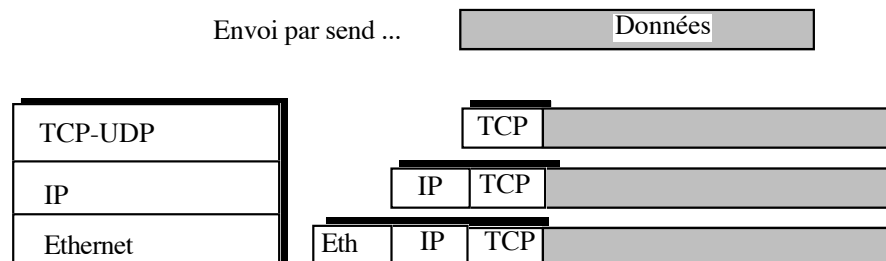
Modèle OSI vs TCP/IP

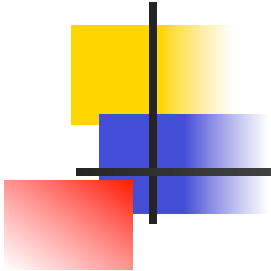


■ Exemple de services sur IP :

- ftp-data 20/tcp
- ftp 21/tcp
- telnet 23/tcp
- smtp 25/tcp
- time 37/tcp
- sunrpc 111/udp
- sunrpc 111/tcp

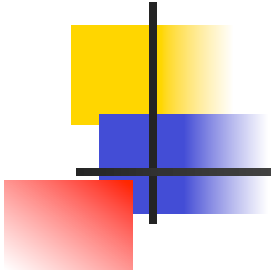
Structure des trames:





TCP vs UDP

- TCP est un protocole orienté connexion au dessus de IP
 - Protocole fiable (avec gestion des erreurs)
 - Paquets de taille maximale (MTU), avec un mécanisme de segmentation pour les données volumineuses
 - Mécanisme de gestion de flux pour éviter de saturer le réseau (algo. de Nagle)
- UDP est un protocole plus simple que TCP
 - Si, « tout va bien » on évite la complexité de TCP
 - On perd le séquençement des paquets, la gestion du flux
 - Utile pour les applications légères, le temps réel mou (multimédia)

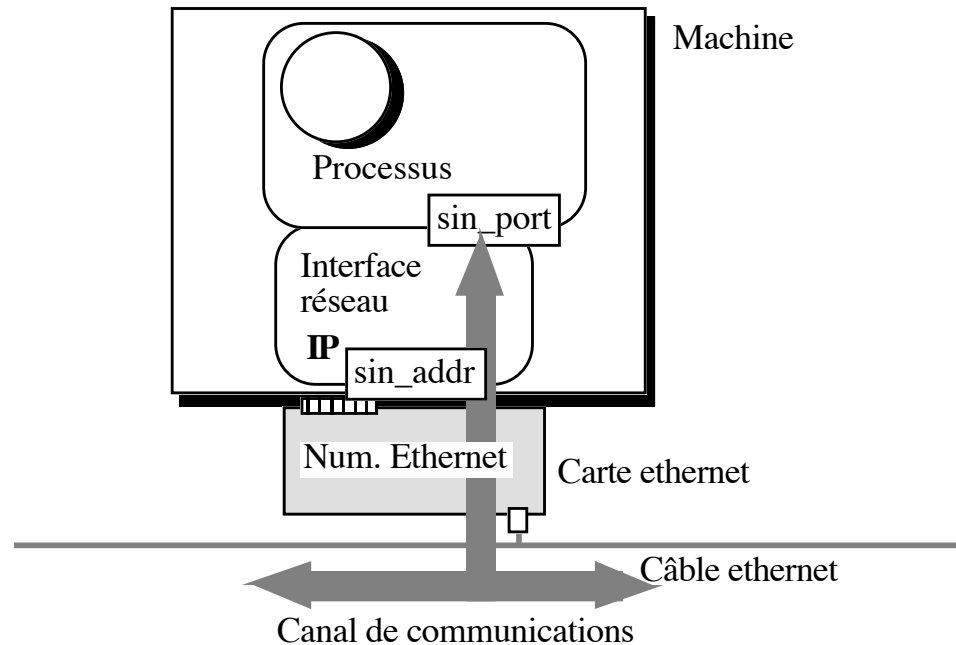


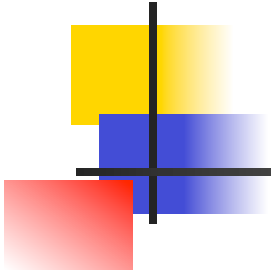
BSD Sockets, une API pour IP, TCP et UDP

- Aucune API n'est définie par l'IETF, BSD Sockets est le modèle le plus couramment employé dans le monde Unix
 - Repris sur la plupart des plates-formes, dont Windows
- Calquée sur le modèle de ressources Unix : tout est fichier
 - Les sockets suivent la sémantique producteur/consommateur et s'utilisent avec *open/read/write*
 - Liens forts avec l'API standard pour le cycle de vie de la connexion

Domaine des sockets: AF_INET

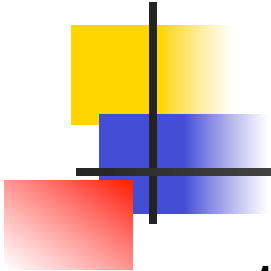
- Dans AF_INET, le socket est un port vers la couche 4 du protocole Internet (IP) :





Mode connecté : TCP

- L'appelant (ou client) :
 - crée une socket ;
 - construit l'adresse réseau (IP+port) du serveur (connu généralement par le **nom**)
 - Pas d'annuaire pour les services, seulement pour les noms de sites (DNS)
 - se connecte au serveur en donnant l'adresse Internet du serveur et le numéro de port du service. Cette connexion attribue automatiquement un numéro de port au client ;
 - lit ou écrit sur la socket ;
 - ferme la socket.
- L'appelé (ou serveur) :
 - crée une socket ;
 - **associe** une adresse socket (adresse Internet et numéro de port) au service ;
 - se met en attente des connexions entrantes ;
 - **pour chaque** connexion entrante :
 - "accepte" la connexion (une nouvelle socket est créée);
 - lit ou écrit sur la nouvelle socket ;
 - ferme la nouvelle socket.



Serveur C multi-tâches TCP (UNIX)

- 1- création d'un port d'écoute et attente d'une demande d'établissement de connexion
- 2- gestion concurrente des communications avec ces clients

```
struct sockaddr ServerAddr, ClientAddr;
int ClientAddrLen;

// creation d'un port d'ecoute
ServerSocket = socket(PF_INET, SOCK_STREAM, 0);
bind(ServerSocket, &ServerAddr, sizeof(struct sockaddr));

while (1){// Boucle d'attente sur le port d'ecoute
    // accept est bloquant
    Socket = accept(ServerSocket, &ClientAddr, &ClientAddrLen);
    if (fork() == 0){
        close(ServerSocket);
        Handle(Socket);
    }
    close(Socket);
}
```

Canevas d'un client TCP

- 1- construire l'adresse du serveur
- 2- demander l'établissement de la communication

```
// Pas de bind : l'appelant n'a pas besoin de sa propre adresse réseau  
Socket = socket(PF_INET, SOCK_STREAM, 0);
```

```
struct sockaddr ServerAddr;
```

```
...
```

```
// Trouver l'adresse IP du serveur
```

```
ServerAddr.sin_addr = gethostbyname(« www.enst.fr »);
```

```
// Pas d'annuaire pour les services !!!
```

```
ServerAddr.sin_port = ServerPort;
```

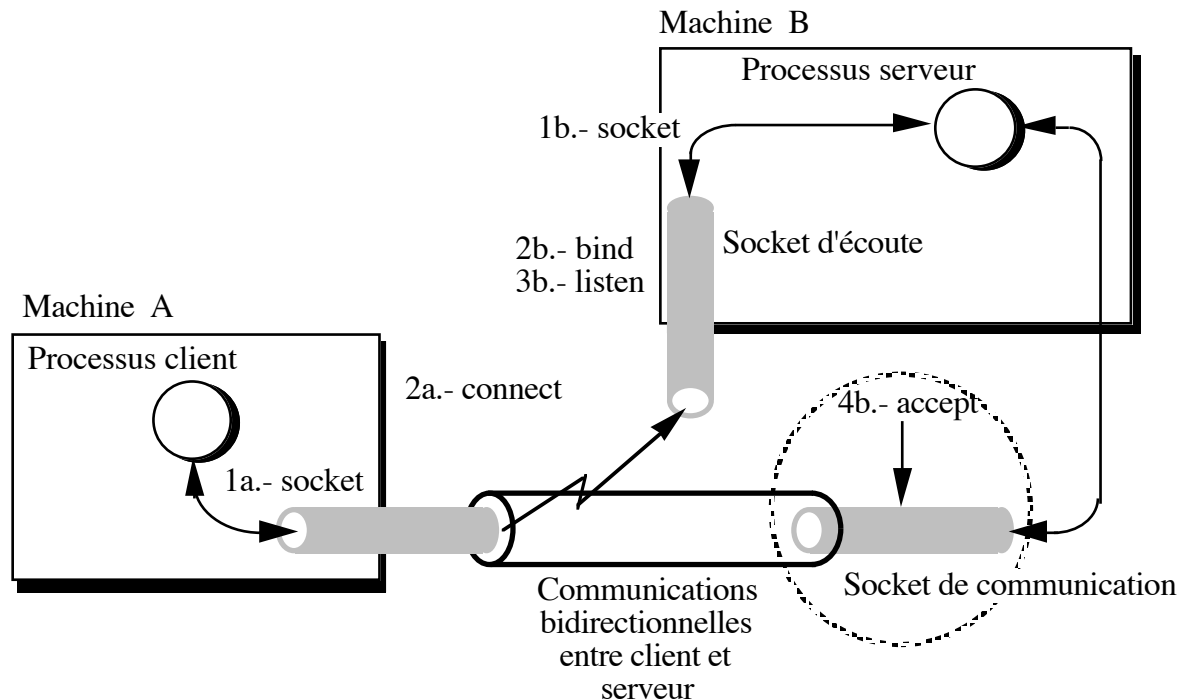
```
// Connecter au serveur (éventuellement timeout)
```

```
connect(Socket, &ServerAddr, sizeof(struct sockaddr));
```

```
// Après connexion, dialoguer en utilisant read ou write sur Socket;
```

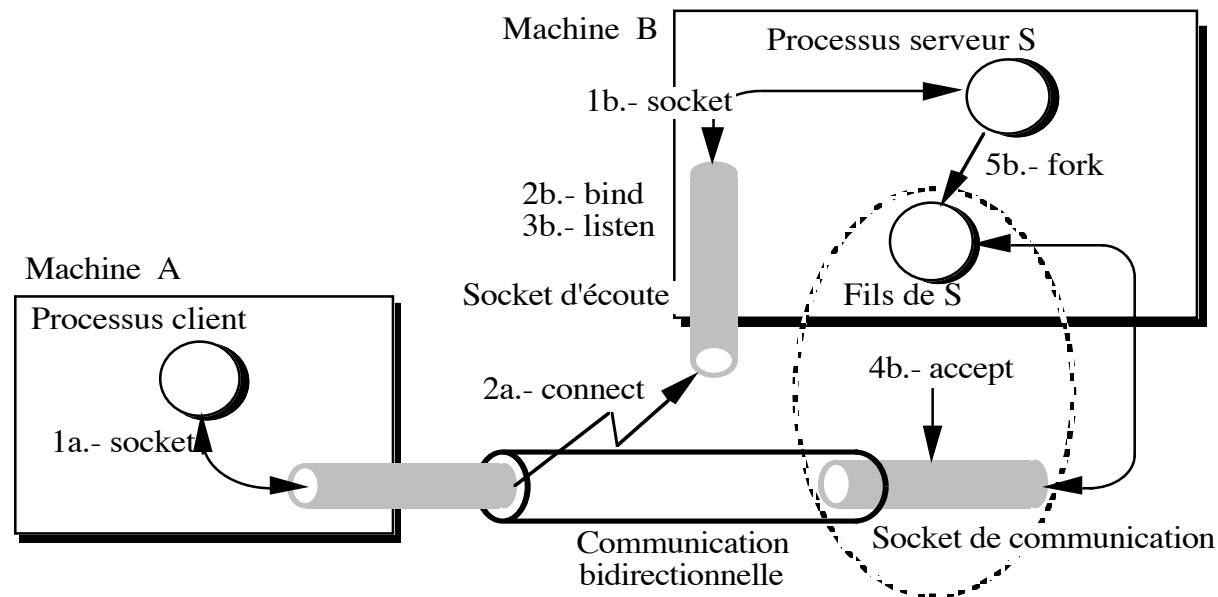
Mode connecté

- Schéma fonctionnel de la communication de base



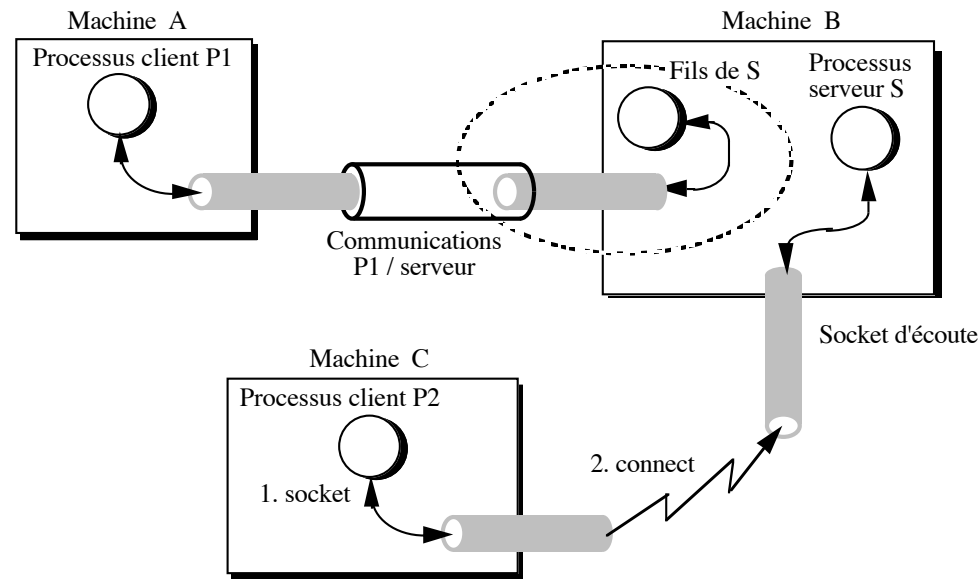
Mode connecté : gestion concurrente des clients (1/2)

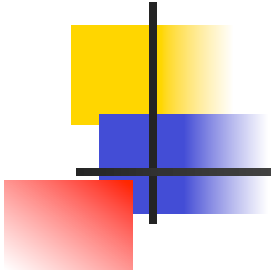
- Communication avec un premier client



Mode connecté : gestion concurrente des clients (2/2)

- Exemple : « acceptation » d'une communication pendant le dialogue avec un client déjà connecté :



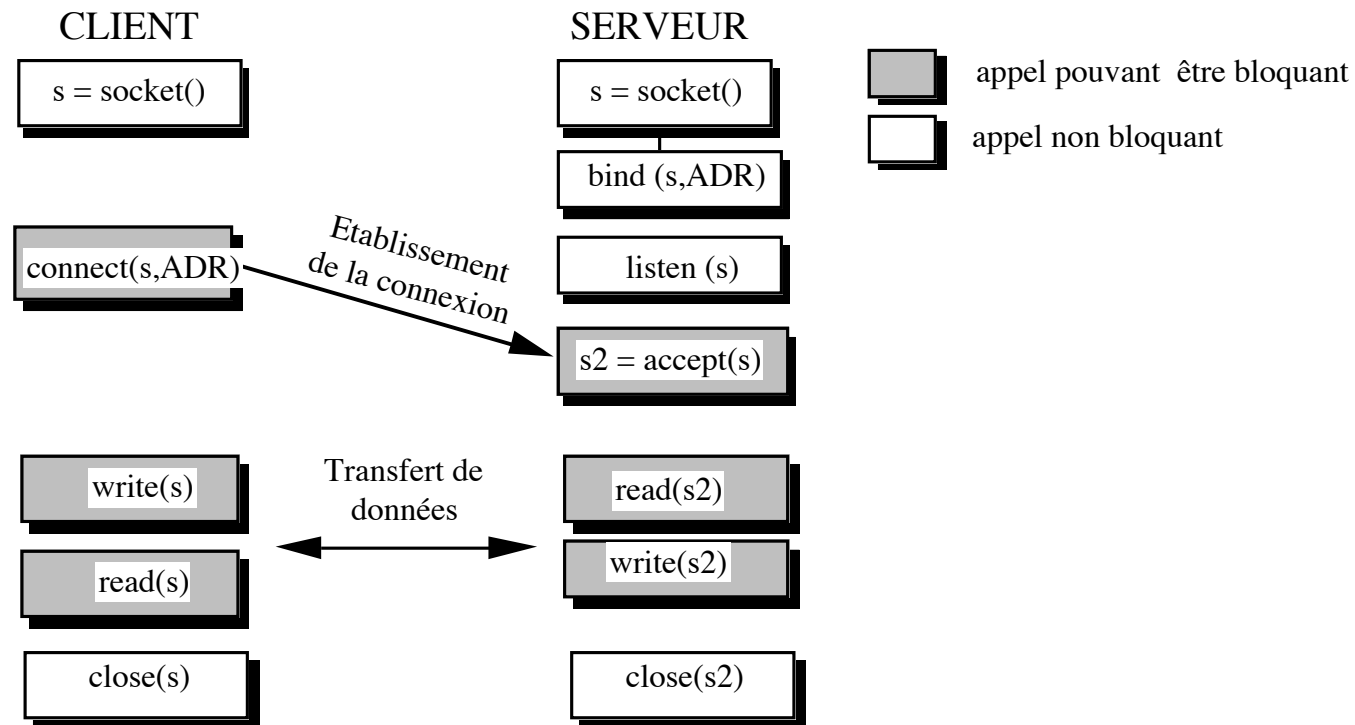


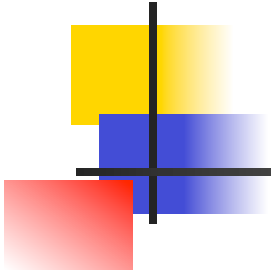
Création de sockets

- `Socket = socket(Domain, Type, Protocol)`
 - `Socket`: entrée dans la table des fichiers ouverts par le processus

- Pour attribuer un nom :
 - `bind(Socket, &ServerAddr, ServerAddrLen);`
 - `Socket` numéro renvoyé par la primitive `socket`
 - `ServerAddr` structure définissant le serveur
 - `ServerAddrLen` taille de cette structure (`sizeof`)

Schéma de communication en mode TCP





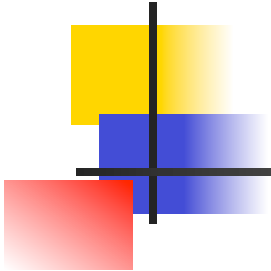
Communication en mode connecté

- Côté client

- Appel à la fonction :

`connect(Socket, &ServerAddr, ServerAddrLen)`

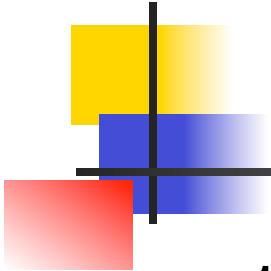
- Cette fonction lance une demande d'établissement de communication vers un serveur dont on a donné l'adresse (IP, port) dans la structure `ServerAddr`



Communication en mode connecté

■ Côté serveur

- `listen(ServerSocket, Nb_Clients)`
 - définition de la taille de la file d'attente sur un socket :
- `Socket=accept(ServerSocket, &ClientAddr, &ClientAddrLen)`
 - attente d'une demande de connexion (faite par un connect) :
 - **Socket numéro de socket renvoyé par accept** dès qu'il reçoit une demande de connexion
 - `ServerSocket` numéro renvoyé par socket
 - `ClientAddr` structure décrivant le client appelant

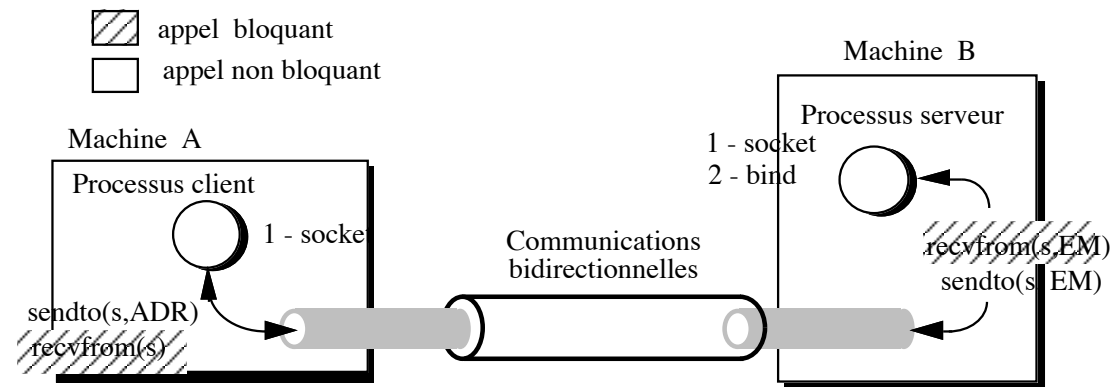


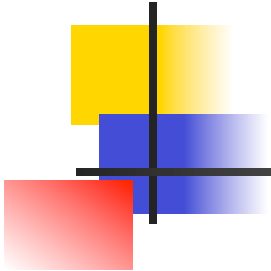
Emissions/Réceptions

- 1- Utilisation des fonctions standards destinées aux fichiers :
 - **read/write**(Socket, Message, MessageLen)
- 2- Utilisation des fonctions spécifiques (meilleure gestion des erreurs) :
 - **send/recv**(Sock, Message, MessageLen, Option)
 - Exemple d'options : MSG_OOB,
 - Une façon élégante de récupérer un caractère envoyé "OOB", est de traiter le signal SIG_URG envoyé par le noyau lors de la réception de ce caractère

Communications en mode datagramme (UDP)

- Client (appelant) :
 - Crée une socket ;
 - Lit ou écrit sur la socket ;
- Serveur (appelé) :
 - Crée une socket ;
 - « binding »
 - Lit ou écrit sur la socket .





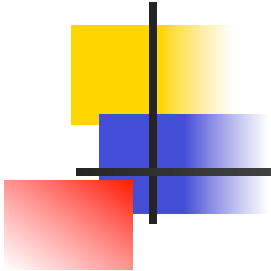
Communications en mode datagramme (UDP)

- Emission

- `sendto(Socket, Message, MessageLen, 0, &ReceiverAddr, ReceiverAddrLen)`

- Reception

- `recvfrom(Socket, Message, MessageLen, Flags, &SenderAddr, &SenderAddrLen)`



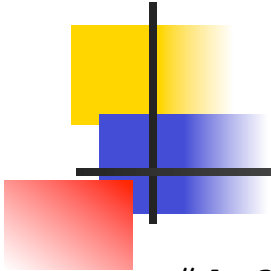
Canevas de client en mode datagramme (UDP)

```
char Message[] = "Coucou";

/* creation de la socket */
Socket = socket(AF_INET, SOCK_DGRAM, 0);

/* Construction de l'adresse du récepteur */
ReceiverAddr.sin_addr = gethostbyname(ReceiverName);
ReceiverAddr.sin_port = htons(ReceiverPort);

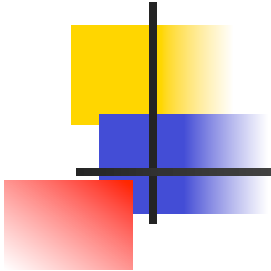
/* Emission du message */
sendto (Socket, Message, strlen(Message), 0,
        &ReceiverAddr, ReceiverAddrLen)
```

Canevas d'un serveur datagramme

```
#define ReceiverAddrPort 7777
struct sockaddr ReceiverAddr, SenderAddr;
int SenderAddrLen;
char Message[256]

/* Initialisation de la socket avec un numéro donné */
    Socket = socket(AF_INET, SOCK_DGRAM, 0);
    ...
    ReceiverAddr.sin_port = htons(ReceiverAddrPort);
    ...
    bind(Socket, &ReceiverAddr, ReceiverAddrLen)
/* Attente d'un message */
    recvfrom(Socket, Message, sizeof(Message), ...,
             &SenderAddr, &SenderAddrLen);
```

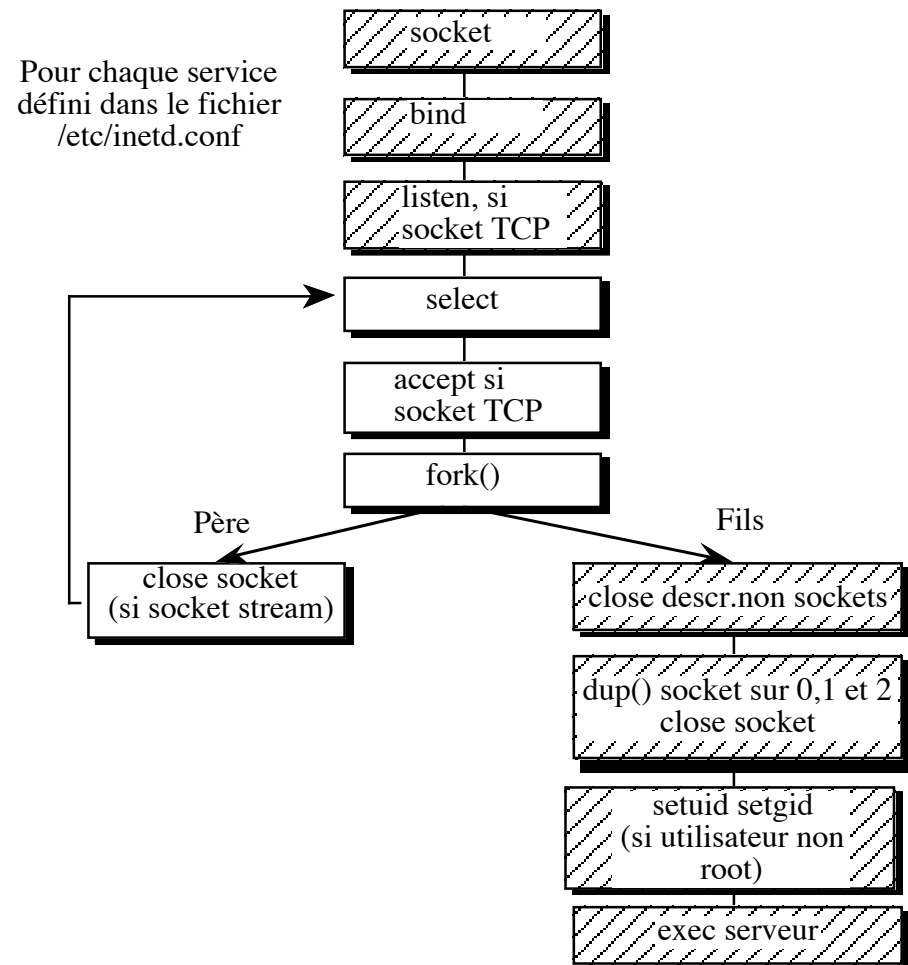


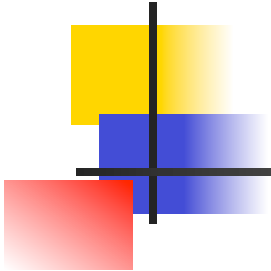
Multiplexage: select()

- Pour attendre sur plusieurs ports simultanément, on utilise la fonction `select`
- `MaskLen=select(MaskLenMax, R_Mask, W_Mask, E_Mask, Timeout)`
 - `MaskLen` nombre de ressources ayant répondu
 - `MaskLenMax` nombre de ressources sur lesquelles on attend
 - `R_Mask` les ressources sur lesquelles on attend une écriture
 - `W_Mask` les ressources sur lesquelles on attend une lecture
 - `E_Mask` les ressources sur lesquelles on attend un événement
 - `Timeout` temps d'attente avant réponse
- Remarques :
 - `select()` est une fonction Unix, non spécifique aux sockets, qui permet de multiplexer tous les fichiers fonctionnant en **producteur/consommateur**,
 - pour positionner les masques, utiliser :
 - `FD_ZERO` et `FD_SET`

Multiplexage : le serveur `inetd`

- `inetd` trouve la liste des serveurs qu'il doit lancer dans le fichier de configuration : `/etc/inetd.conf`





API : divers

- Informations sur les machines

- `gethostbyaddr(struct sockaddr *HostAddr, int HostAdddrLen, int Type);`
- `gethostbyname(char *HostName),`
- `gethostent()`

- Informations sur les services

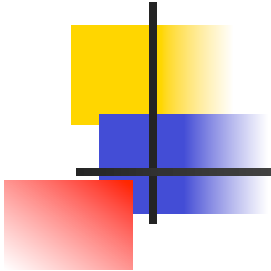
- `getservbyport(int Port, char *Proto)`
- `getservbyname(char *Nom, char *Proto)`
- `getservent()`

- Informations sur les ports

- `Getsockname(Socket, &sa, &len)`
- `getpeername(Socket, &sa, &len)`

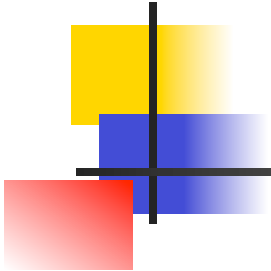
- Fin de communication et fermeture de socket

- `shutdown(Socket, Direction)`
- `close(Socket)`



API : normalisation des données

- Pas de protocole de normalisation des données
- Il existe des fonctions convertissant des informations sur 16 ou 32 du format réseau au format local et vice-versa :
 - `htonl(net_long host_long)`
 - `htons(net_short host_short)`
 - `ntohl(host_long net_long)`
 - `ntohs(host_short net_short)`
 - ...



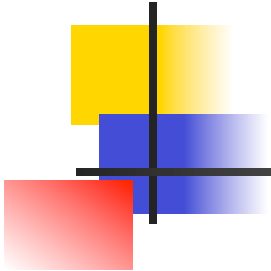
API : Traitements spéciaux

`setsockopt(...)`

- Exemple d' options : taille des buffers, réutilisation d' un port

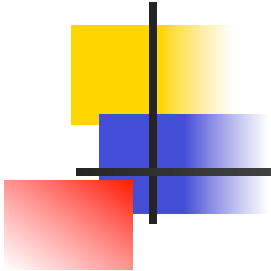
`readv()`, `writev()`

- Pour et de recevoir des données éparpillées sans avoir à les rassembler dans un buffer unique
- Mode « scatter/gather »



Bilan API sockets (1)

- Faiblesses :
 - Service d'annuaire qui associe nom de machine et adresse IP (`gethostbyname`), pas d'annuaire au niveau **service** (c'est-à-dire pour les numéros de ports, sauf ceux des *well know services* dans `/etc/services`)
 - Représentation des données
 - **Pas d'obligation de normalisation sur les données applicatives**
 - Jeu d'utilitaires pour convertir les information de niveau réseau (`htons`, `htonl`, etc), utilisable pour les données applicatives.



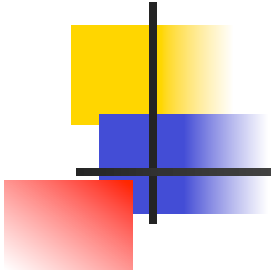
Bilan API sockets (2)

- API très puissante:
 - Multicast, mode RAW (setsockopt), etc

- ... mais syntaxe lourde :

- Exemple :

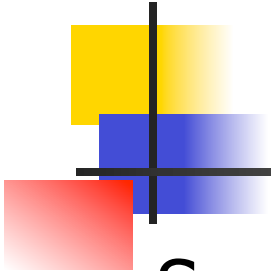
```
sockaddr_ecou.sin_family      = AF_INET;  
sockaddr_ecou.sin_addr.s_addr = INADDR_ANY;  
sockaddr_ecou.sin_port       = htons (PORT_SERV);  
bind(sock_ecou, (struct sockaddr *)&sockaddr_ecou,  
      sizeof(sockaddr_ecou));
```

API Java

- L' API Java reprend les concepts de l'API C standard
- Squelette du code Java pour le serveur (objets de type `ServerSocket` et `Socket`) :

```
// creation d'un port d'écoute et « bind »  
    ServerSocket  serverSocket=  new ServerSocket(Port);  
  
// Le serveur se bloque sur le port d'écoute Port.  
// Dès qu'il sort du accept, il récupère un port de  
// communication  
    Socket socket = serverSocket.accept();
```



Patron de conception : serveur Java multi-tâches

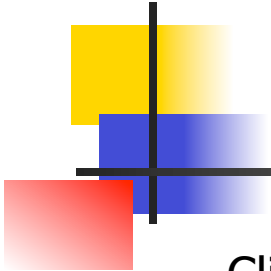
■ Serveur : multiplexage en utilisant des threads :

```
// creation d'un port d'écoute et attente sur ce port
ServerSocket serverSocket = new ServerSocket(Port);
Socket socket = serverSocket.accept();
```

```
// Un thread va gerer la comm. avec le client
new Thread_Dialogue(socket).start();
```

■ Thread qui dialogue avec un client :

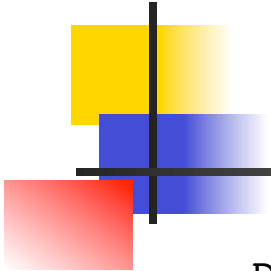
```
class Thread_Dialogue extends Thread {
    Socket socket;
    public Thread_Dialogue(Socket socket){
        this.socket = socket;
    }
    public void run(){
        ... = ... (socket.getOutputStream());
    } }
}
```



Canevas client Java TCP

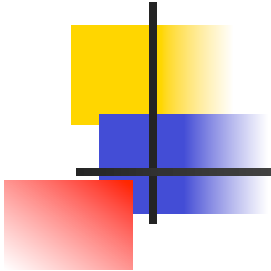
- Client qui communique avec le serveur TCP précédent :
 - Il envoie au serveur des chaînes de caractères puis attend ensuite une réponse qu'il affiche à l'écran

```
socket = new Socket("machine", Port);  
// Pour écrire sur la socket :  
PrintWriter Sortie_TCP = new PrintWriter(socket.getOutputStream());  
// Pour lire sur la socket :  
BufferedReader Entree_TCP = new  
    BufferedReader(newInputStreamReader(socket.getInputStream()));  
...  
while(...){  
    Sortie_TCP.println (...);  
    ...  
    System.out.println(Entree_TCP.readLine());  
}
```



Canevas pour un serveur Java UDP

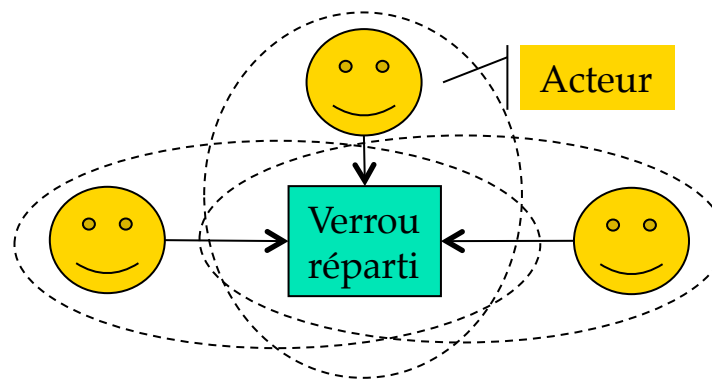
```
DatagramSocket      socket;  
DatagramPacket      message = null;  
byte[]              buffer;  
InetAddress          address;  
int                  port = 0;  
  
// Initialisations  
socket              = new DatagramSocket(PORT_SERV_UDP);  
buffer              = new byte[256];  
message             = new DatagramPacket(buffer, buffer.length);  
// Attendre un message  
socket.receive(message);  
// Envoyer la reponse vers le client  
address             = message.getAddress();  
port                = message.getPort();  
Message = new DatagramPacket(buffer, buffer.length, address, port);  
socket.send(message);
```



Canevas pour un client Java UDP

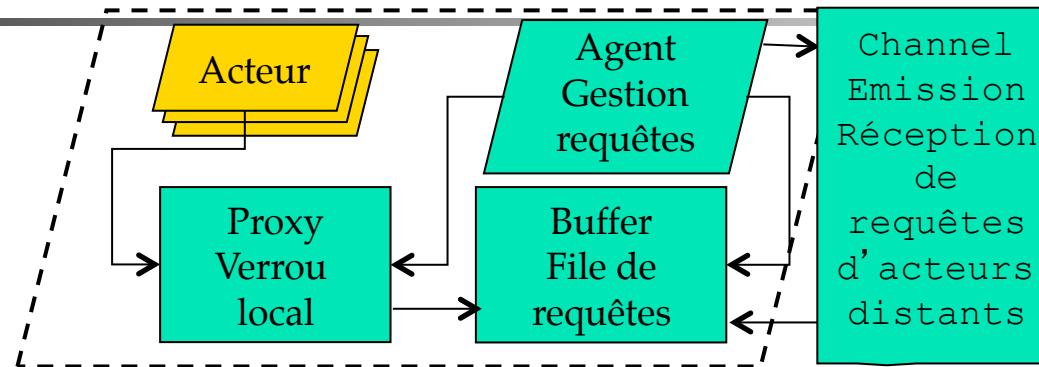
```
DatagramSocket socket;  
DatagramPacket message;  
byte[]          buffer;  
InetAddress     address;  
  
// Initialisations  
buffer          = new byte[256];  
socket          = new DatagramSocket();  
address         = InetAddress.getByName(...);  
// Emission  
packet = new DatagramPacket(buffer, buffer.length, address, port);  
socket.send(packet);  
// Reception  
packet = new DatagramPacket(buffer, buffer.length);  
socket.receive(packet);  
String message = new String(packet.getData());
```

Retour sur le cas d'étude

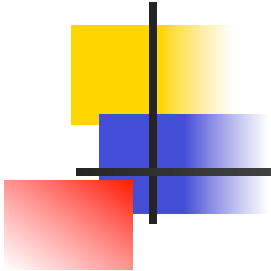


Etude de cas (Java sockets)

Modélisation des Acteurs

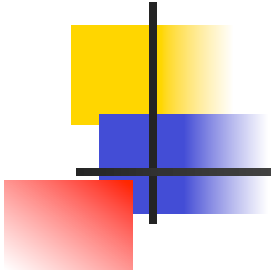


- Chaque acteur est modélisé par un processus léger et se trouve intégré avec le proxy et l'agent dans un même processus lourd,
- Chaque agent A_n modélisé par un processus léger, dispose d'un tampon circulaire T_n de requêtes,
- Channel reçoit à l'aide de sockets les requêtes émises vers le nœud n et les dépose dans le tampon T_n
- L'agent A_n , un processus cyclique qui lit les requêtes de T_n :
 - interagit avec le proxy P_n ,
 - et peut envoyer des requêtes au travers de Channel à l'aide de sockets.



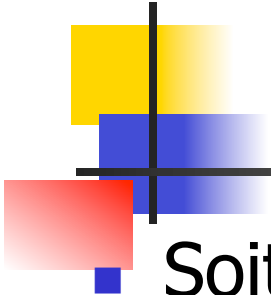
Etude de cas (Java sockets)

- Grâce à l'architecture adoptée, nous pouvons **réutiliser** les codes du Proxy et de l'Agent,
- Channel ouvre sur un site une socket vers chacun des autres sites (et éventuellement vers soi-même),
- Un thread gère de manière cyclique une socket en recevant les requêtes pour les déposer dans le tampon du site (rappel : la réception est une opération bloquante).



Etude de cas (Java sockets)

- Chaque site contient n threads supplémentaires dans `Channel` pour établir un maillage complet des sites,
- Cependant, `accept()` et `connect()` étant deux opérations asymétriques, il faut établir un protocole afin d'éviter les interblocages (chaque site attendant sur `accept` par exemple).

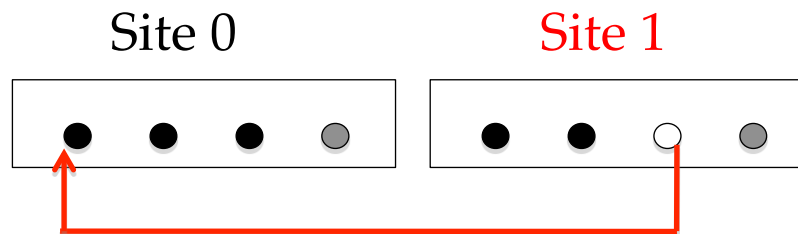


Cas d'étude (Java sockets)

Protocole de maillage

- Soit des sites numérotés de $0 \dots m$, chaque site n :
 - Crée $n + 1$ threads qui se connectent aux sites **0 .. n** de la liste des sites existants (notamment avec lui-même),
 - « Accepte » les connexions des sites de **n .. m** (notamment avec lui-même) et crée autant de threads pour gérer ces sockets.

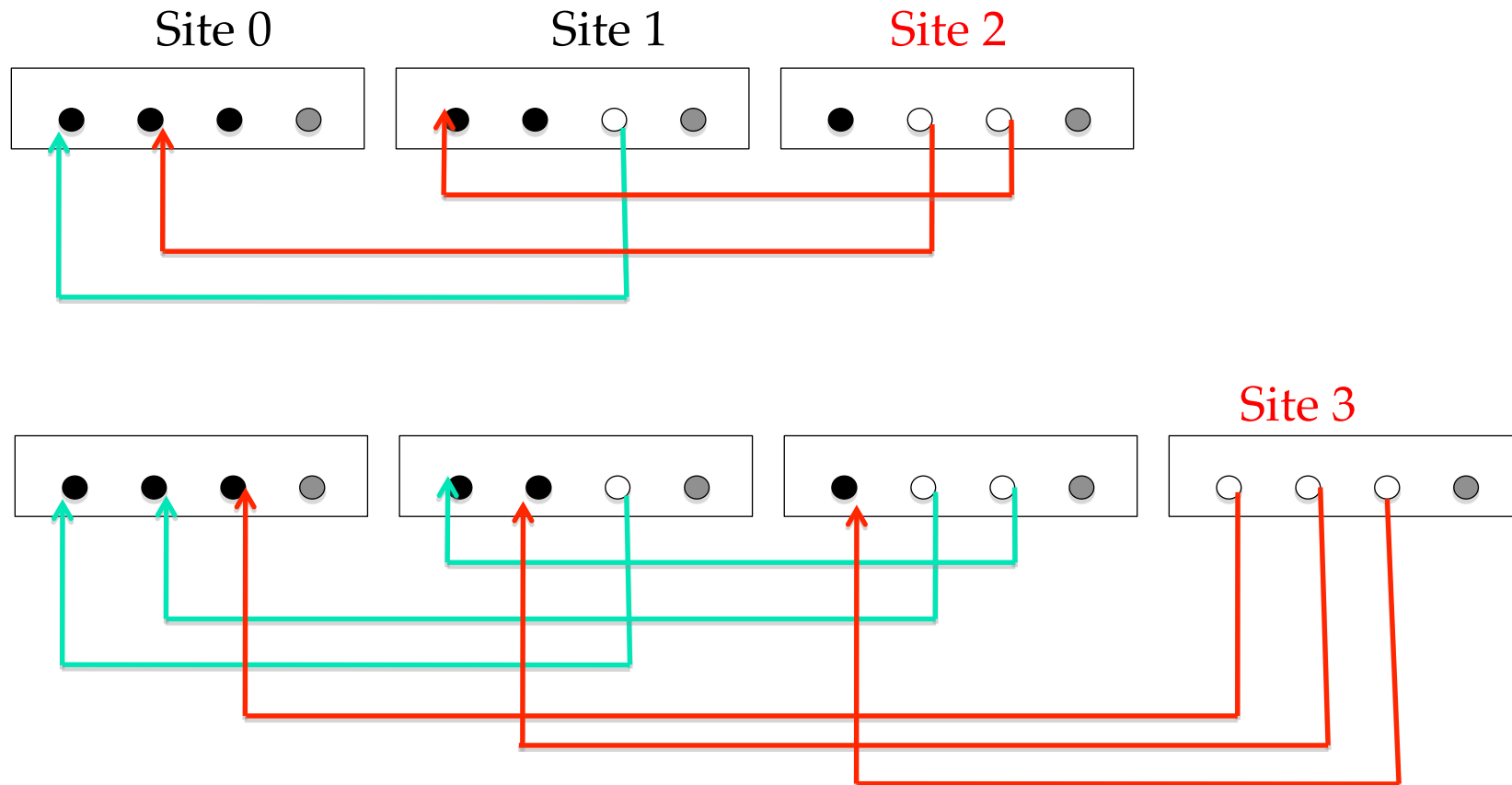
Maillage : 2 sites sur 4

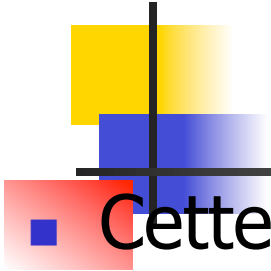


Légende :

- Thread bloqué sur un « accept », en attente d'un connect fait depuis un autre site.
- Thread faisant « connect » vers un autre site.
- Nouvelles connexions sur ajout du site

Maillage : 3, puis 4 sites sur 4

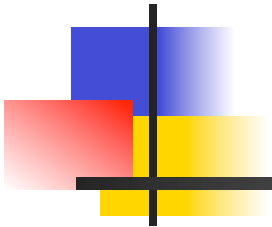




Cas d'étude (Java sockets)

Phase de maillage

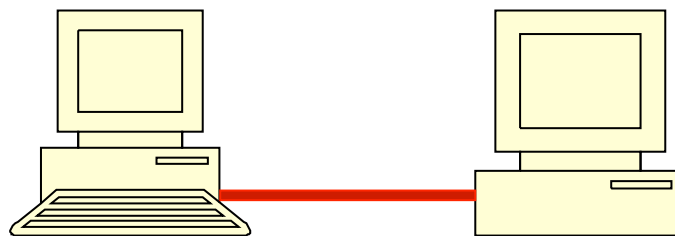
- Cette phase constitue la phase d'initialisation mais les threads gérant les sockets ne peuvent pas encore délivrer les requêtes dans le tampon de l'agent qui n'a pas été encore créé
- Après l'opération `Channel.initialize`, le sous-programme principal `Main` peut créer `Actor`, `Proxy` et `Agent`
- La dernière opération `Channel.activate` débloque les threads gérant les sockets et donc autorise l'accès au tampon



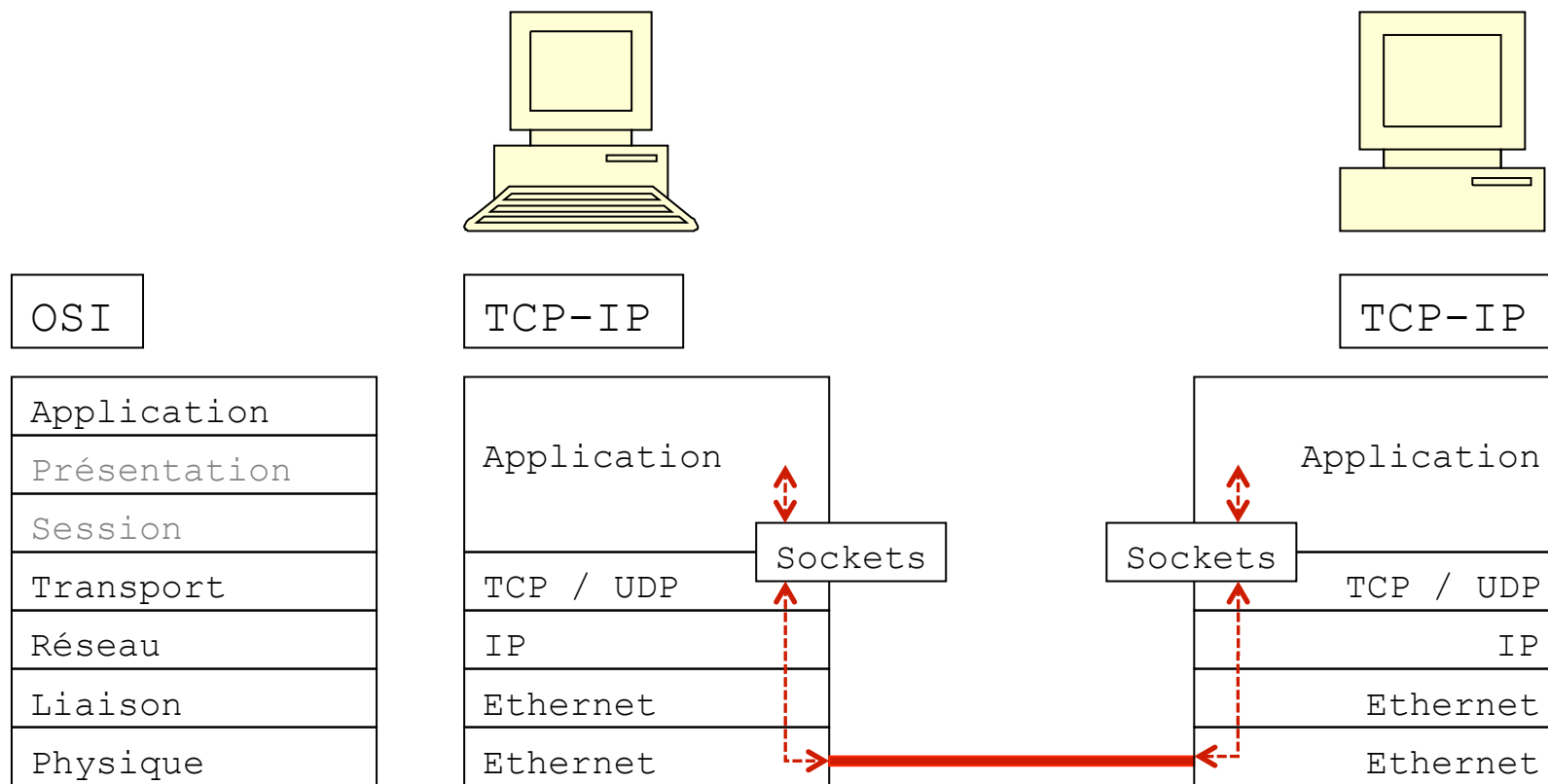
Intergiciels, concepts de base

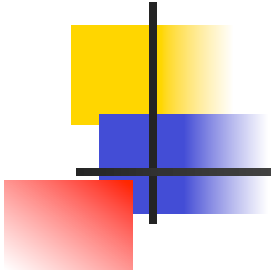
Rappel : système réparti

- Système constitué de multiples ressources informatiques (ex: processus logiciels, équipements matériels, ...) interconnectées via un support de communication (ex: réseaux, communication interprocessus, mémoire partagée, ...)



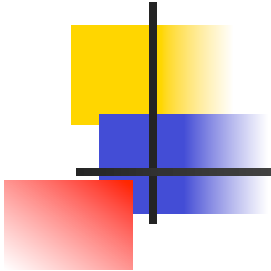
Rappel : modèle OSI et TCP/IP





Des sockets aux intergiciels

- Limites de l'API socket :
 - Travail fastidieux, répétitif, sujet aux erreurs : configuration, construction de requêtes, déploiement, exécution
- Solution :
 - Factoriser les opérations sur les sockets à l'aide de bibliothèques de plus haut niveau
- Intergiciel (« middleware ») : abstractions pour la répartition
 - Fournit un modèle de répartition : entités inter-agissantes suivant une sémantique claire
 - Au dessus des briques de base de l'OS (dont les sockets)
 - Définit un cadre complet pour la conception et la construction d'applications distribuées

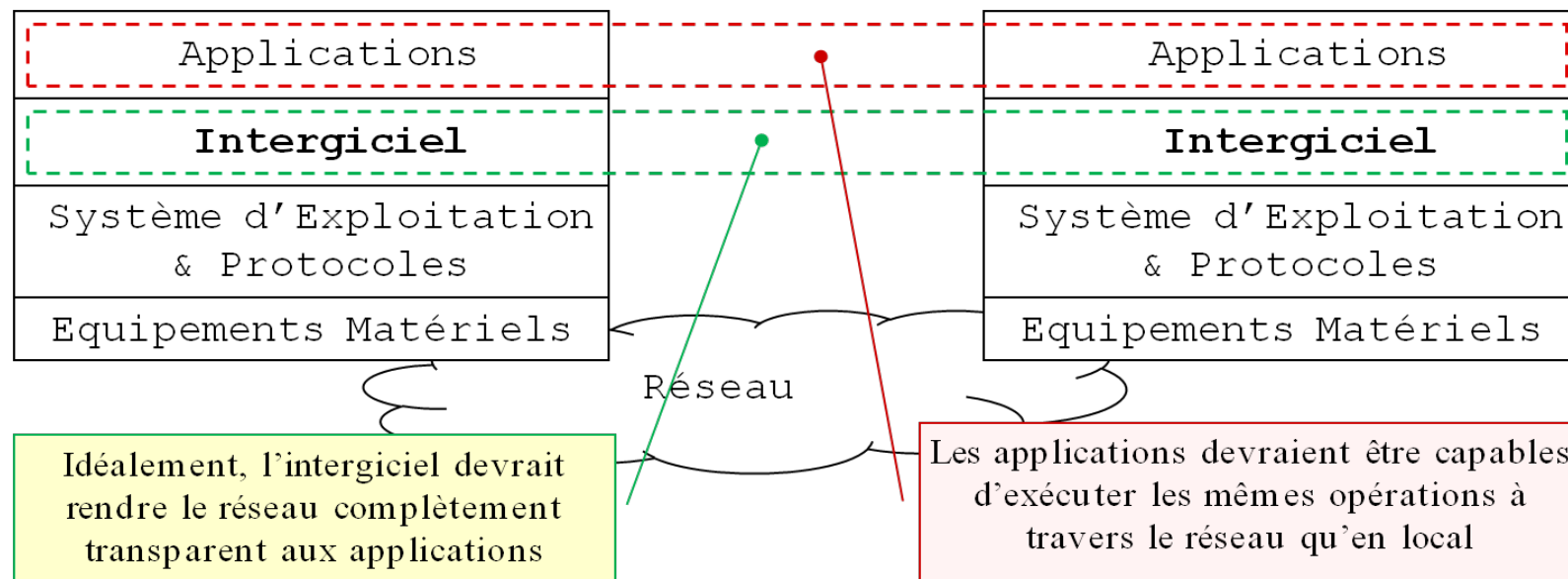


Intergiciels - définitions de base

- Logiciel réutilisable qui fait la médiation entre les applications et le réseau
- Gère et facilite l'implantation de l'interaction entre des applications qui s'exécutent sur des plateformes distantes et/ou hétérogènes

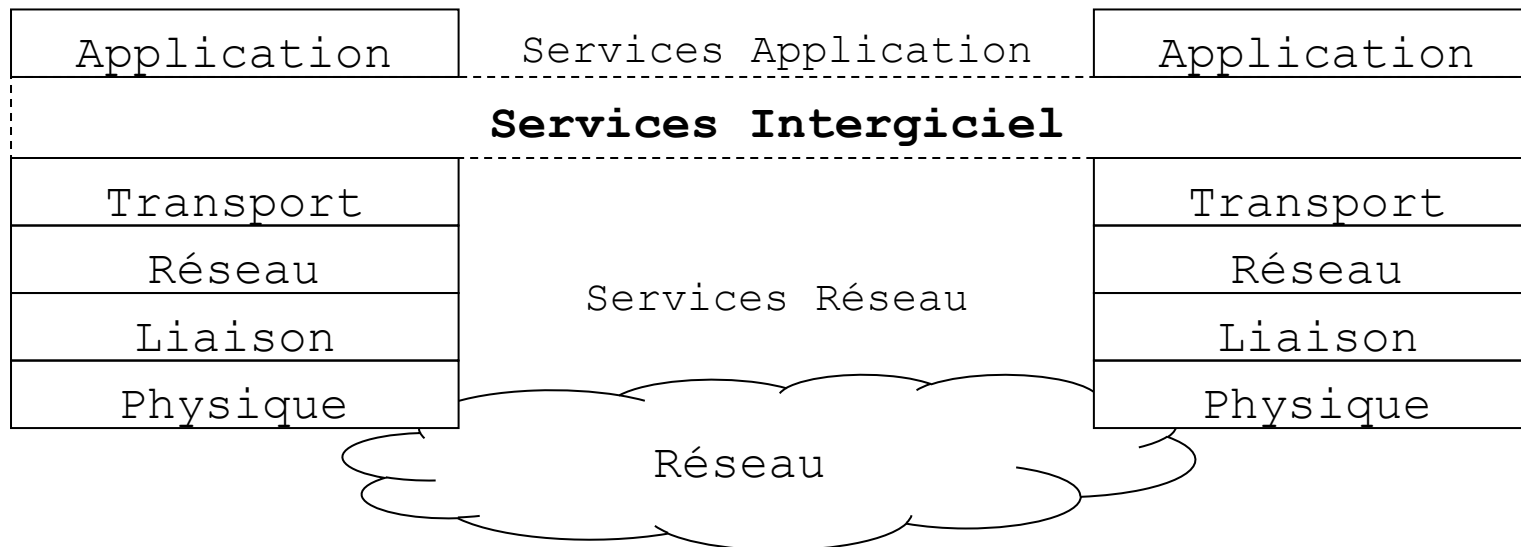
Intergiciel – positionnement (1)

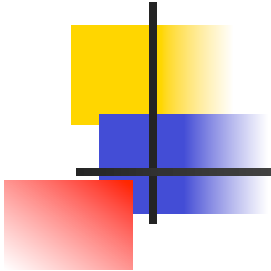
- Au dessus du réseau
- En dessous des applications (logique de business), bases de données, ...
- Sur toutes les machines participantes



Intergiciel – positionnement (2)

- L'intergiciel - où se trouve-t-il par rapport au modèle OSI?

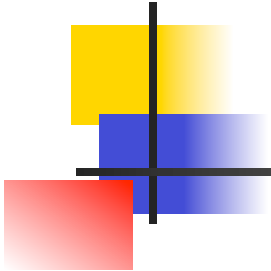




Intergiciel - contenu

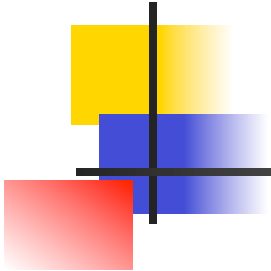
- Une collection réutilisable et extensible de services non-fonctionnels
 - A l'origine : des services de communication
 - Et ensuite : nommage, transactions, sécurité, persistance, gestion d'instances, journalisation (logging), ...

Un service *non-fonctionnel* n'est pas conscient des buts fonctionnels (de business) et de la sémantique de l'application qu'ils servent



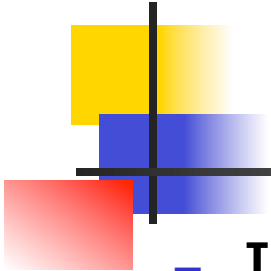
Intergiciels – importance

- Aide à définir et concevoir un système réparti
 - Aide les développeurs, en les protégeant :
 - des détails liées à la plateforme sous-jacente
 - des difficultés liées à la distribution
 - => Plus besoin de programmer de sockets
- Amortit les coûts de développement initiaux par la réutilisation et par la dissémination de l'expertise
 - Communication : création de session, (de)*marshalling*, collecte et « bufferisation » de requêtes, contrôle de la concurrence, ...
 - D'autres services non-fonctionnels : nommage, transactions, sécurité, ...



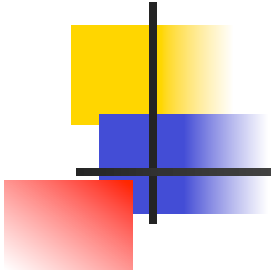
Intergiciels – caractéristiques

- Se fonde sur des mécanismes de répartition
 - Passage de messages, sous-programmes distant
 - Objets répartis, objets partagés (mémoire répartie)
 - Transactions
- Fournit un contrôle sur ces mécanismes
 - Langages de description ou de programmation
 - Interfaces de bibliothèques ou de services
- S'accompagne d'un intergiciel de mise en œuvre
 - Bibliothèques + outils



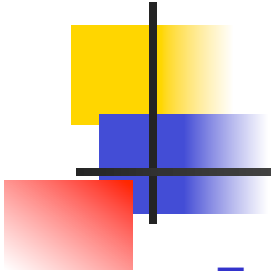
Modèle de répartition – envoi de messages

- Interface réseau
 - Interface bas niveau - UDP, TCP, ATM
 - IPv6, multicast, SSL/TSL
- PVM/MPI
 - Calcul massivement parallèle
 - Communication de groupe
- Java Messaging Service (JMS)
 - Interface haut niveau (Standard Java)
 - Message Oriented Middleware (MOM)
 - Point à point
 - Publish / Subscribe



Modèle de répartition - appel de procédure à distance

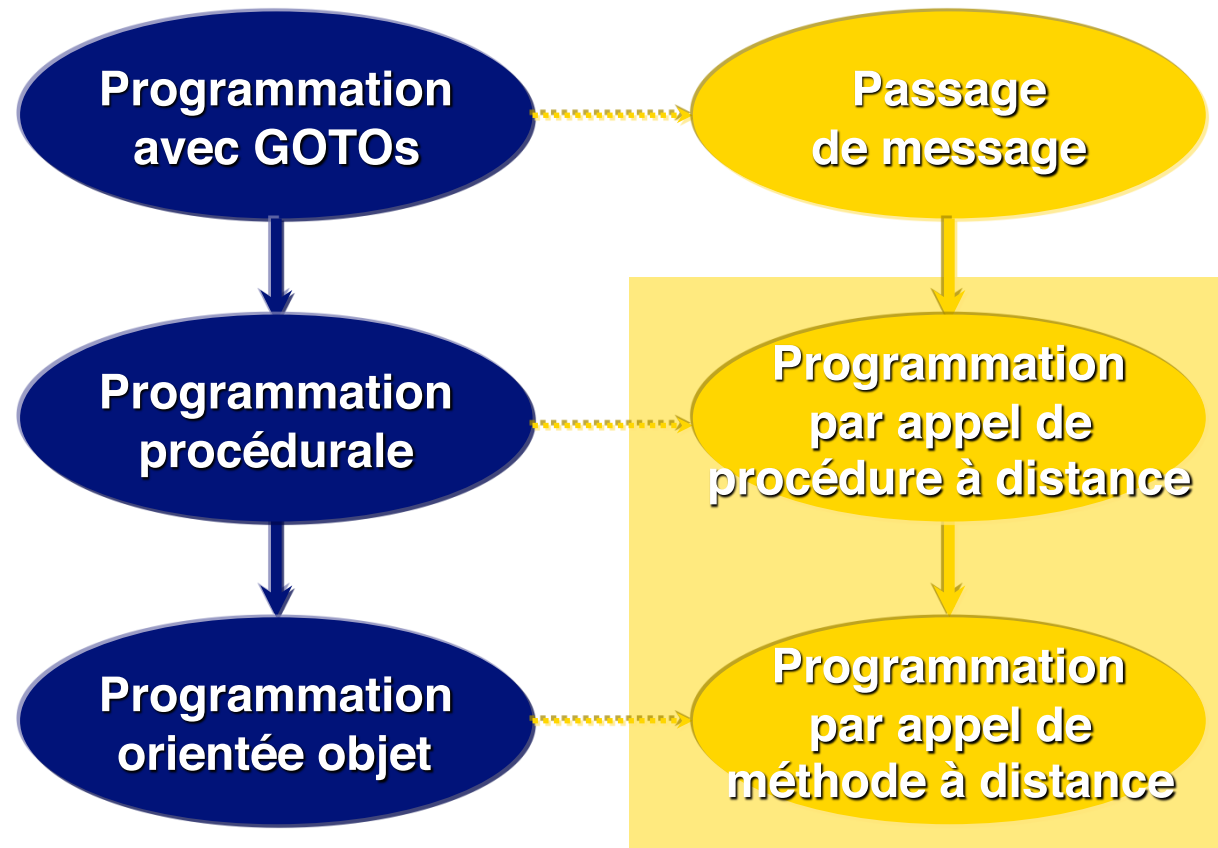
- **RPC (Remote Procedure Call) - Sun**
 - Service de nommage ou de localisation (portmapper)
 - Compilateur rpcgen qui produit les souches et squelettes
 - Intégré dans DCE (Distributed Computing Environment)
- **DSA (annexe des systèmes répartis d'Ada95)**
 - Réduit la frontière entre réparti et monolithique (local et uni-thread)
 - Services de nommage et de localisation internes
 - Annulation d'appel de sous-programmes distants
 - Référence sur des sous-programmes distants
- **SOAP (Simple Object Access Protocol)**
 - Protocole léger de communication d'informations structurées
 - Le plus souvent : RPC à base de HTTP + XML (protocole uniquement)

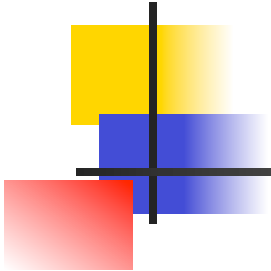


Appel de méthode à distance - exemples -

- Approche dépendante du langage de programmation
 - Modula-3 : extension au langage Modula-2
 - RMI (remote method invocation) : extension de Java
 - DSA (annexe systèmes répartis d'Ada95) : extension au langage Ada95
- Approche indépendante du langage de programmation
 - CORBA (OMG IDL - langage de définition d'interface)
 - DCOM, .NET (Microsoft IDL)

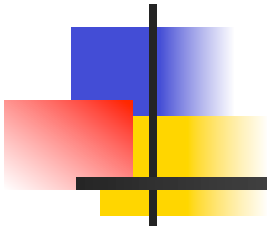
Motivations – Distribution (Analogie)





Motivations - Hétérogénéité

- Appel de méthodes à distance sur objets répartis hétérogènes, indépendamment des :
 - langages de programmation
 - systèmes d'exploitation
 - plates-formes d'exécution
 - fournisseurs de technologie
 - protocoles réseau
 - formats des données
- Consensus pour l'interopérabilité
 - Mais, de nombreux intergiciels restent spécifiques à un langage, un vendeur, un OS

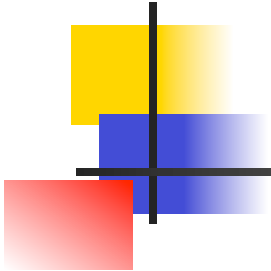


Java RMI

- Remote Method Invocation -

Ressources :

- <http://docs.oracle.com/javase/tutorial/rmi/index.html>
- <http://docs.oracle.com/javase/6/docs/technotes/guides/rmi>
- ...



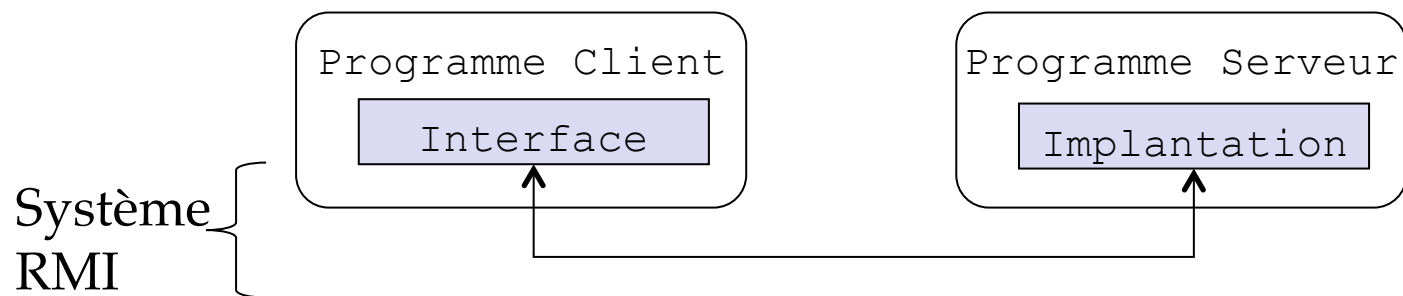
Objectifs de Java RMI

- Définir une architecture distribuée qui s'intègre dans le langage Java de façon transparente, donc qui permette :
 - L'**invocation de méthodes** sur des objets situés sur des machines virtuelles différentes, de façon similaire à une invocation locale,
 - L'utilisation de l'**environnement de sécurité Java** classique lors de l'exécution d'applications distribuées,
 - L'affranchissement du programmeur de la gestion de la mémoire, en définissant une extension distribuée au **garbage collector**.

Architecture RMI

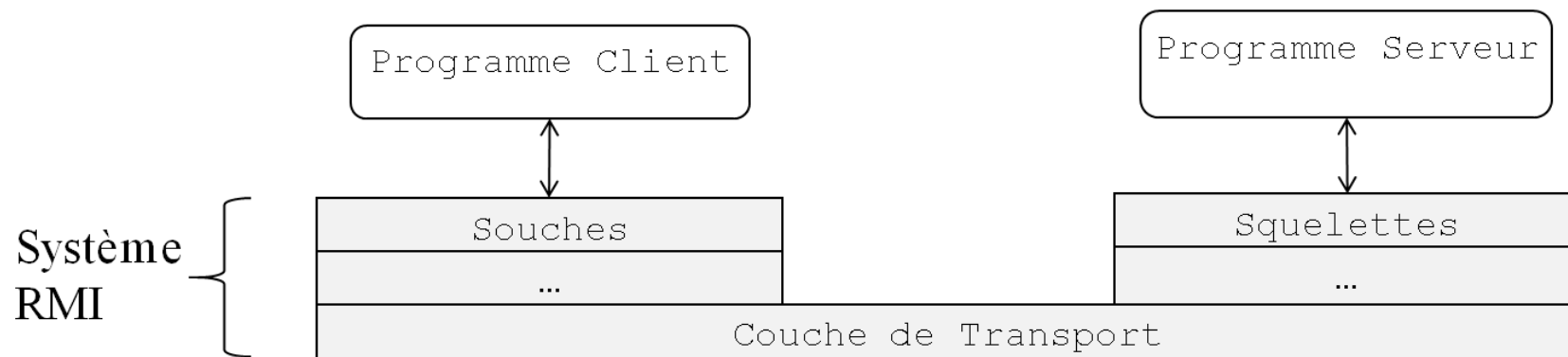
Interface - principe important

- Séparation de deux concepts :
 - *Définition* d'un comportement – Java Interface
 - *Implantation* d'un comportement – Java Class
- Dans un système réparti :
 - Le client s'intéresse à la description d'un service – Interface
 - Le serveur fournit l'implantation - Class



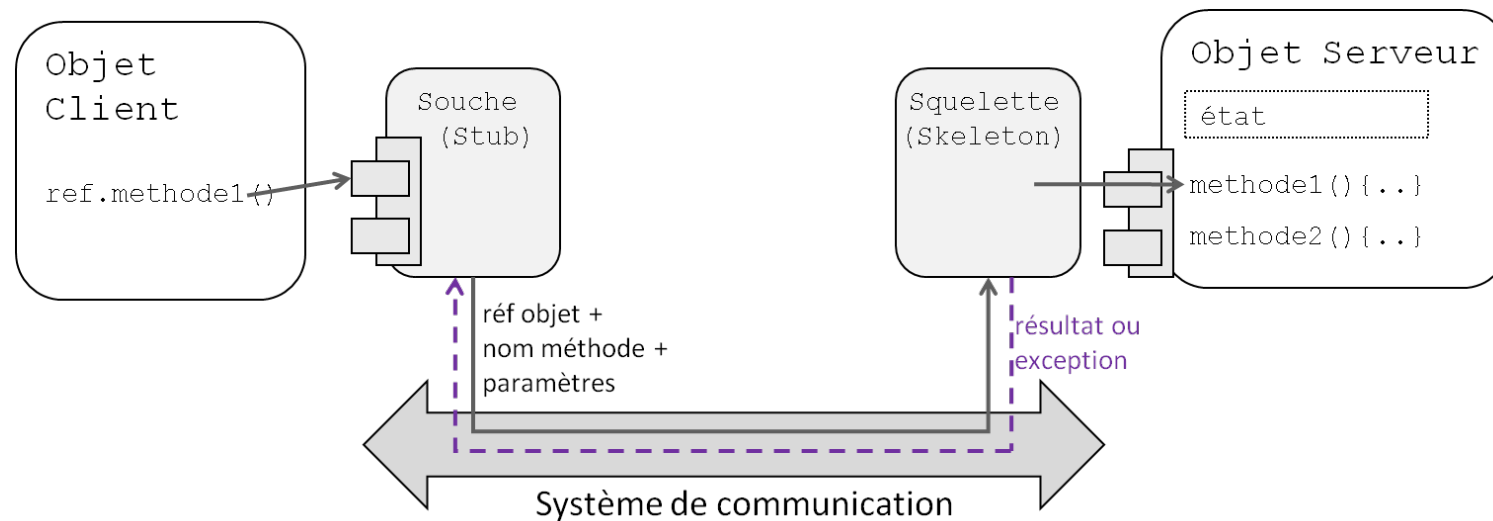
Architecture RMI – plusieurs couches

- Souches (stubs) et Squelettes (skeletons)
 - Juste en dessous du programme développeur
 - Transforment les appels Java en messages réseau
- ...
- Couche de Transport
 - Fait la connexion entre les JVMs
 - Utilise le protocole TCP/IP

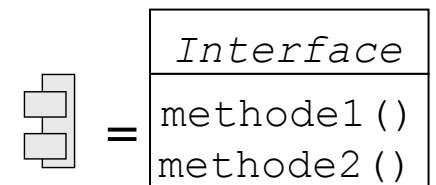


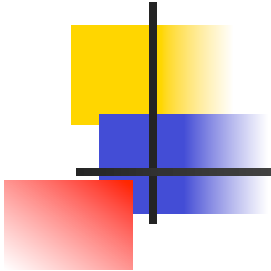
Architecture RMI

- Stub (Souche)
 - Fait la transition entre l'appel du client et le réseau
 - Skeleton (Squelette)
 - Fait la transition entre le réseau et l'appel vers l'Objet Serveur
- > *Visibles ou invisibles par le développeur, selon la version de JDK*



A. Diaconescu, L. Pautet & B. Dupouy

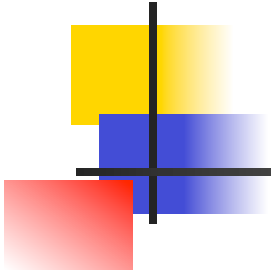




Objet réparti – RMI

- stubs et skeletons -

- Extension du mécanisme d'appels de méthodes à la répartition :
 - Objet « proxy » qui reprend la même signature que l'Objet Serveur : souche (« stub »), manipulé par le client,
 - Code côté serveur pour traiter la requête, et transmettre la réponse à l'Objet Client (« skel »),
 - RMI + JVM ajoutent la logique de traitement de la requête: localisation de l'objet, construction de la requête, transmission, ...



Objet réparti – RMI

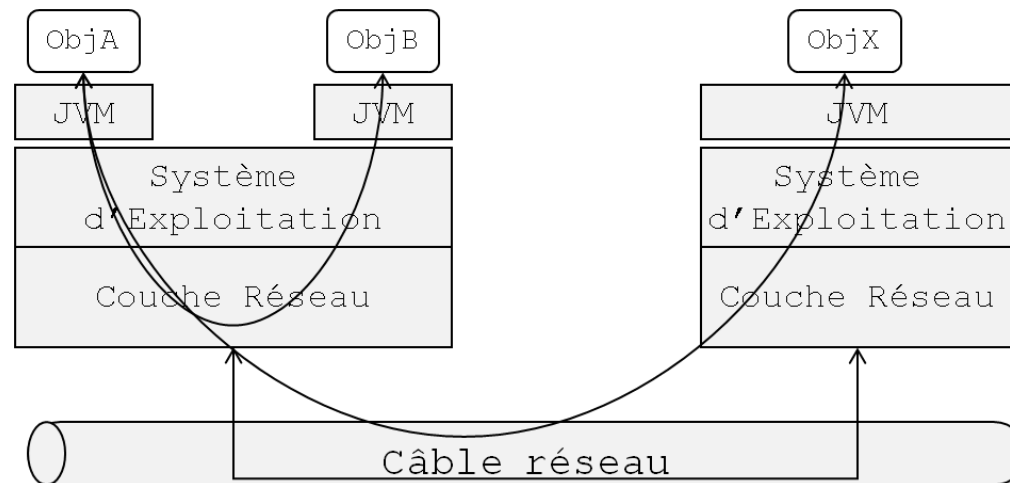
- terminologie et définitions -

- Un objet distant (**Remote Object**) est un objet dont les méthodes peuvent être invoquées par des clients situés sur des JVM différentes (logiquement/physiquement),
- Cet objet distant est manipulé au travers d'interfaces, appelées **Remote Interfaces** qui listent les méthodes que les clients pourront invoquer sur ces objets distants,
- Une référence distante (**Remote Reference**) est une référence qui permet d'adresser un objet situé sur une JVM différente,
- Une invocation de méthode distante (**Remote Method Invocation**) appelle une méthode définie dans une interface d'objet distant. Elle a la même syntaxe qu'une invocation de méthode locale.

Passage d'appels distants

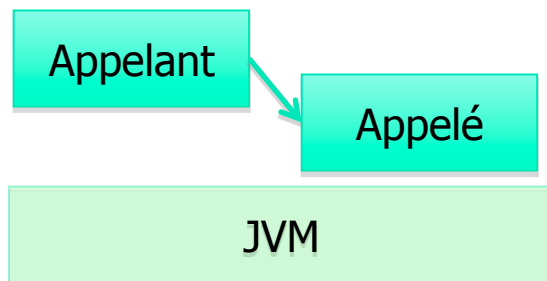
Objets s'exécutant sur :

- La même JVM - appel local
- Des JVMs différentes sur la même machine – appel distant
 - Passant par la couche réseau de la machine
- Des machines différentes – appel distant
 - Passant par une connexion réseau entre les machines

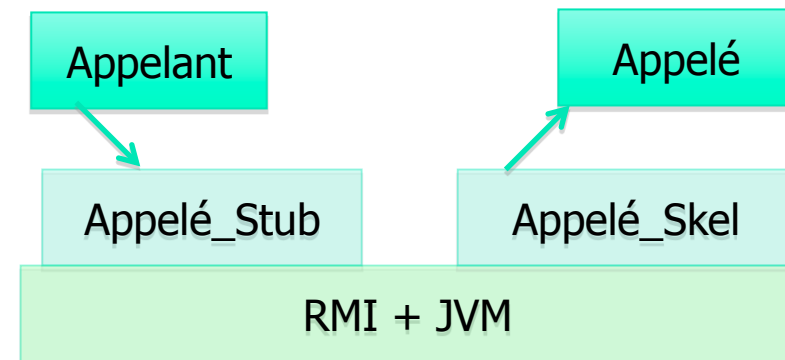


Objet réparti - RMI

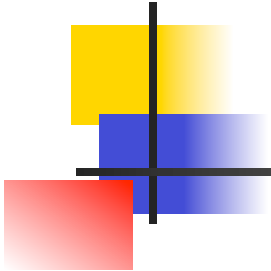
- Récapitulatif :



Appel local
(même JVM)

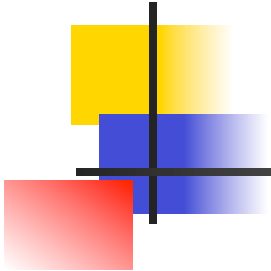


Appel distant
(JVMs différentes)



Différence appel local/distant

- Les clients manipulent des proxies qui implémentent les `remote interfaces` (pas directement les objets qui implémentent ces interfaces),
- Les paramètres de type simple (`int`, ...) sont passés par copie,
- Les objets locaux sont sérialisés (`serializable`) et passés par copie pour être envoyés : différence avec le modèle de Java, dans lequel les objets sont passés par référence,
- Les objets distants sont passés par référence : les méthodes sont bien exécutées sur l'objet lui-même, quelle que soit la machine virtuelle dans laquelle il réside,
- Les méthodes distantes provoquent des exceptions supplémentaires : les clients doivent traiter des exceptions liées au caractère distribué des méthodes qu'ils invoquent,
- Le bytecode n'est transmis à la JVM qu'au moment de son utilisation.



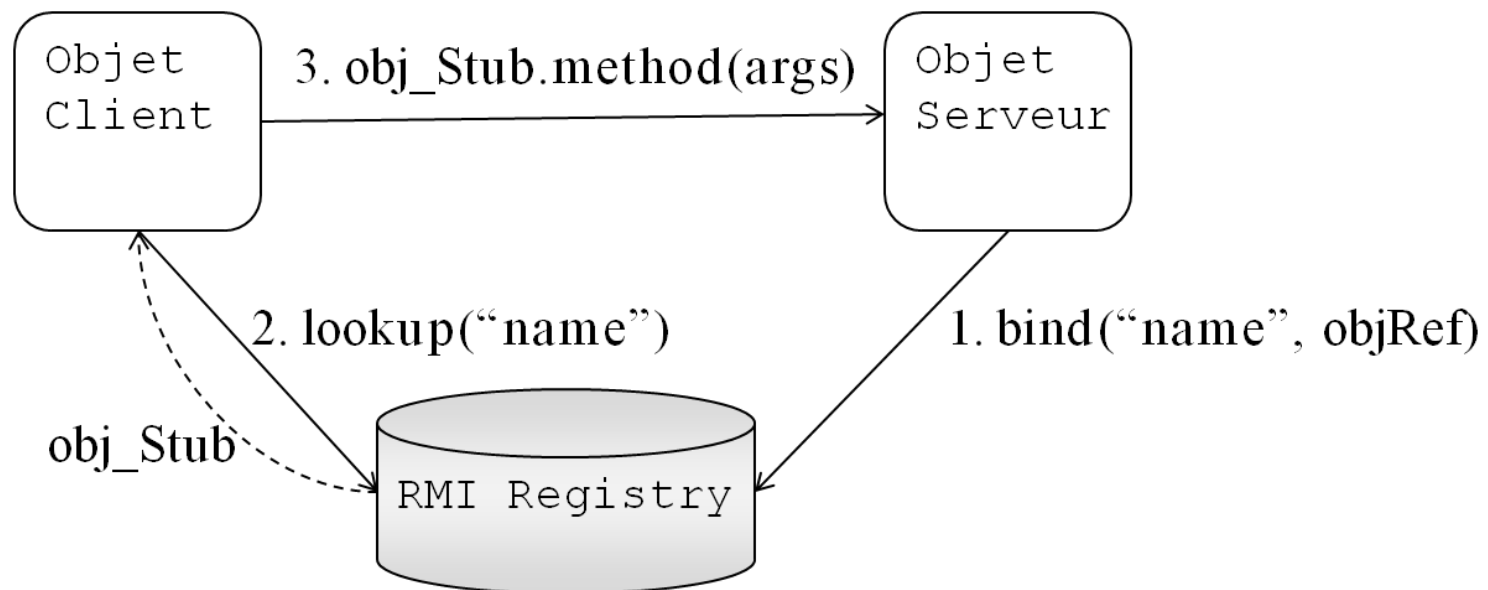
Comment trouver les objets?

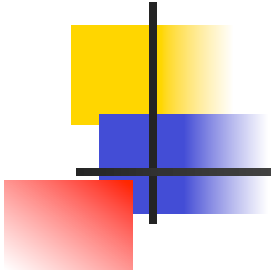
RMI Registry

- Les clients utilisent un service de nommage – RMI Registry (`rmiregistry`), afin de trouver les références des objets distants
- RMI Registry se situe à une adresse bien connue (machine et port)
- Les objets distants doivent être enregistrés avec RMI Registry afin de pouvoir être trouvés et appelés

Appel distant

- Etapes de connexion entre un client et un serveur





Fonctions RMI

RMI assure les fonctions de localisation (1), communication (2) et de chargement du bytecode (3)

- 1- Localisation des objets distants

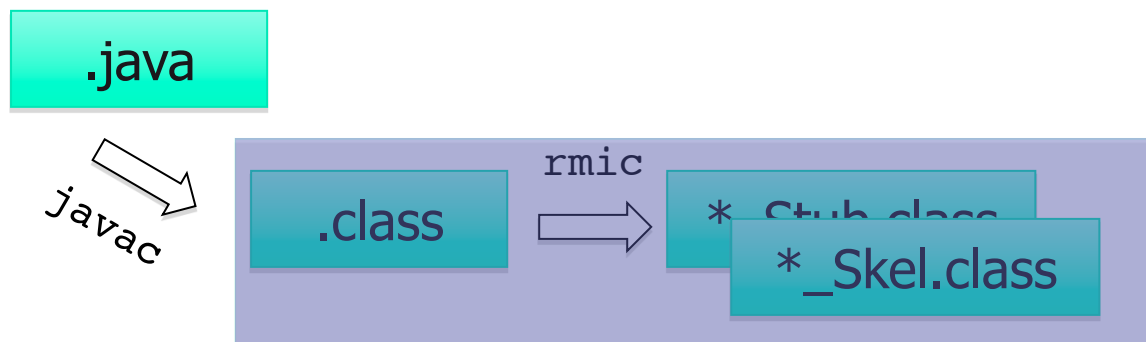
- Le serveur enregistre les objets qu'il exporte auprès de rmiregistry en utilisant `bind()`
- Le client trouve des références sur ces objets exportées et récupère les stubs correspondants en donnant le nom des objets correspondants lors de l'appel à `lookup()`

- 2- Communication transparente grâce au protocole RMI

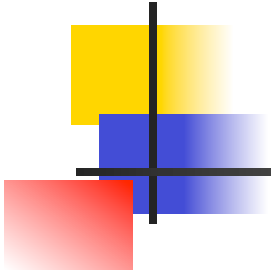
- 3- Téléchargement transparent de bytecode grâce au protocole RMI à partir d'un URL donné (ex : utilisation de serveurs Web, Ftp, ...)

Compilation Java vs Java + RMI

- Le cas réparti nécessite la génération d'éléments de code supplémentaires : codage/décodage des requêtes, gestion des tables de nommages, correspondance requêtes/code utilisateur
- La chaîne de compilation est étendue pour générer le code supplémentaire, qui va scruter le bytecode Java, et rechercher les éléments utiles (implantant les interfaces remote, serializable, ...)

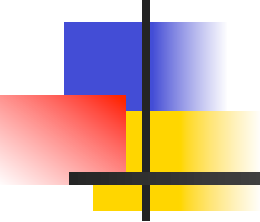


Application
prête à être déployée



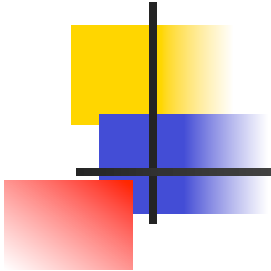
rmic

- Une fois qu'on a écrit une classe qui implémente les interfaces distantes, on crée le stub et le squelette correspondants grâce à **rmic** :
 - Ces deux classes sont rangées dans des fichiers dont les noms sont créés par ajout de `_Stub` au nom de la classe pour la souche (stub), et `_Skel` pour le squelette (skeleton)
- Notes
 - L'option `-keepgenerated` conserve les sources des stub et squelette
 - Object contient une méthode `getClass()` qui donne la classe de l'objet : en ajoutant les suffixes adéquats, on obtient les noms du stub et du squelette



Java RMI

1^{er} exemple



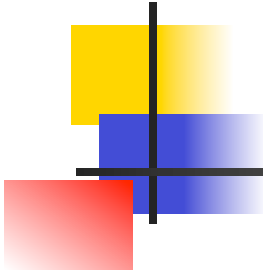
Exemple de base

- Soient les fichiers : `Hello.java`, `HelloServeur.java` du côté serveur, et `HelloClient.java` du côté client.
- Contenu de `Hello.java` : l'interface (ce que « voit » le client):

```
public interface Hello extends java.rmi.Remote {  
    String lireMessage() throws java.rmi.RemoteException;  
}
```

- `Remote` signifie que les méthodes de cet objet `Hello` doivent être appelables depuis une JVM autre que la JVM locale.
- `HelloServeur.java` implémente cette interface :

```
public class HelloServeur extends UnicastRemoteObject  
    implements Hello {...}
```



Exemple : canevas du serveur

//e.g. dans une méthode main :

```
// Creation de l'objet qui va etre invoque par les clients
HelloServeur monServeur = new HelloServeur();
```

```
// On enregistre le service auprès de rmiregistry
//(args[0] donne le port sur lequel écoute le rmiregistry)
```

```
myHostname machine = new myHostname();
String nomService = "//" + machine.QualifiedHost()
                    + ":" + args[0] + "/HelloServeur";
Naming.rebind(nomService, monServeur);
```

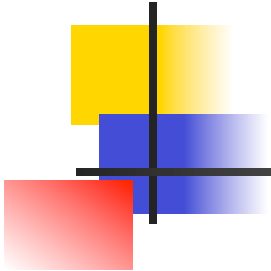
L'adresse IP de
la machine du RMI
Registry

Le nom de l'objet
serveur

Le port sur
lequel écoute le
RMI Registry

■ Après la compilation du serveur:

- pour avoir le stub (`HelloServeur_Stub.class`) destiné aux clients, et
- le squelette (`HelloServeur_Skel.class`),
- il faut faire : `rmic HelloServeur`



Exemple de base: canevas du client

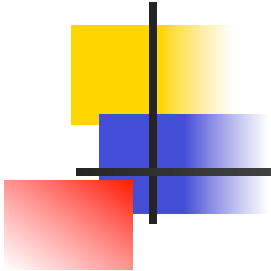
- Voici comment invoquer une méthode sur l'objet distant :

```
// Donner le nom du service demandé
String nomService = "//" + machineRmiRegistry + ":"
                  + portRmiRegistry + "/HelloServeur";
Hello obj = (Hello)Naming.lookup(nomService);

// On peut maintenant invoquer la méthode de l'objet
// qui est hébergé par le serveur
message = obj.lireMessage();
```

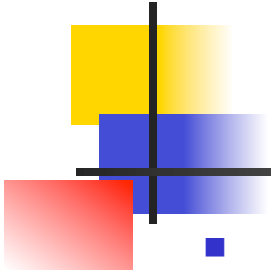
- On rappelle l'interface :

```
public interface Hello extends java.rmi.Remote{
    String lireMessage() throws java.rmi.RemoteException;
}
```



Exemple : exécution

- En étant toujours dans le même répertoire :
 - Lancer `rmiregistry` en arrière plan,
 - Lancer le serveur sur la même machine (<LaMachine>) que celle où tourne `rmiregistry`,
 - Lancer le client sur une machine quelconque, mais en étant dans le même répertoire pour retrouver les stubs :
 - `java HelloClient <LaMachine>`
- Le partage du bytecode (pour les stubs) est assuré par NFS, puisque le client et le serveur, même s'ils ne sont pas sur la même machine, partagent le même système de fichiers.
- Dans le cas général, il faut faire transiter les stubs par un serveur HTTP, FTP,



Synthèse

- Côté serveur

- Ecrire :

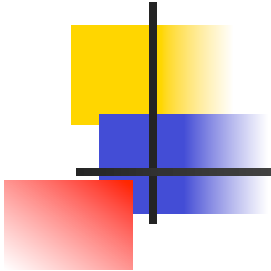
- Les interfaces "prototypant" les méthodes distantes (extends `java.rmi.Remote`)
 - Les objets qui les implémentent

- Lancer :

- 1- `rmic` : création du stub et du squelette à partir des fichiers `.class` des implémentations,
 - 2- `rmiregistry` : l'enregistrement des méthodes distantes sera fait lors de l'appel à `bind`
 - Rendre accessibles par un serveur Web ou Ftp les objets à télécharger (stubs, bytecode),

- Côté Client

- L'appel à `lookup()` passera un stub au client



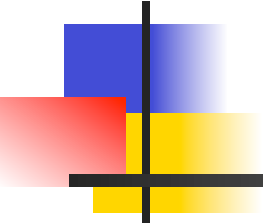
Déploiement de l'application

- Pour lancer un serveur :

```
java -Djava.rmi.server.codebase=http://perso.enst.fr/~$USER/tp  
-Djava.rmi.server.hostname=$HOSTNAME  
-Djava.security.policy=java.policy HelloServeur $*
```

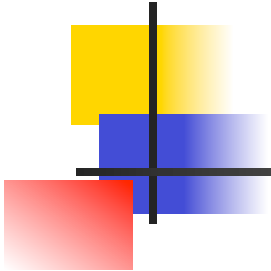
- `-server.codebase` donne le nom du serveur web d'où les stubs seront téléchargés,
- `-server.hostname` donne le nom de la machine où se trouve le serveur
- `-java.policy` donne les droits d'accès qui seront vérifiés par le security manager -
voici le contenu de ce fichier :

```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
};
```



Java RMI

Agent distribué et RMI

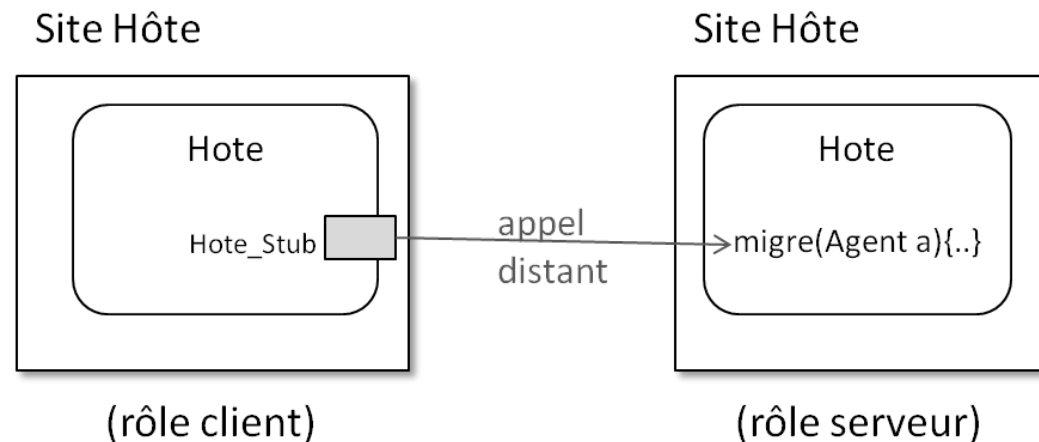


Exemple: agent mobile

- Un "client" voulant acheter un produit va créer un **agent** au lieu d'interroger lui-même tous les sites détenteurs de magasins vendant le produit.
- L'agent donnera au "client" le nom du site qui propose le meilleur prix.
- Mise en œuvre :
 - Le client va lancer sa demande depuis un site appelé **Initiateur**.
 - Il initialise un tableau de sites à parcourir, puis
 - Il invoque à distance sur le premier site la méthode **migre()**, à laquelle il a passé l'agent en argument.

Scénario

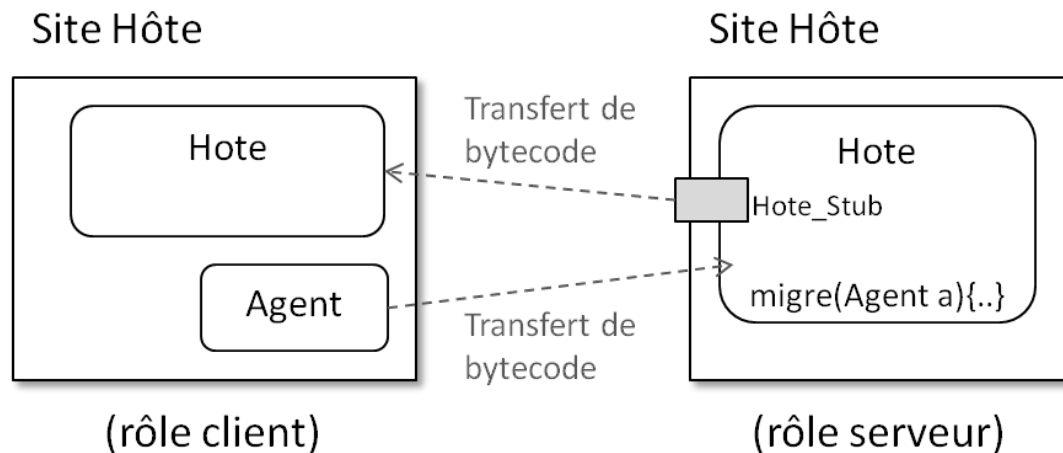
- Un "client" (Initiateur) va envoyer un Agent interroger successivement plusieurs serveurs (Hôtes)



- Chaque Hôte prendra successivement le rôle de serveur et de client
- Le site Initiateur démarre (et termine) le processus

Transfert de bytecode

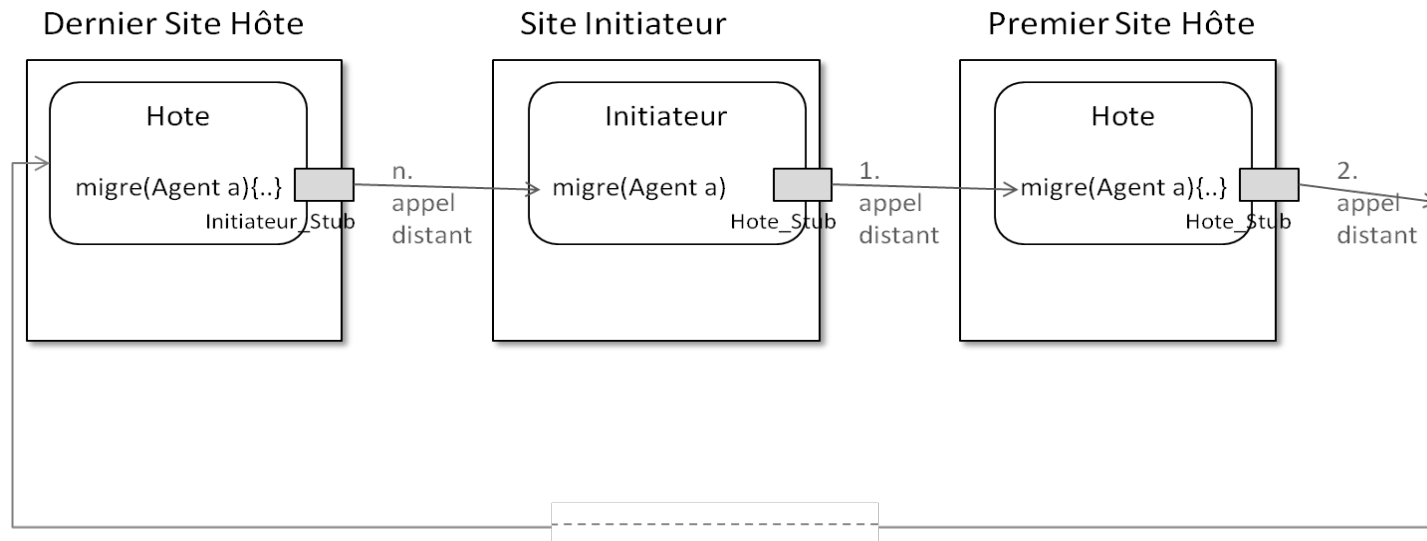
- Remarquez les différents objets téléchargés :
 - Le Stub (de Hôte et d'Initiateur) qui permet l'exécution distante de la méthode **migre**
 - L'Agent – transmis en paramètre de la méthode **migre**



NOTE: il ne faut pas confondre l'envoi de *l'instance agent* (en paramètre de la méthode **migre**) avec le transfert du *bytecode de la classe Agent*

Récapitulatif du fonctionnement

- La liste des Sites à parcourir se trouve dans l'agent
- Le Site Initiateur appelle le premier Site Hôte (dans la liste)
- Chaque Site Hôte appelle le prochain Site Hôte (en suivant la liste des Site de l'Agent)
- Le dernier Site Hôte appelle l'Initiateur
- L'agent est transmis en paramètre de chaque appel distant – `migre( agent )`
- La méthode `migre` de l'Initiateur est particulière - elle demande d'afficher les résultats obtenus



Descriptif des transferts de l'agent

- Observez ci-dessous le circuit parcouru par l'agent :

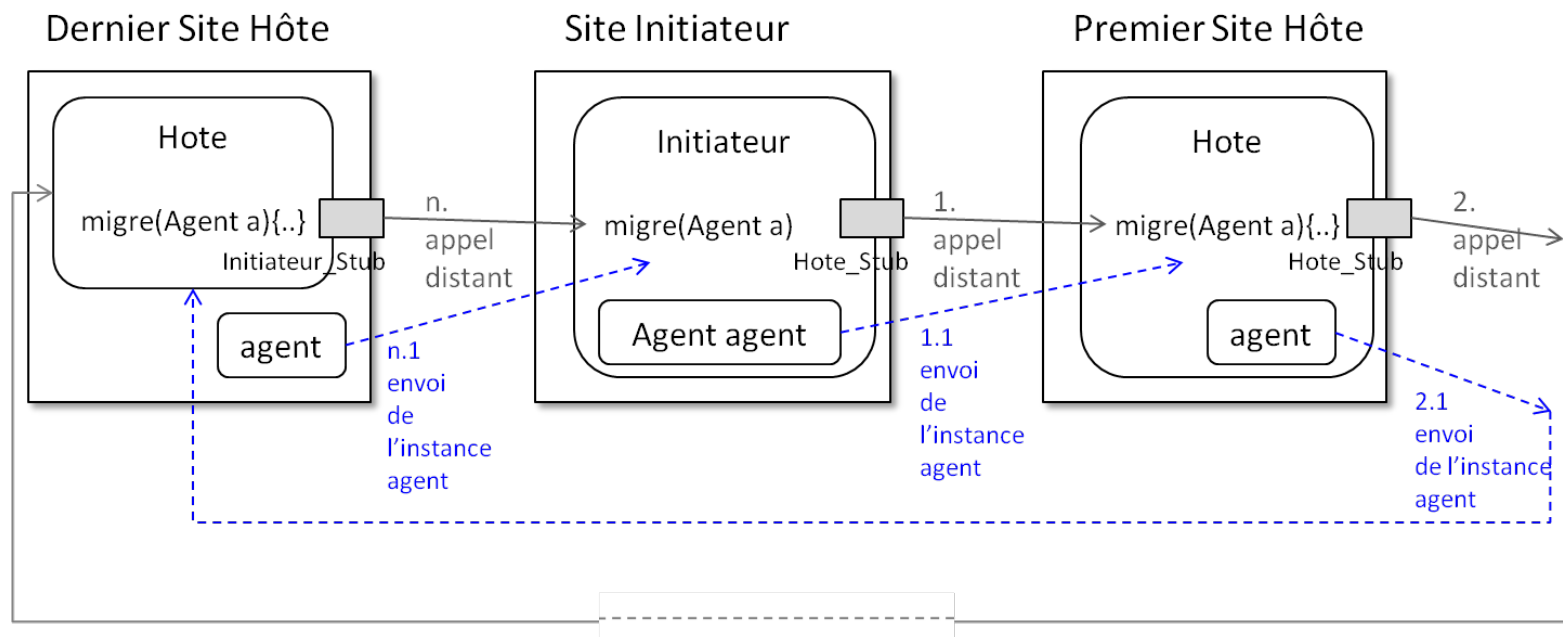
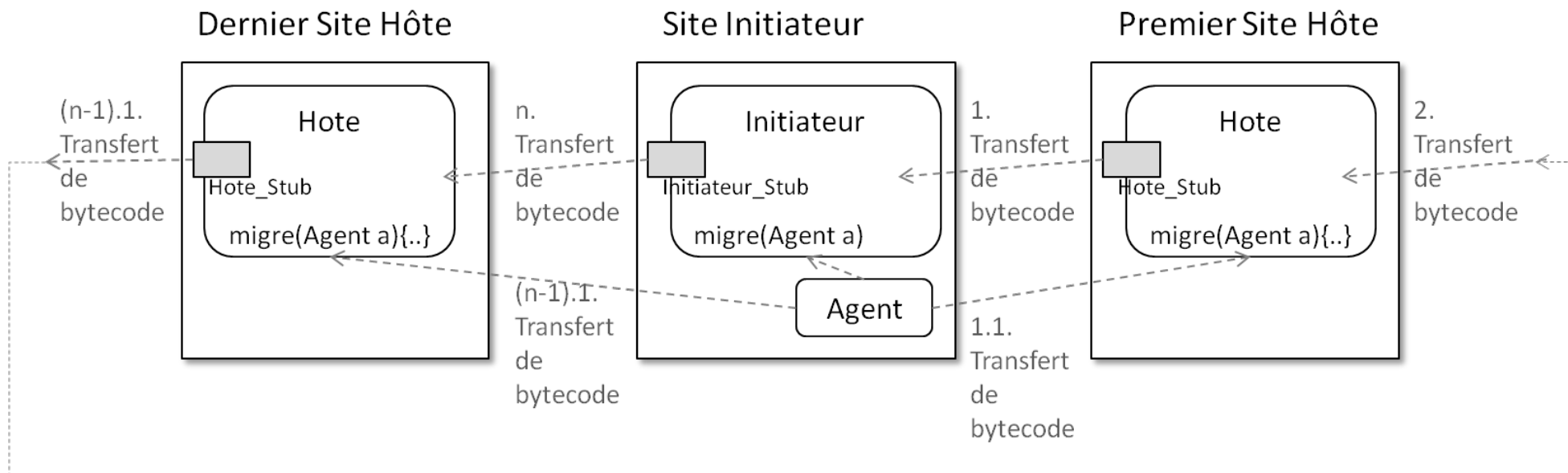
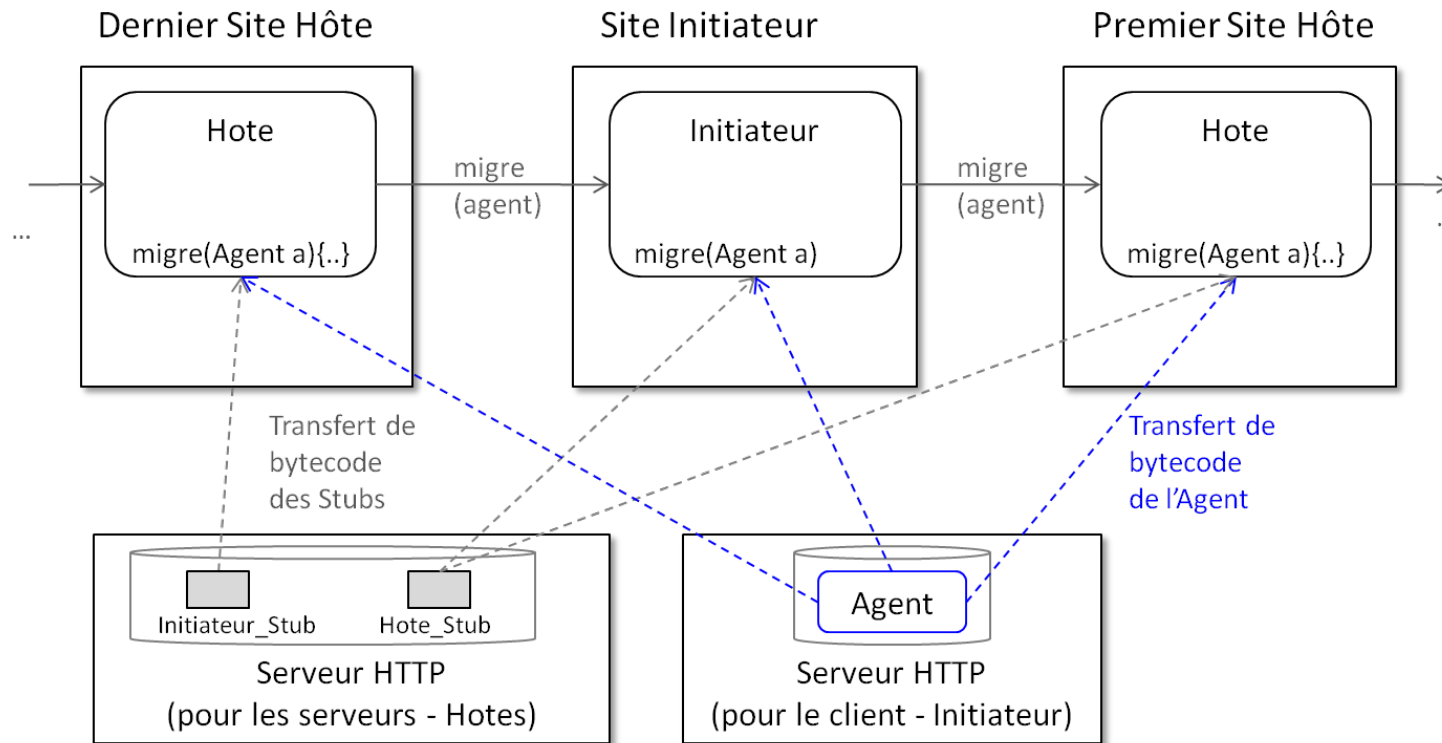


Schéma des transferts de bytecode - codebase local -



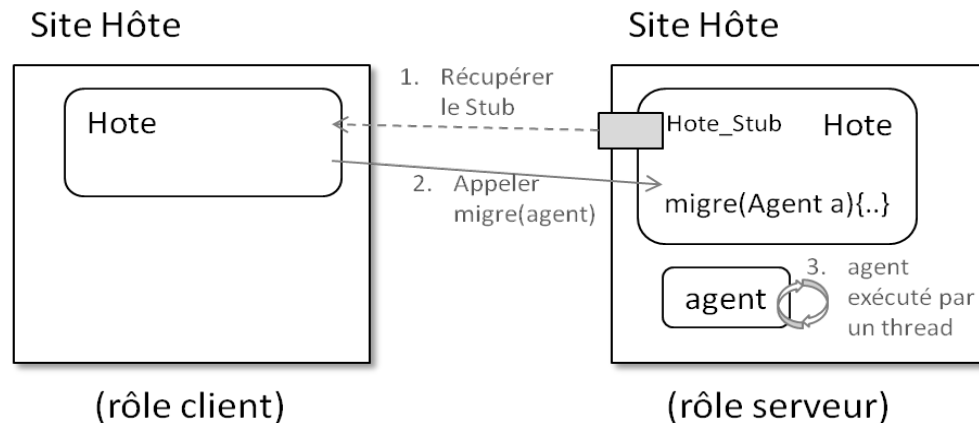
Résumé des transferts de bytecode

- codebase sur serveur http -



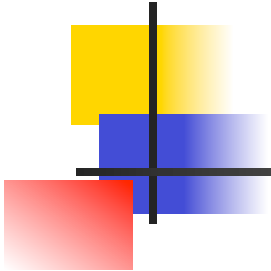
Le serveur

- Le serveur exécute la fonction **migre** pour le client :
 - il récupère alors le code de l'agent.
 - il le fait exécuter par un thread qui appelle lui-même migrer sur le site suivant :



- Voici l'interface fournie par le serveur ([Hote.java](#)) :

```
import java.rmi.Remote;  
public interface Hote extends Remote {  
    void migre(Agent a) throws RemoteException; }
```
- Le code d'Agent n'est pas présent lors de l'écriture de ce serveur



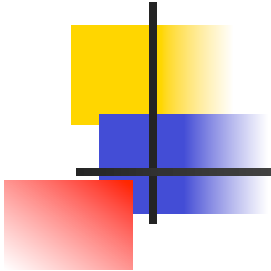
Le serveur

- Implémentation de l'interface (`Hote_implem.java`) :
 - La méthode `migre` invoquée à distance crée un `thread` - elle rend donc la main sans attendre la fin du traitement de l'agent (exécution puis migration).

```
public class Hote_implem
    extends UnicastRemoteObject implements Hote{

    public Hote_implem(String fichier) throws RemoteException{
        super();
    }

    public void migre(Agent a){
        threadAgent monThread = new threadAgent(a, monMagasin);
        monThread.start();
    }
}
```

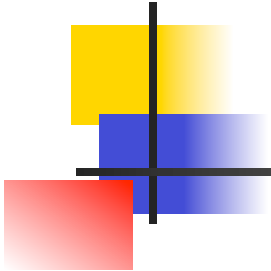


L'Agent

- `Agent.java` est l'interface pour les agents.
- L'objet sera implémenté par le "client" (Initiateur).

```
import java.io.Serializable;
public interface Agent extends Serializable {
    // Traitement effectue par l'agent sur chaque hote
    void traitement();
    // Affiche le resultat des traitements : effectue par
    // l'agent lorsqu'il revient sur le site initiateur
    void afficheResultat();
    // Renvoie le nom de l'hote suivant a visiter par l'agent
    String hoteSuivant();
}
```

- `Serializable` indique que les paramètres utilisés seront sérialisés et normalisés lors de l'appel distant (marshalling).

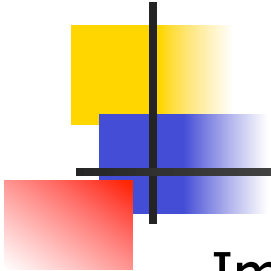


Le thread qui gère l'agent

- **ThreadAgent** est le support système offert par un Hôte pour traiter un agent : cette classe définit le thread qui va être lancé chaque fois qu'un agent arrive sur un site.

```
class ThreadAgent extends Thread
{
    public void run() {

        // On effectue ici le traitement demandé par l'agent
        // ...
        // On fait ensuite migrer l'agent vers l'hôte suivant
        Hote hote = (Hote) Naming.lookup(leSuivant);
        hote.migre(monAgent);
    }
}
```



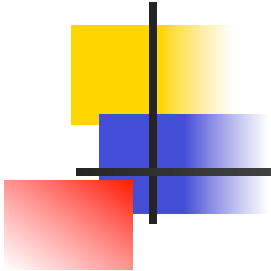
L'Initiateur

- Implémentation de l'Initiateur, c'est à dire du site "client" qui crée et lance l'agent vers les différents serveurs.
 - La méthode `migre` est ici particulière : elle ne déclenche pas l'exécution de l'agent mais lui demande d'afficher les résultats obtenus.

```
public class Initiateur extends UnicastRemoteObject
    implements Hote {

    public void migre(Agent a) {
        a.afficheResultat();
    }

    public static void main(String args[]) {
        AgentIngredient agent = new AgentIngredient();
        hote.migre(agent);
    }
}
```



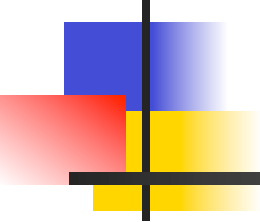
Initiateur

- Voici la commande pour lancer l'initiateur :

```
java
```

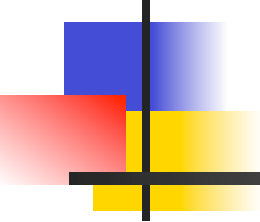
```
-Djava.rmi.server.codebase=http://perso.enst.fr/~$USER/tprmi  
-Djava.rmi.server.hostname=$HOSTNAME  
-Djava.security.policy=java.policy  
Initiateur $*
```

- `server.codebase` donne le nom du serveur web d'où l'agent sera téléchargé,
- `server.hostname` donne le nom de la machine où se trouve l'Initiateur
- le fichier `java.policy` donne les droits d'accès qui seront vérifiés par le security manager



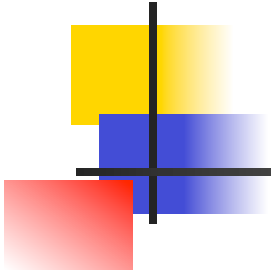
OMG CORBA

- www.omg.org
- www.corba.org



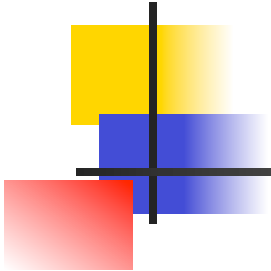
OMG CORBA

Introduction



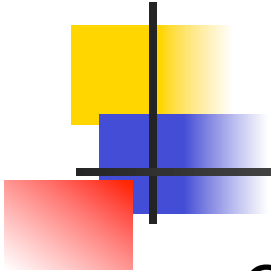
CORBA – contexte et motivations

- Contexte et besoins (~1990) :
 - La popularité croissante des systèmes répartis
 - L'interconnexion de systèmes informatiques variés via des diverses réseaux de communication
 - La réutilisation et l'intégration de systèmes existants (legacy)
- Contraintes :
 - Pas de consensus sur un seul langage de programmation, système d'exploitation, plate-forme matérielle ou protocole réseau
 - => Fortes besoins d'interopérabilité



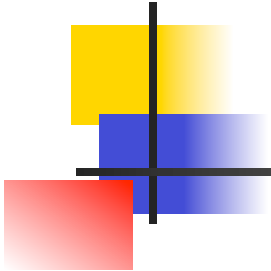
CORBA – approche et objectifs

- Proposer une **architecture et un intergiciel standards** pour les systèmes répartis :
 - Permettant l'**interopérabilité** entre des applications et des plates-formes **hétérogènes**
 - Basé sur des modèles de programmation **Orientés Objet**
- Environnement aux spécifications standardisées
- Conception modulaire par l'orienté objet
- Transparence vis à vis des détails d'implémentation



OMG – Object Management Group Organisation

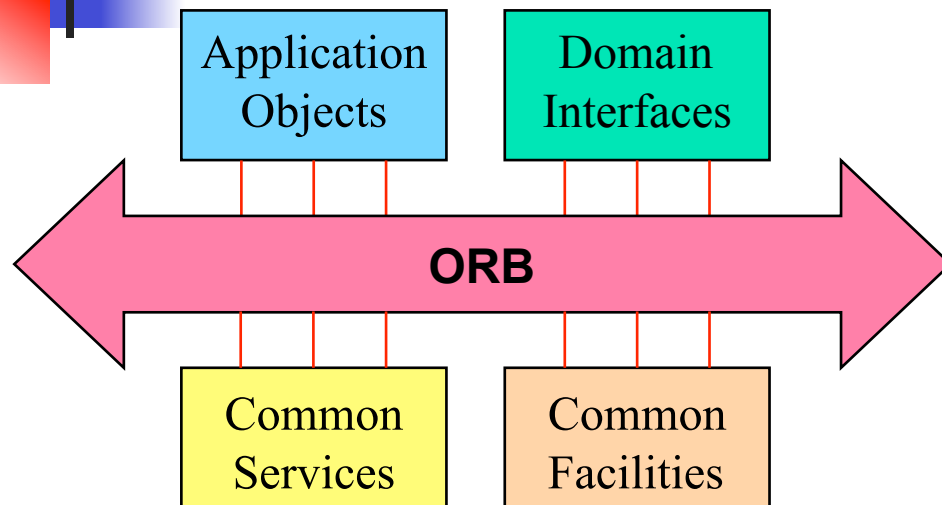
- Consortium créé en 1989 de plus de 800 membres
 - Constructeurs (Sun, HP, DEC, IBM, ...)
 - Environnements systèmes (Microsoft, Novell, ...)
 - Outils et Langages (Iona, Borland, ...)
 - Industriels (Boeing, Alcatel, ...)
- Mission
 - Promouvoir l'orienté objet dans les systèmes répartis
 - Fournir une architecture pour l'intégration d'applications réparties garantissant réutilisabilité, interopérabilité et portabilité (OMA)
- Objectif
 - Favoriser interopérabilité et portabilité d'applications réparties (CORBA) :
 - une terminologie unique dans le domaine de l'objet
 - un modèle d'objets abstrait
 - une architecture du modèle de référence
 - des interfaces et des protocoles communs



Définitions

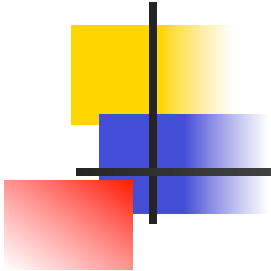
- **OMG : Object Management Group**
 - Consortium international, non-profit
- **OMA : Object Management Architecture**
 - Une architecture globale dédiée à la programmation orientée objet repart
- **CORBA : Common Object Request Broker Architecture**
 - Intergiciel (mechanisms et services)
- **IDL : Interface Description Language**
 - Langage commun de définition d'interfaces
- **ORB : Object Request Broker**
 - Système de communication pour objets répartis
 - Bus logiciel analogue à un bus matériel

OMA - architecture



- Application Objects
 - Applications mises en place par les développeurs
- Object Request Broker
 - Système de communication
 - CORBA: specs de l'ORB

- Common Object Services
 - Specs de services de base
 - Nommage
 - Événement
 - Transaction ...
- Common Facilities
 - Infrastructures indépendantes des domaines (horizontales)
 - Interface utilisateur
 - Gestion de l'information
 - Gestion du système
 - Gestion des tâches
- Domain Interfaces
 - Infrastructures dépendantes des domaines (verticales)
 - Simulation
 - Télécommunications ...



OMA

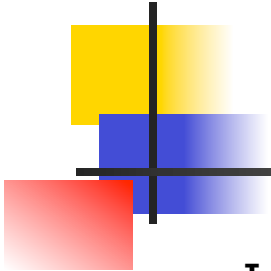
Objets d'application et Services communs

- Objets d'application

- spécifications d'interfaces IDL
- définis par une application de l'utilisateur;
- hors du champ de standardisation de l'OMG
- possibilité de standardisation pour des objets émergents

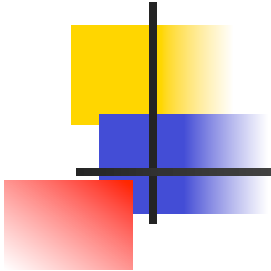
- Services communs

- spécification d'interfaces IDL
- leurs fonctionnalités peuvent être étendues ou spécialisées par héritage
- interfaces indépendantes des domaines d'application
- interfaces horizontales
- objectif : étendre les fonctions de l'ORB
- spécification de services
 - Nommage
 - Événement
 - Transaction
 - Sécurité
 - Persistance
 - ...



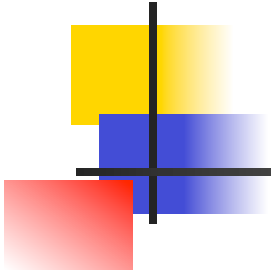
OMA - définitions

- Interface
 - Description d'un ensemble d'opérations disponibles sur un objet, spécifiée en IDL.
- Client
 - Entité capable d'émettre des requêtes vers des objets qui fournissent des services.
 - Le client manipule des références vers des objets distants.
- Référence ou proxy
 - Objet manipulé par le client pour invoquer des services sur un objet distant
 - Un proxy est un représentant local au client d'un objet distant
- Objet implémentation
 - Objet situé sur le serveur, implémente les méthodes des opérations définies en IDL
- Requête
 - Emise par un client, demande l'exécution d'une opération par un objet cible
 - Contient l'identifiant de l'objet cible, le nom de l'opération et les paramètres
- Opération
 - Entité identifiable caractérisée par une signature décrivant les paramètres de la requête et les valeurs de retour



CORBA - définition

- Common Object Request Broker Architecture
- Un standard qui définit une infrastructure ouverte et indépendante de fournisseur pour la programmation par objet reparté (Distributed Object Computing – DOC)
- Note: CORBA est un standard et pas un produit
 - Quelques implantations connues : Orbix et Orbacus de IONA, VisiBroker de Borland, ...



CORBA

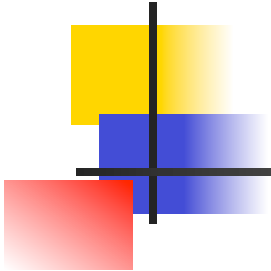
Implementations

- Commerciaux :

ORBIX	IONA	www.iona.com
Visibroker	Borland	www.borland.com
ORBacus	IONA	www.orbacus.com

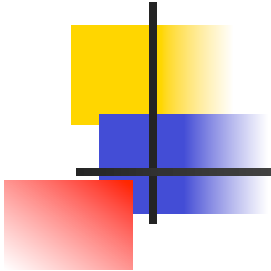
- Libres :

omniORB	omniorb.sourceforge.net
MICO	www.mico.org
ORBit	www.orbit.net
TAO	www.cs.wustl.edu/~schmidt/TAO.html
JacORB	www.jacorb.org
PolyORB	libre.act-europe.fr/polyorb



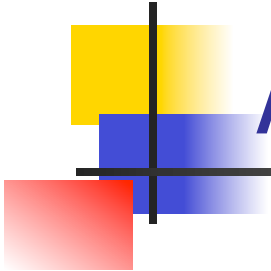
CORBA - objectifs

- Fournir une **spécification d'intergiciel** indépendante :
 - des fournisseurs et
 - des implantations
- Support pour la **hétérogénéité** :
 - Interopérabilité entre divers langages de programmation, plates-formes et réseaux
 - Via l'Interface Definition Language (IDL)
 - Transformation standard de l'IDL vers différents langages de programmation
- Support pour la **portabilité** :
 - Les applications peuvent être portées sur différents implantations de CORBA, provenant de fournisseurs différents
- Support pour **l'interopérabilité** :
 - Entre diverses implantations de CORBA
 - Via des protocoles standards de réseau :
General Inter-ORB Protocol (GIOP); Internet Inter-ORB Protocol (IIOP)



CORBA - principes fondamentaux

- Transparence à la localisation
 - l'utilisation d'un service est indépendante à sa localisation
- Transparence d'accès
 - les services sont accédés en invoquant des opérations sur des objets
- Séparation des interfaces et des implantations
 - les clients dépendent des interfaces, pas des implantations
- Interfaces typées
 - les références d'objet sont typées par les interfaces
- Support de l'héritage multiple d'interfaces
 - l'héritage permet de faire évoluer et de spécialiser les services



Applications CORBA

- Une application CORBA est un **ensemble d'Objets communicants**

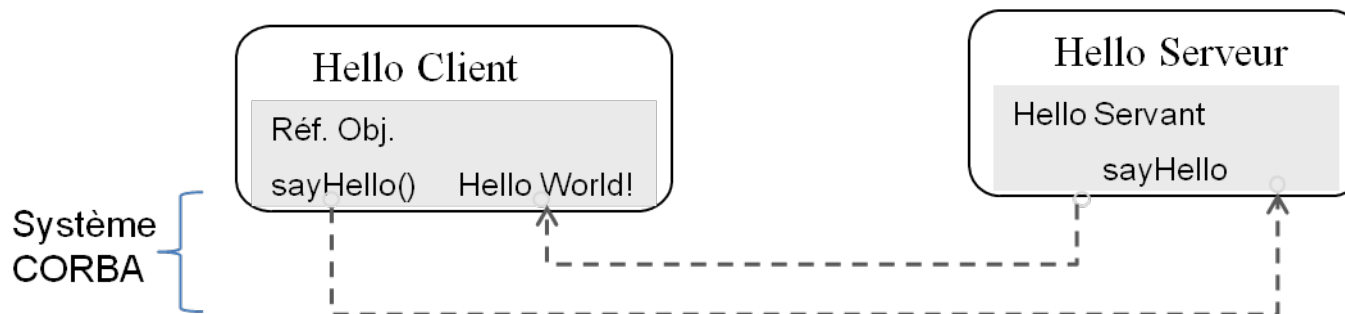
Rappel de définitions :

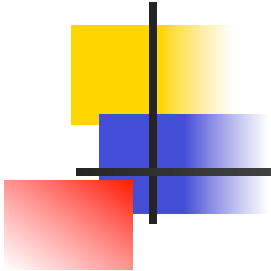
- Interface : ensemble d'opérations et d'attributs d'un objet
- Implantation : code associé aux opérations (définies dans une interface)
- Localisation : machine physique sur laquelle réside l'objet
- Référence : structure (« pointeur ») pour accéder à l'objet

- L'**interface** est un des éléments fondamentaux d'une application CORBA
 - Est définie dans un langage dédié (IDL)
 - Représente un contrat entre le client et le serveur
 - Définit les services que le serveur offre au client

CORBA – Hello World exemple

- Client Java : `hello_client.java`
- Serveur C++ : `hello_impl.h`, `hello_impl.cc`,
`hello_server.cc`
- Interface IDL : `hello.idl`



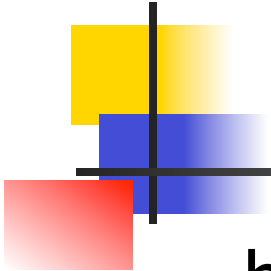


CORBA – Hello World exemple

Definir l'interface

- hello.idl :

```
interface Hello {  
    void sayHello();  
};
```

CORBA – Hello World exemple

Implanter l'Objet Hello (C++)

- hello_impl.h :

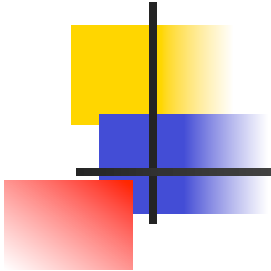
```
#include <hello_skel.h>

class Hello_impl : public Hello_skel{
    public: Hello_impl();
           virtual void sayHello();
};
```

- hello_impl.cc :

```
#include <CORBA.h>
#include <hello_impl.h>

Hello_impl::Hello_impl() { }
void Hello_impl::sayHello() {
    cout << "Hello World!" << endl;
}
```



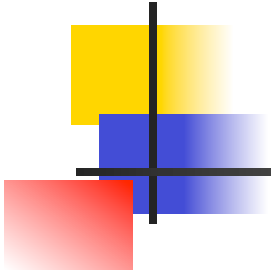
CORBA – Hello World exemple

Implanter le Serveur

- `hello_server.cc` :

```
#include <CORBA.h>
#include <hello_impl.h>
#include <fstream.h>

int main(int argc, char* argv[], char*[]) {
    CORBA_ORB_var orb = CORBA::ORB_init(argc, argv); //init. de l'orb
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
    Hello_var p = new Hello_impl; //instanciation obj Hello_impl
    CORBA_String_var sRef = orb -> object_to_string(p); //création réf
    const char* refFile = "Hello.ref";
    ofstream out(refFile);
    out << sRef << endl; //sauvegarde réf
    out.close();
    boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}
```



CORBA – Hello World exemple

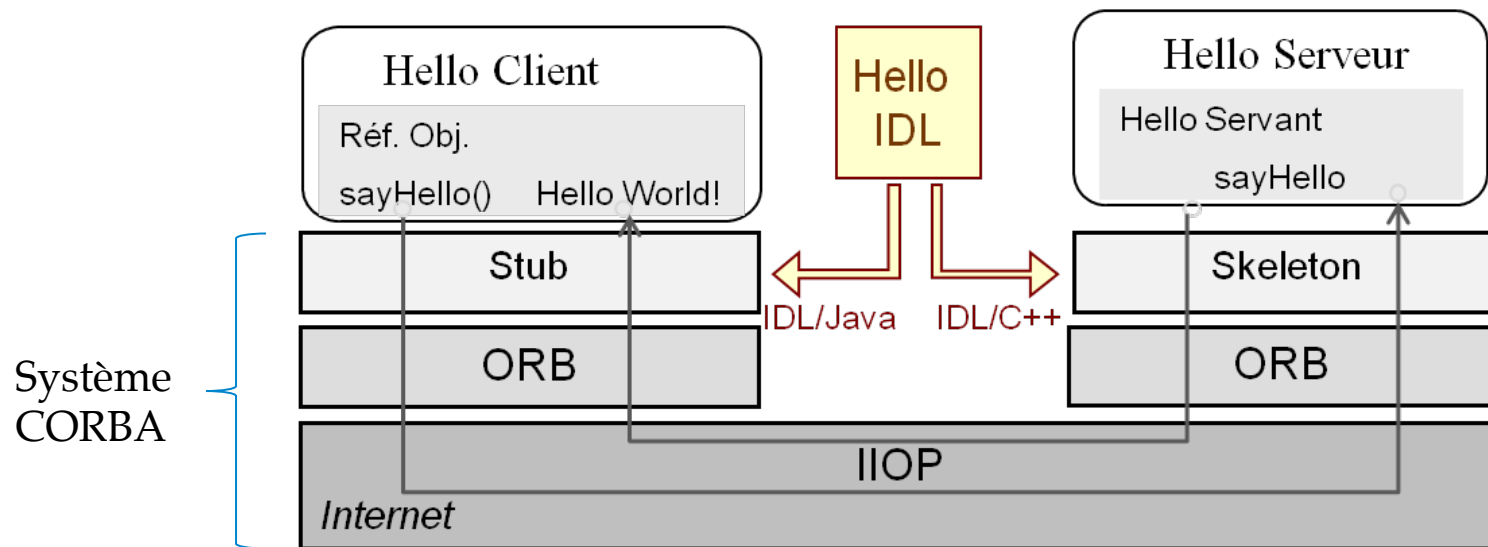
Implanter le Hello Client (Java)

- `hello_client.java` :

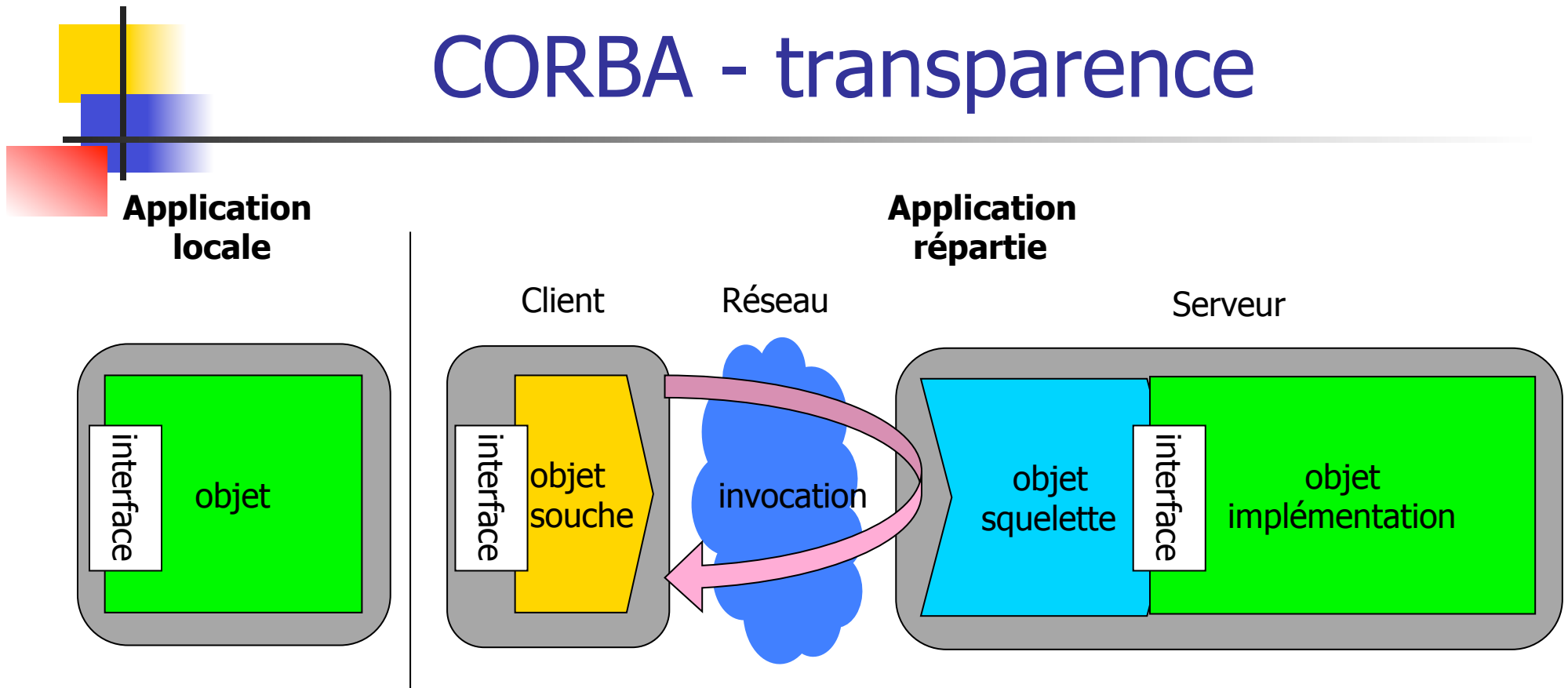
```
class hello_client {  
    public static void main( String args[] ) {  
        try{  
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();  
            IORHolder ior_holder = new IORHolder();  
            String iorString = ior_holder.readIORFile( "Hello.ref" );  
            org.omg.CORBA.Object object =  
  
            orb.string_to_object(iorString );  
            Hello hello = HelloHelper.narrow( object );  
            hello.sayHello(); }  
        catch ( org.omg.CORBA.SystemException e ) { ... }  
    }  
}
```

CORBA - Hello World exemple

- Vue d'ensemble



CORBA - transparence

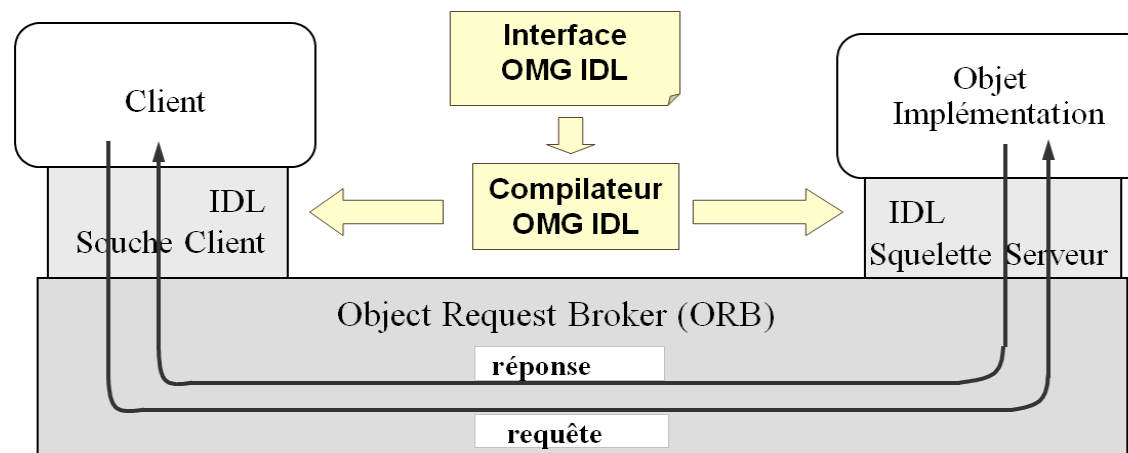


- **La souche** relie le client à l'ORB en transformant des appels de méthodes en émission de requête et réception de réponse
- **Le squelette** relie l'ORB à l'objet d'implémentation en transformant des appels de méthodes en réception de requête et émission de réponse
- => Pour le client et l'objet implémentation, l'emballage des paramètres, le transport des requêtes, la localisation des objets ... sont cachés

CORBA

Hétérogénéité et Interopérabilité (1)

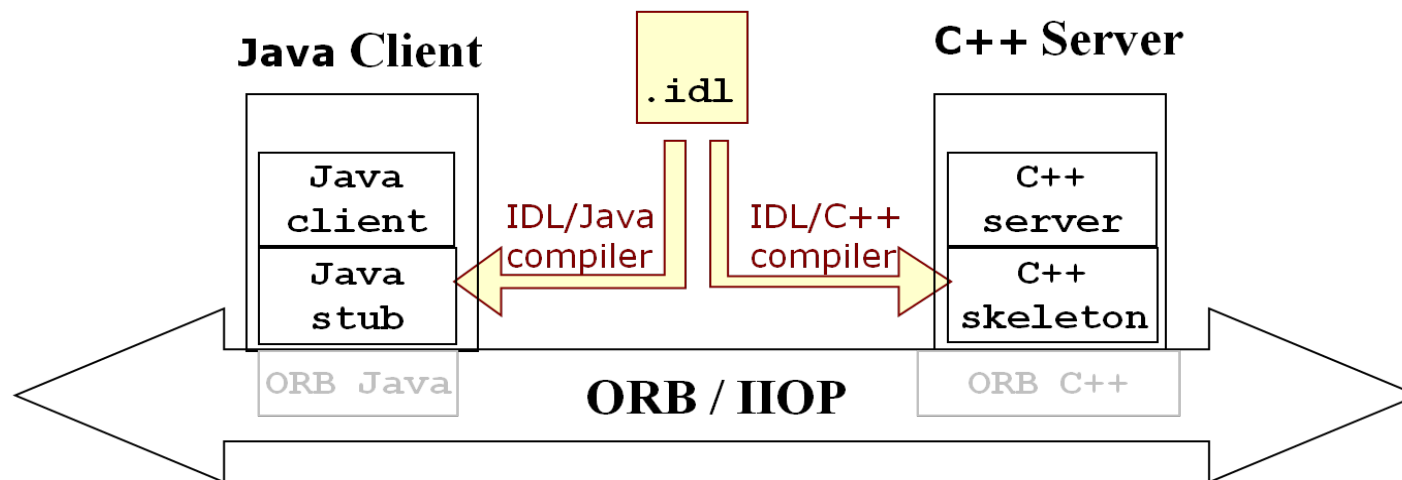
- Séparation entre l'interface et l'implémentation :
 - IDL standard pour définir les interfaces
 - Différents langages de programmation pour les implémentations
- => CORBA rend possible l'interopérabilité entre des langages de programmation différents



CORBA

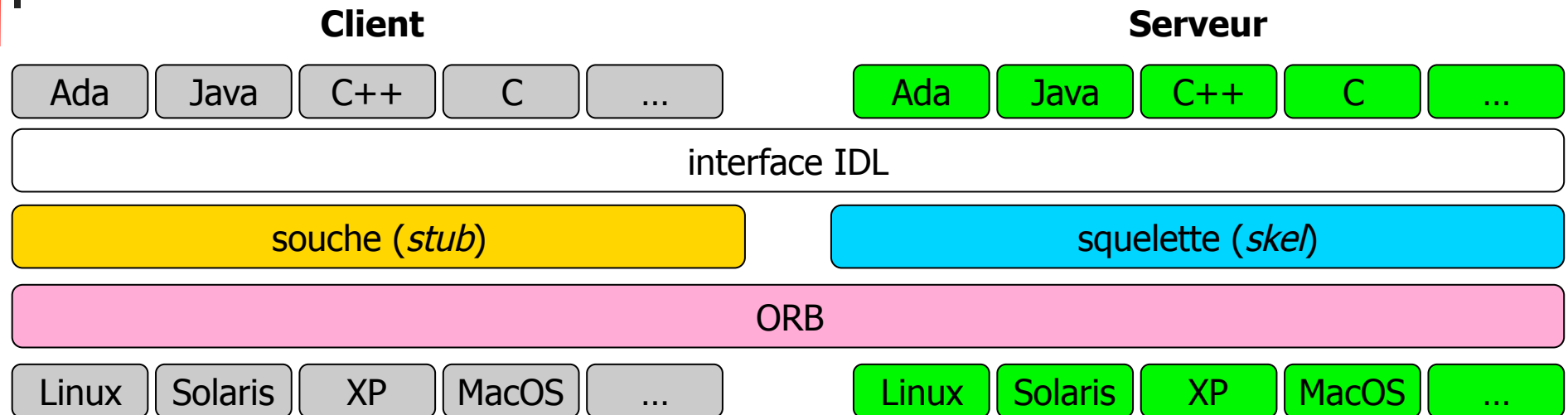
Hétérogénéité et Interopérabilité (2)

- Exemple :
 - Client Java
 - Serveur C++



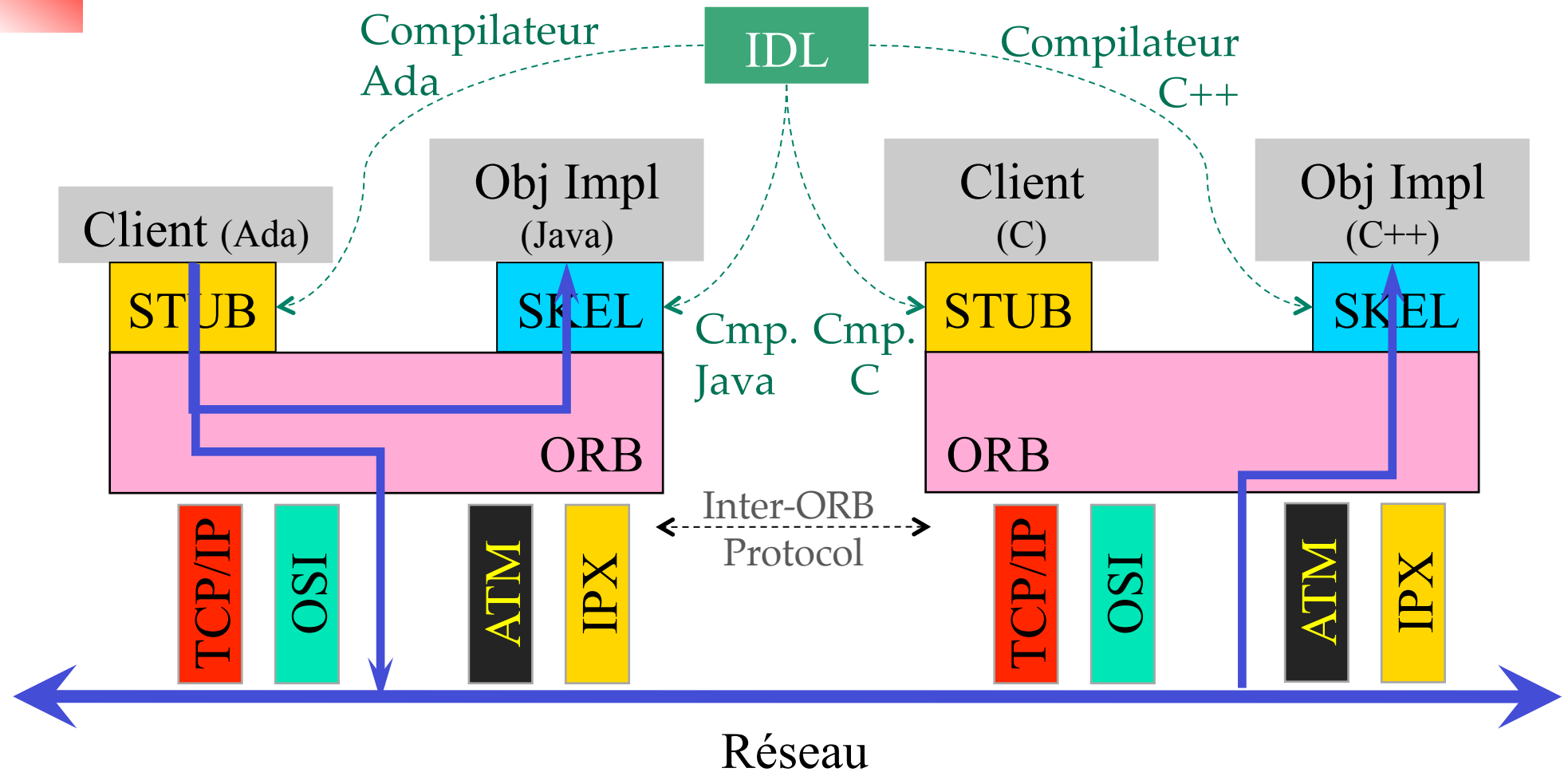
CORBA

Hétérogénéité et Interopérabilité (3)

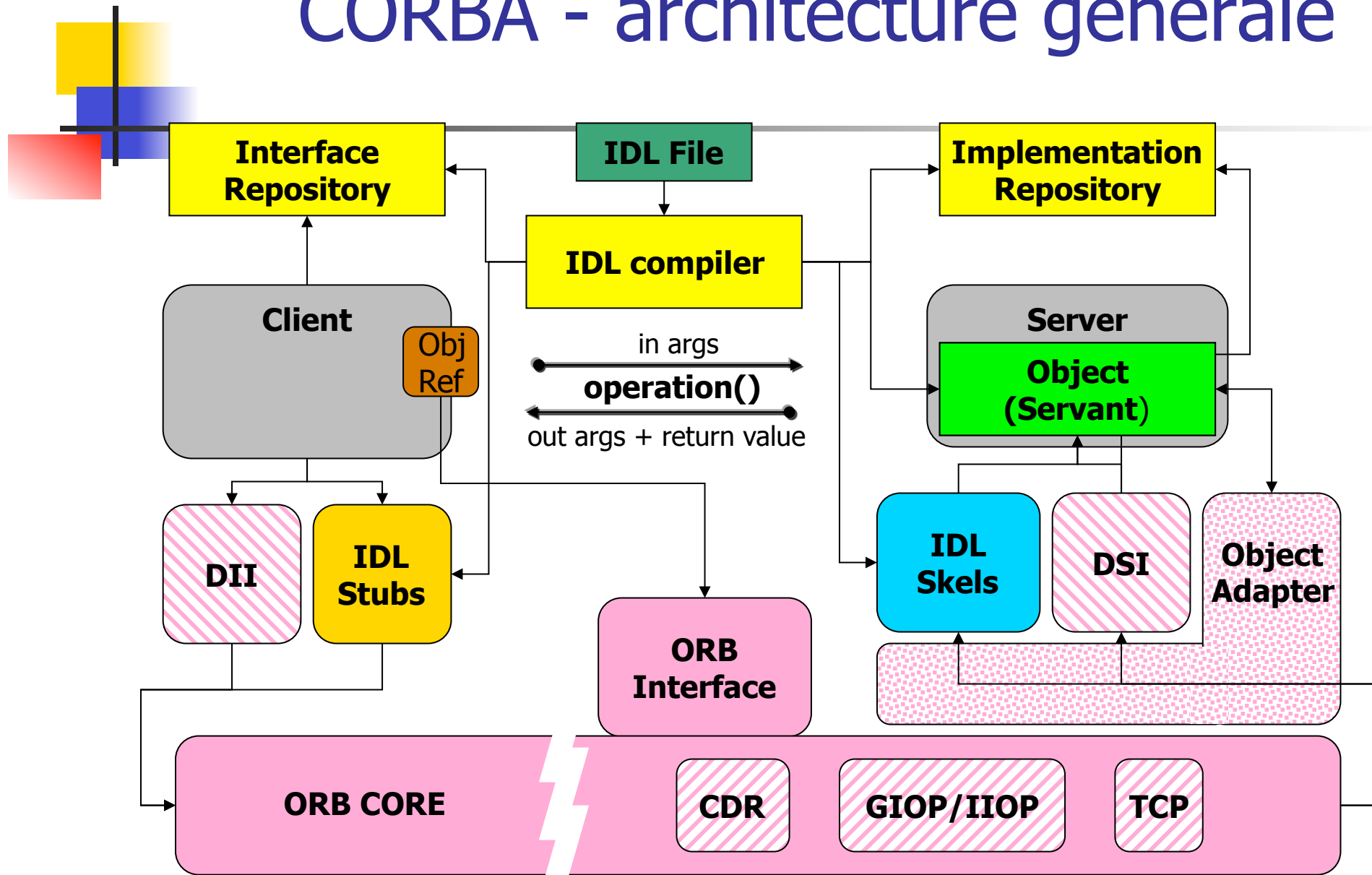


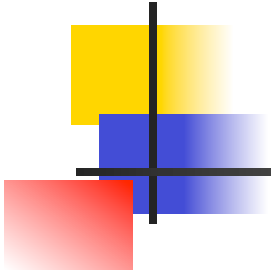
- Souche et squelette générés automatiquement par un compilateur pour un langage de programmation à partir de l'interface
- Interopérabilité pour traiter des différentes hétérogénéités (OS, langage, matériel, ...)
- Un compilateur applique à l'interface la projection du langage IDL vers un langage de programmation
- Un client écrit en L1 invoque une souche en L1 alors que ...
- Un squelette écrit en L2 invoque un objet implémentation en L2

CORBA - vue générale



CORBA - architecture générale





CORBA

ORB (Courtier ou Médiateur)

- ORB (Object Request Broker)
 - Composant central du standard CORBA qui fournit un point d'accès vers :
 - localisation et la désignation des objets
 - em/dépaquetage des paramètres (un/marshalling)
 - invocation des méthodes et gestion des exceptions
 - protocole de communication (TCP, ATM, ...)
 - gestion des ressources (processus, mémoire, ...)
- Interface de l'ORB (ORB Interface)
 - rend l'ORB accessible au travers d'un ensemble de primitives

CORBA

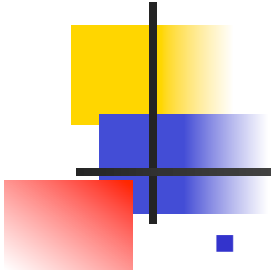
Souches statiques et dynamiques

- Souche
 - prépare les paramètres d'entrée de l'invocation
 - décode les paramètres de sortie et le résultat
- Souche statique
 - une par type d'objet serveur à invoquer
 - identique aux souches clientes RPC
 - générée à la compilation à partir de l'interface IDL
- Souche dynamique
 - souche générique construisant dynamiquement tout type de requêtes
 - permet d'invoquer des objets serveurs que l'on découvre à l'exécution (i.e. dont on ne connaît pas l'interface à la compilation)

CORBA

Squelettes statique et dynamique

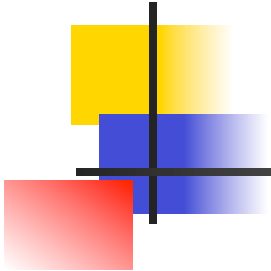
- Squelette
 - symétrique de la souche
 - décode les paramètres d'entrée des invocations
 - prépare les paramètres de sortie et le résultat
- Squelette statique
 - un par type d'objet serveur à invoquer
 - identique aux squelettes clients RPC
 - généré à la compilation à partir de l'interface IDL
- Squelette dynamique
 - squelette générique invoquant dynamiquement les méthodes de tout type d'objet d'implémentation
 - permet d'invoquer des objets serveurs que l'on découvre à l'exécution (i.e. dont on ne connaît pas l'interface à la compilation)



CORBA

Adaptateur et Référence

- Adaptateur d'objets
 - réceptacle pour les objets serveurs
 - interface entre les objets serveurs et l'ORB
 - gère l'instanciation des objets serveurs
 - crée les références d'objets
 - aiguille les invocations de méthodes vers les objets serveurs
 - plusieurs adaptateurs peuvent cohabiter sur une même machine dans des espaces d'adressage
- Référence d'objets
 - info identifiant de manière unique un objet dans l'ORB
 - <interface de l'objet> <adresse réseau> <clé de l'objet>
 - Interface de l'objet : différencie les types d'objets
 - Adresse réseau : adresse IP et numéro de port
 - Clé de l'objet : identité du couple adaptateur et objet

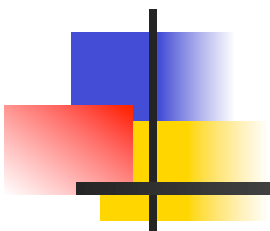


CORBA

Référentiels

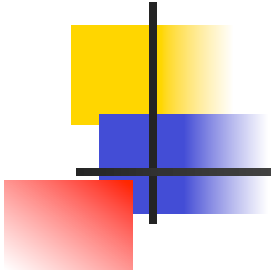
- Le référentiel d'interfaces (**Interface Repository**)
 - base de données des informations sur les types d'IDLs
 - opérations permettant l'accès dynamique aux informations (ainsi que la modification d'informations)
 - un par environnement (groupement logique de machines)
 - possibilité de fédérer les référentiels de différents environnements

- Le référentiel d'implantation (**Implementation Repository**)
 - base de données des informations sur:
 - la localisation de serveurs qui gèrent des objets distants – vers où faut-il acheminer les requêtes client pour un certain objet distant?
 - Les instructions à exécuter pour démarrer un serveur, dans le cas où ce serveur serait indisponible à l'arrivée d'une requête client
 - L'état courant de chaque serveur enregistré
 - un sur chaque site accueillant des objets serveurs



OMG CORBA

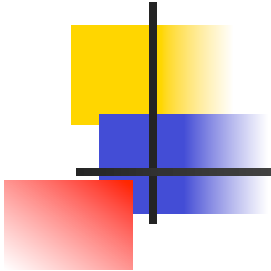
IDL - langage de description des interfaces
(Interface Definition Language)



OMG IDL

Interface Description Language

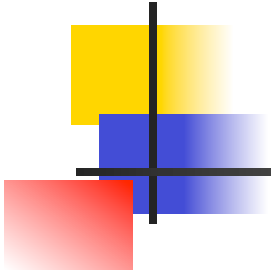
- Langage pivot de spécification - Espéranto entre les langages de programmation
- Utilisé pour décrire les interfaces d'Objets CORBA
- Une interface décrit les opérations et les attributs d'un Objet CORBA
- Indépendant des langages de programmation
- Projections (mapping) vers des langages de programmation orientés objet ou non
 - Standardisés : C++, Java, Ada, Smalltalk, C, Cobol
 - Exotiques: Python, Perl, Modula, ...
- Une projection est appliquée par un compilateur pour produire
 - Souches
 - Squelettes (et Patrons pour implémentation)
- Chaque ORB offre des compilateurs pour les langages qu'il supporte



OMG IDL

caractéristiques

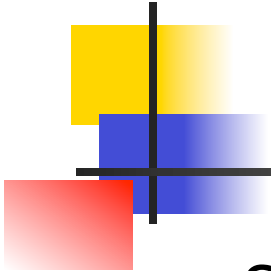
- Langage déclaratif fortement typé et orienté objet
- Opérations et attributs sur des objets
- Héritage multiple sans surcharge d'opérations ou d'attributs
- Encapsulation de l'objet implémentation



OMG IDL

syntaxe

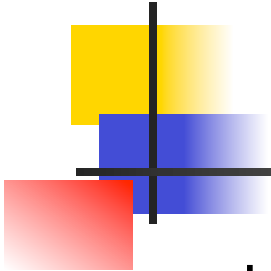
- Syntaxiquement proche du C++ mais ...
- Sémantiquement très différent avec ...
- De nouveaux mots-clés
 - module, interface, attribute, readonly, oneway, any, sequence
- Identificateurs:
 - séquence de caractères alphabétiques, chiffres ou _
 - casse des identificateurs n'est pas déterminante
 - typedef long duration;
 - typedef long Duration;
 - définissent le même type
 - respect de la casse de la définition dans une référence
 - typedef int duration;
 - typedef Duration period; // référence incorrecte à duration



CORBA IDL

définitions et syntaxe

- Sujets abordés :
 - Spécification (« name scoping ») et Module
 - Type
 - Constante
 - Interface
 - Attribut
 - Opération
 - Exception
 - Héritage
- Pour chaque sujet :
 - Description générale et exemple
 - Définition formelle



OMG IDL - spécification et module

- Une définition IDL est constituée de plusieurs modules
- Chaque module représente un contexte de nommage pour les définitions contenues
- Les modules et les interfaces forment des espaces de nommage

Exemple :

```
module BanqueDef {  
    interface Banque { //... };  
    interface Compte { //... };  
};
```

Référence à l'intérieur des interfaces

- Banque
- Compte

Référence à l'extérieur des interfaces

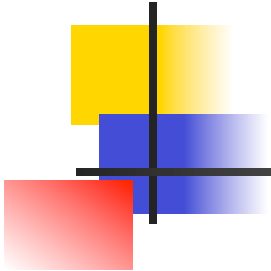
- BanqueDef::Banque
- BanqueDef::Compte

OMG IDL - spécification et module

```
<specification> ::= <definition>+
<definition> ::=
    <module> | <interface>
    | <type> | <constant> | <exception>
<module> ::= <definition>+
<interface> ::= <header> <body>
<body> ::= <export>*
<export> ::=
    <attribute> | <operation>
    | <type> | <constant> | <exception>
```

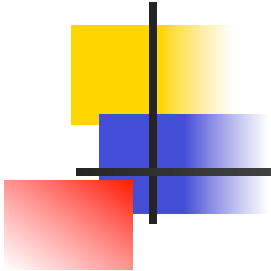
```
const long N = 100;
module Namespace {
    module Types {
        typedef string Name;
    };
    interface Group {
        ::Namespace::Types::Name
        Users[N];
    };
};
```

- Une spécification ou un module comporte
 - Des modules (imbriqués) servant d'espaces de noms définissant ...
 - Des interfaces définissant en plus des attributs et des opérations ...
 - Des types, des constantes, des exceptions, ...
- L'opérateur :: permet de référencer des entités



OMG IDL – types de données

- Types « simples »
 - `short, long, float, double, char, string, boolean, octet, ...`
 - `any` (permet la spécification des valeurs de tout type IDL)
 - `long long, long double, wchar, wstring, fixed, ...`
- Types « complexes »
 - `enum`
 - `struct`
 - `union`
 - `array`
 - `sequence`
- ...

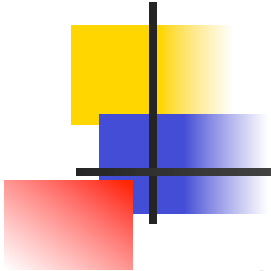


OMG IDL – le type `enum`

- Enumération
- Attribue une collection de valeurs possible à une variable
- A tout moment, le variable peut prendre une des valeurs spécifiées dans l'énumération

Exemple :

```
module BanqueDef {  
    enum Monnaie{euro, pound, dollar, yen};  
    interface Compte {  
        readonly attribute Montant solde;  
        readonly attribute Monnaie soldeMonnaie;  
        //... };  
};
```

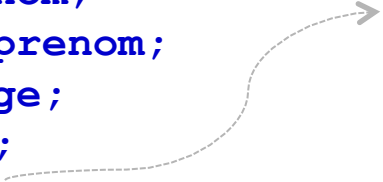



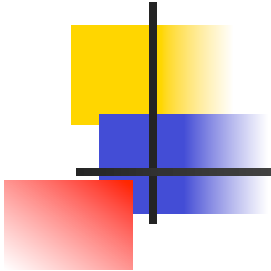
OMG IDL – le type `struct`

- Structure
- Permet de grouper différents membres
 - Chaque membre doit être nommé et typé
- Doit inclure au moins un membre

Exemple :

```
module BanqueDef {  
    struct DetailsClient {  
        string clientID;  
        string nom;  
        string prenom;  
        short age;  
        //... };  
    interface Banque {  
        DetailsClient getDetailsClient  
            (in string clientID);  
        //...  
    };  
};
```





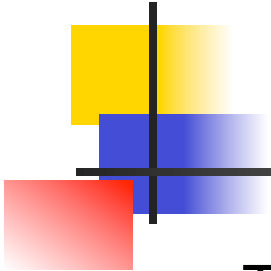
OMG IDL – le type **union**

- Définit une structure qui peut contenir seulement un des plusieurs membres à chaque moment
- Economise la mémoire
 - Consomme seulement l'espace requis par le plus grand des membres

Exemple :

```
enum typeDate { numerique, strMMJJAA, strJJMMAA }  
struct structureDate { short Jour; short Mois; short Annee; };  
union Date switch (typeDate) {  
    case numerique: long formatDigital;  
    case strMMJJAA: case strJJMMAA: string formatStr;  
    default: structureDate formatStruct;  
};
```

Le *discriminateur* peut être :
integer, char, boolean ou enum



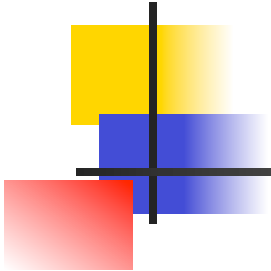
OMG IDL – le type `array`

- Tableau
- Permet de définir des tableaux
 - multidimensionnels
 - pour tout type de donnée IDL
 - dimensions prédéfinies

Exemple :

```
typedef Compte portfolio[MAX_TYPES_COMPTE][MAX_COMPTES]
```

- Doit être défini avec **typedef** pour être utilisé comme paramètre, attribut ou résultat retourné



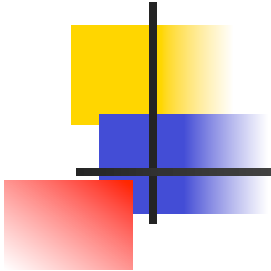
OMG IDL – le type *sequence*

- Tableau unidimensionnel
 - pour tout type de donnée IDL
 - dimensions prédéfinies ou flexibles (non-définies)

Exemple :

```
struct ComptesLimites {  
    string banqueID<10>;  
    sequence<Compte, 50> comptes; // max. longueur de la séquence est  
    50 };
```

```
struct UnlimitedAccounts {  
    string banqueID<10>;  
    sequence<Compte> comptes; // pas de longueur max. de la séquence  
};
```



OMG IDL

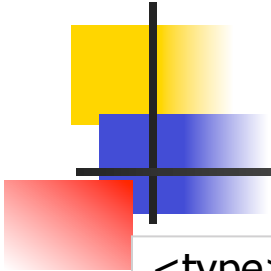
définition de types de données

- **Typedef**
- Permet d'attribuer de noms plus simples / pratiques aux types de données existants

Exemple :

```
module BanqueDef {  
    interface Compte { //... };  
    typedef Compte CompteStandard;  
};
```

//**CompteStandard** constituera un alias pour le type **Compte** dans les définitions IDL ultérieures



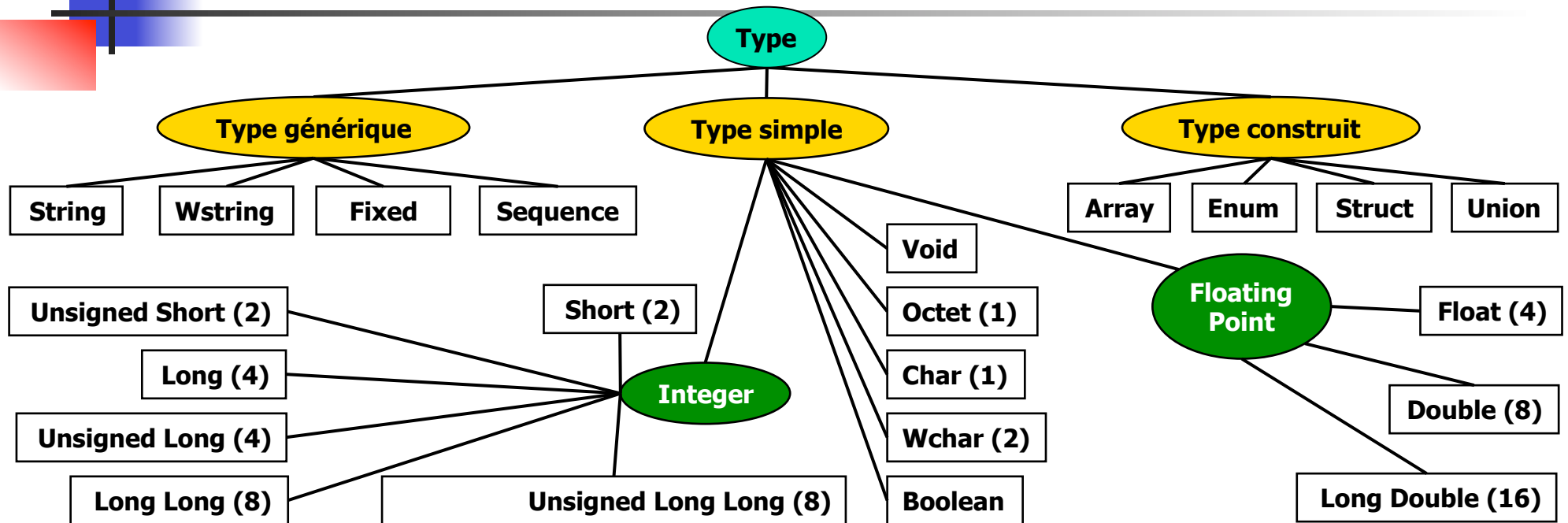
OMG IDL – types de données

```
<type> ::= <constructed_type>
| <simple_type> | <template_type>
<constructed_type> ::= <union_type>
| <struct_type> | <enum_type> | <array>
<simple_type> := <floating_point_type>
| <integer_type> | <object_type>
| <any_type> | <octet_type> | char_type>
| <wide_char_type> | <boolean_type>
<template_type> := <sequence_type>
| <string_type> | <wide_string_type>
| <fixed_point_type>
```

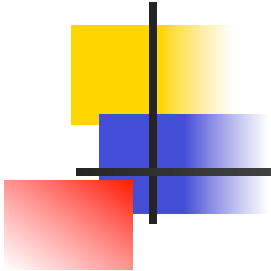
```
typedef string Name;
sequence <Name> Names;
struct Group {
    string Aliases[3];
    Names Users;
};
enum Sex {Male, Female};
union Status switch (Sex) {
    case Male: boolean Bearded;
    default:    long Children;
};
```

- Typedef: Définition d'un nouveau type à partir d'un type existant
- Union: type semblable à une union en C – ensemble de choix - paramétré par un discriminant de type discret (switch)
- Enum: type discret composé de valeurs symboliques
- Array: type tableau implicite et contraint

OMG IDL – types de données



- Any: type opaque dont la gestion (em/déballage) revient à l'utilisateur
- Object: type de référence d'objet dont dérive toutes les interfaces
- Séquence: type tableau générique contraint (sequence <type, size>) ou non contraint (séquence <type>)
- Fixed point: flottant à virgule fixe (456,78 correspond à fixed <5,2>)



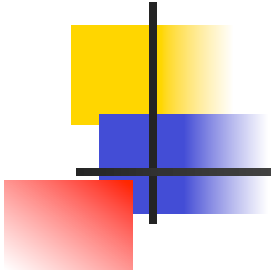
OMG IDL - constante

```
<constante> ::= "const" <constante_type>
               <identifieur> "=" <expression>
<exp> ::= [<subexp>] <operator> <subexp>
<operator> ::= <unary_operator>
               | <binary_operator>
```

- Le type de la constante doit être un type de taille prédéfinie
- La valeur d'une sous-expression <subexp> doit être valide pour le type de la constante

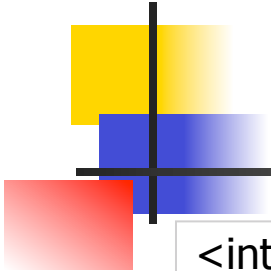
```
const long N_Components = 150;
const long Component_Size = 8;
const long Component_Table_Size =
    N_Components * Component_Size ;
const string Warning = "Beware !";
const Octet Invalid = 2 * 150 - 60;
// (2 * 150) - 60 = 300 - 60 = 240
```

Unary operator	
+ - ~	plus, minus, not
Binary operator	
^ &	or, xor, and
<< >>	shift left, shift right
* /	mul, div
+ - %	plus, minus, mod



OMG IDL - interface

- Une définition d'interface IDL contient typiquement :
 - Définitions d'attributs
 - Définitions d'opérations
 - Définitions d'exceptions
 - Définitions de types
 - Définitions de constantes
- Doivent être spécifiés à l'intérieur d'une interface
- Peuvent être spécifiés en dehors (au plus haut niveau) de définitions d'interfaces

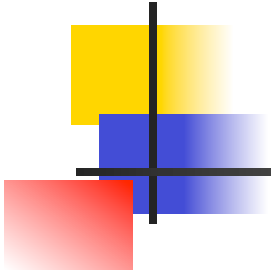


OMG IDL - interface

```
<interface> ::= <header><body>
<header> ::= "interface" <identifier>
    [: <inheritance>]
<inheritance> ::= <interface_name>
    {, <interface_name>}
<body> ::= <export> *
<export> ::= <attribute> | <operation>
    | <type> | <constant> | <exception>
```

```
interface Chicken;
interface Egg;
interface Chicken {
    enum State {Sleeping, Awake};
    attribute State Alertness;
    Egg lay();
};
interface Egg {
    Chicken hatch();
};
```

- L'interface représente la notion fondamentale propre aux objets répartis
- L'héritage multiple et répété (en losange) est possible
- Toute interface dérive du type de référence CORBA::Object
- La pré-déclaration constitue une solution pour les visibilités croisées



OMG IDL - attribut

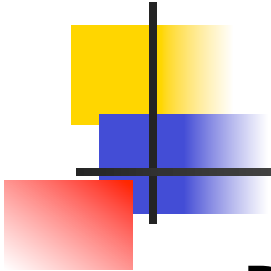
```
<attribute> ::= " attribute " [" readonly "]  
    <type> <declarators>  
<declarators> ::= <declarator> {, declarator}
```

exemple de getter/setter en Java:

```
java.lang.string Title ();  
void Title (java.lang.string value);  
float Balance ();
```

```
interface Account {  
    attribute string Title;  
    readonly attribute float Balance;  
};  
struct Coord {  
    unsigned long X, Y;  
};  
interface Triangle {  
    attribute Coord Points[3];  
};
```

- L'accès à un attribut se fait au travers de méthodes (*getter* et *setter*) quelque soit le langage de programmation
- Un attribut readonly n'est accessible qu'en lecture et ne dispose donc que d'un *getter*

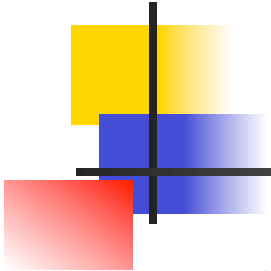


OMG IDL - opération

- Définit la signature d'une fonction de l'objet
- La signature contient généralement :
 - Le type de donnée du résultat
 - Des paramètres et leurs directions
 - Des exceptions

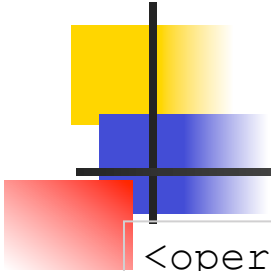
Exemple :

```
module BanqueDef {  
    typedef float MontantSolde; // Type pour représenter le solde  
    //...  
    interface Compte{  
        exception FondsInsuffisants {};  
        void retirer(in Montant montant) raises (FondsInsuffisants);  
        void déposer(in Montant montant);  
    }  
}
```



OMG IDL – direction de paramètres

- Chaque paramètre indique sa direction de passage entre le client et l'objet
- Paramètre :
 - **in** : initialisé seulement par le client et passé à l'objet
 - **out** : initialisé seulement par l'objet et passé au client
 - **inout** : initialisé par le client et passé à l'objet; l'objet peut modifier la valeur avant la retourner au client
- Permet au système CORBA de savoir dans quel sens encoder/décoder (marshalling/unmarshalling) les paramètres



OMG IDL - opération

```
<operation> ::= ["oneway"] <type>
    <identifier> <parameters>
    [<raise>] [<context>]
<parameters> ::= <parameter>*
<mode> ::= "in" | "out" | "inout"
<parameter> ::= <mode> <type>
    <identifier>
<raise> ::= "raises" (<exception>,
    {, <exception>})
<context> ::= "context" (<string>,
    {, <string>})
```

```
interface Stack {
    exception Empty;
    readonly attribute long Length;
    oneway void Push (in long Value);
    long Pop () raises (Empty);
};
```

- Une méthode peut lever une ou plusieurs exceptions
- L'invocation d'une méthode peut donner lieu à un passage de contexte d'exécution auprès du serveur (variables d'environnement)
- Une méthode oneway (sans paramètre inout, out, sans exception, sans paramètre de retour) est asynchrone (non-bloquante)



OMG IDL - exception

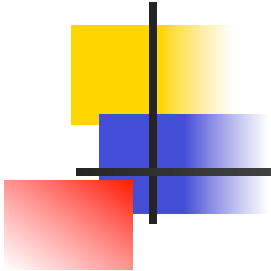
```
<exception> ::= " exception" <identifier>
               <member>*
<member> ::= <type> <declarators>
<declarators> ::= <declarator>
                  {, >declarator}
```

exceptions prédéfinies:
OBJECT_NOT_EXIST
COMM_FAILURE
BAD_PARAM
...

```
exception No_Such_Name ;
exception Not_Unique {
    long Many;
};
```

```
long Lookup (in string Name)
    raises (No_Such_Name,
           Not_Unique) ;
```

- Il existe des exceptions définies par l'utilisateur, d'autres prédéfinies par CORBA et d'autres enfin propres au vendeur
- Dans une méthode, l'exception correspond à un paramètre de retour dont on peut interroger l'état

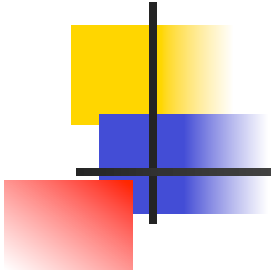


OMG IDL – héritage

- Une interface peut hériter de plusieurs interfaces
- Tous les éléments de l'interface héritée sont disponibles dans l'interface dérivée

Exemple :

```
module BanqueDef{  
    typedef float Montant; // Type pour représenter le montant  
    interface Compte{ //... };  
    interface CompteCourant : Compte{  
        readonly attribute Montant limiteDecouvert;  
        boolean commanderChequier ();  
    };  
    interface CompteEpargne : Compte {  
        float calculerInteret ();  
    };  
};
```

OMG IDL – héritage

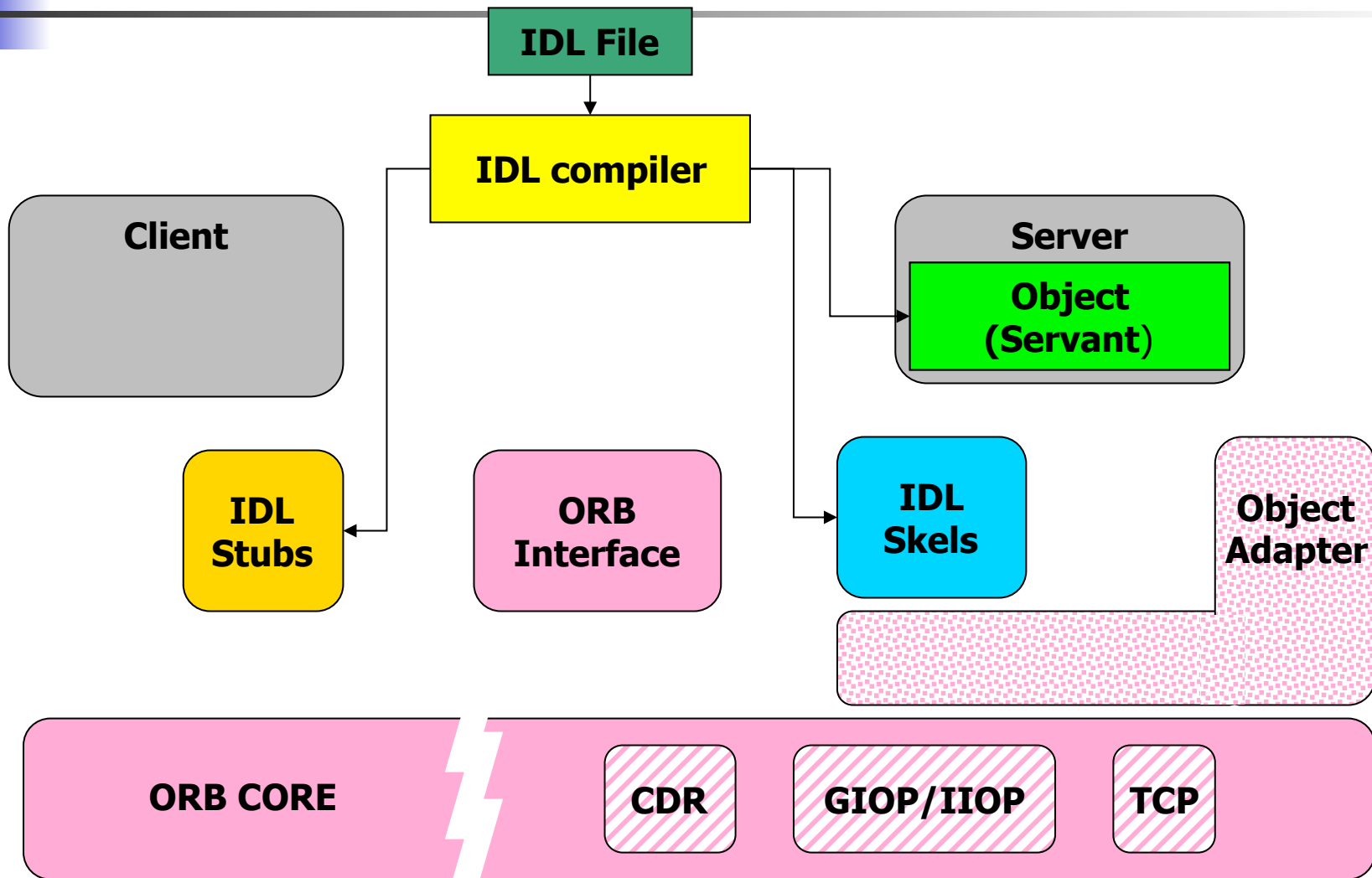
```
<interface> ::= <header><body>
<header> ::= "interface" <identifier>
    [: <inheritance>]
<inheritance> ::= <interface_name>
    {, <interface_name>}
<body> ::= <export> *
<export> ::= <attribute> | <operation>
    | <type> | <constant> | <exception>
```

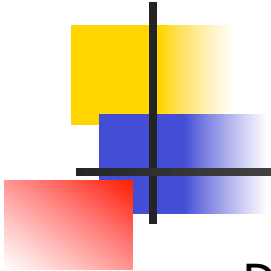
```
interface Bird {
    void eat ();
};
interface Egg;
interface Chicken : Bird {
    Egg lay();
};
interface Egg {
    Chicken hatch();
};
```

- Une interface peut hériter de plusieurs interfaces
- Une interface hérite de constantes, types, exceptions, attributs et opérations de son interface parente
- L'héritage autorise la surcharge de types, constantes et exceptions
- L'héritage interdit la surcharge d'attributs et opérations
- La répétition d'un héritage se fait avec une seule instance

CORBA

Développement

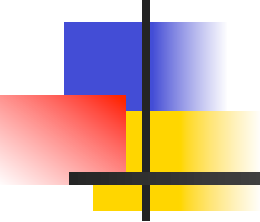




CORBA

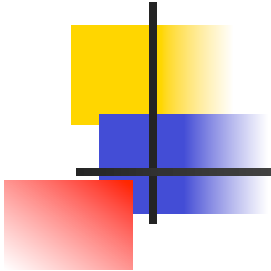
Développement

- Développement en OMG IDL
 - Conception d'une interface OMG IDL
 - Génération pour un langage de programmation
 - D'une souche
 - D'un squelette et d'un patron de l'implémentation
- Développement en langage de programmation
 - Développement de l'implémentation à partir du patron
 - L'arbre d'héritage dans l'arbre de l'IDL est souvent totalement différent de celui du langage de programmation
 - Enregistrement des objets auprès de l'ORB chez le serveur
 - Obtention de références auprès de l'ORB chez le client
 - Première référence étant souvent celle d'un serveur de noms
 - Référence bien connue, chaîne de caractères (fichier, cmdline)
 - Courbe d'apprentissage forte selon la projection du langage (Ada << Java << C++ <<<< C)



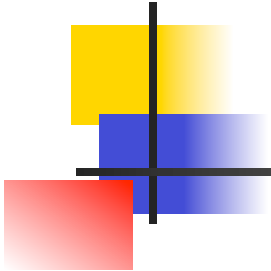
OMG CORBA

Projection IDL vers Java



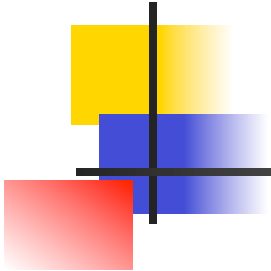
IDL-JAVA: objectif

- Générer les classes Java nécessaires pour permettre :
 - Aux clients Java d'accéder à des Objets CORBA
 - Aux objets Java (servants) d'être publiés et rendus accessibles en tant qu'Objets CORBA
- Exemple : stubs, skeletons, interfaces Java, ...



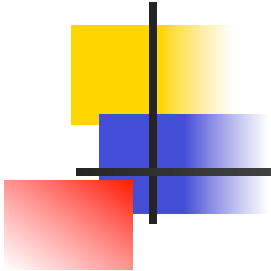
IDL-Java – projection automatique

- Compilateur IDL vers Java
 - conforme au standard OMG
- Les noms et les identificateurs IDL sont projetés en Java sans modification
- Un élément IDL peut être projeté en plusieurs éléments Java
 - Exemples : interfaces IDL ou différents structures IDL (enum, struct, union, ...)



IDL-JAVA : fichiers générés

- Une interface IDL est projetée en plusieurs classes Java : nom original + suffixes
- Exemple: BanqueDef.idl >> compilation IDL – Java :
 - BanqueDef.java - interface (côté client / applicatif)
 - BanqueDefOperations.java - interface (côté serveur / Obj. CORBA)
 - BanqueDefHelper.java - méthodes liées au type définit
 - BanqueDefHolder.java - emballage params `out/inout`
 - _BanqueDefStub.java - stub
 - BanqueDefPOA.java - skeleton
 - BanqueDefPOATie.java - skeleton (variant pour la délégation)
- Le développeur doit implanter la class BanqueDefImpl.java



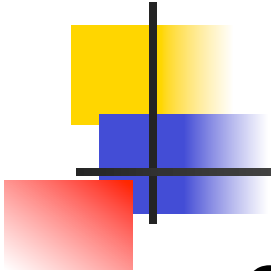
IDL – Java : class Helper

- Offre des opérations statiques pour manipuler le type
- Exemple :

```
org.omg.CORBA.Object obj =  
    orb.string_to_object (ior);
```

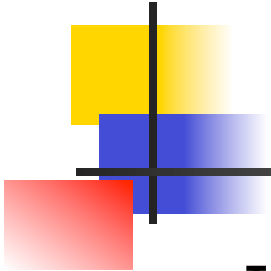
//ior : la référence de l'objet CORBA

```
Hello hello =>HelloHelper.narrow (obj);
```

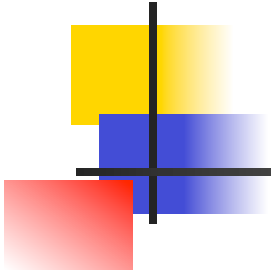
IDL – Java : class Holder

- Offre du support pour l'implantation de paramètres de type `out` et `inout`
- Exemple
- Definition.idl
 - `void methode(out long i) ;`
- Client.java
 - `LongHolder myLongHolder =`
`new LongHolder(17) ;`
 - `myCORBAObj.methode(myLongHolder) ;`
 - `myLongHolder.value . . .`



Java-IDL : projections « directes »

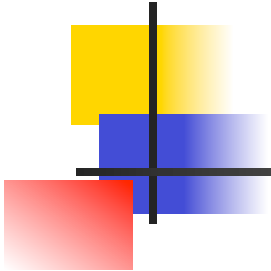
- IDL module → Java package (même nom)
- IDL interface → Java public interface (même nom)
- Héritage d'interfaces IDL → héritage d'interfaces Java
- Attributs IDL → une paire de méthodes Java (get/set)
 - Si l'attribut IDL est défini comme **readonly**
→ seulement une méthode get en Java



IDL-Java - Projection

IDL	Java
[unsigned] short	short
[unsigned] long	int
[unsigned] long long	long
float	float
double	double
long double	<i>java.math.BigDecimal</i>
char, wchar	char
string, wstring	java.lang.String
boolean	boolean
Octet	byte
any	org.omg.CORBA.Any
void	void

IDL	Java
module	Java package
interface	Java interface
attribute	Java methods getter/setter
operation	Java method
struct	Java class
enum	Java class
union	Java class
fixed	java.math.BigDecimal
array	Java array
sequence	Java array
const	static final (interface attribut)
exception	java.lang.Exception subclass



Java

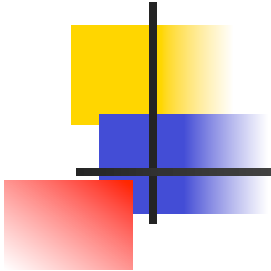
Syntaxe et Module

- identificateur OMG IDL
=> identificateur Java
- En cas de conflits ...
 - Mots clés réservés du langage Java
 - Identificateurs réservés pour la projection
 - Holder
 - Helper, ...
- L'identificateur Java est précédé d'un
`\_`
—

- module OMG IDL
=> package Java

```
// OMG IDL  
module Namespace {  
    ...  
};
```

```
// Java  
package Namespace {  
    ...  
};
```



Java

Structure et Enumération

- structure OMG IDL
=> classe Java

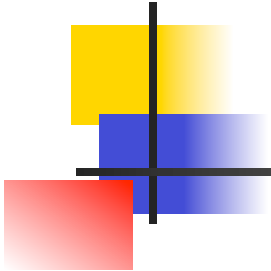
```
// OMG IDL
struct Point {
    long X, Y;
};

// Java
public final class Point {
    public int X;
    public int Y;
    public Point (int X, int Y) {...};
};
```

- enum OMG IDL
=> classe Java

```
// OMG IDL
enum Color {Red, Green, Blue};

// Java
public final class Color {
    public static final int _Red = 0;
    public static final Color Red = new
        Color(_Red);
    ...
    public static final int _Blue = 2;
    public static final Color Blue = new
        Color(_Blue);
    public int value () {...};
    public static Color from_int (int value) {...};
    protected Color (int) {...};
};
```



Java

Union et Redéfinition de type

- union OMG IDL
=> classe Java

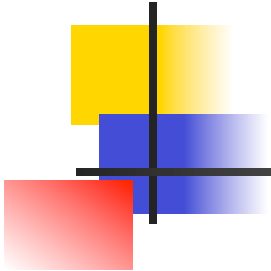
```
// OMG IDL
union Status (boolean) {
    case TRUE: boolean Bearded;
    case FALSE: long Children;
};

// Java
public final class Status {
    public Status() {...};
    public boolean discriminator () {...};
    public boolean Bearded () {...};
    public void Bearded (boolean v) {...};
    public long Children () {...};
    public void Children (long v) {...};
};
```

- typedef OMG IDL
=> remplacé par le type aliasé

```
// OMG IDL
typedef int Duration;
struct Animal {
    Duration Age;
};

// Java
public final class Animal {
    public int Age;
    public Animal (int Age) {...};
};
```



Java

Constante et Exception

- constante OMG IDL
=> public static final

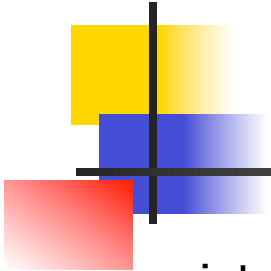
```
// OMG IDL
interface Math {
    const double Pi = 3.14;
};
module Math {
    const double Pi = 3.14;
};
```

```
// Java
public final class Math {
    public static final double Pi = 3.14;
};
public final class Pi {
    public static final double value = 3.14;
};
```

- exception OMG IDL
=> classe Java
- membre OMG IDL
=> classe Java (struct)

```
// OMG IDL
exception Error {
    long Code;
};
```

```
// Java
public final class Error extends
    org.omg.CORBA.UserException {
    public int Code;
    public Error(){...};
    public Error(int Code){...};
};
```



Java

Interface et Génération

- interface OMG IDL
=> interface Java

```
// OMG IDL
interface Stack {
    void Pop (out long v) raises Empty;
    void Push (in long v);
    long Top () raises Empty;
};

// Java
interface Stack {
    void Pop (IntHolder v) throws Empty{...};
    void Push (int v) {...};
    int Top () throws Empty {...};
};
```

- Pour chaque interface <i>
 - une interface <i>
 - Une interface pour les opérations <i>Operations
 - une classe de souche <i>Stub
 - une classe de squelette <i>POA (_<i>ImplBase)
 - une classe d'objet implémentation <i>Impl
 - une classe <i>Holder
 - une classe <i>Helper

Java

Construction imbriquée et Attribut

- Constructions imbriquées
=> dans un package Java

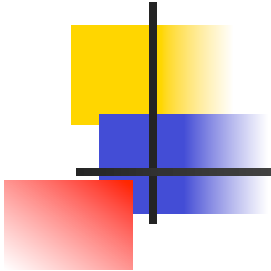
```
// OMG IDL
interface Stack {
    exception Empty;
    ...
};

// Java
package StackPackage;
public final class Empty extends
    org.omg.CORBA.UserException {};
```

- attribut OMG IDL
=> getter/setter Java

```
// OMG IDL
interface Account {
    readonly attribute long Balance;
    string attribute Name;
    ...
};

// Java
interface Account {
    int Balance () {...};
    void Name (string value) {...};
    string Name () {...};
    ....
};
```



Java

Opération et Holder

- opération OMG IDL
=> méthode Java
- Mais paramètre Java passé par valeur
...

// OMG IDL

```
long M (in T x, inout T y, out T z);
```

// Java

```
int M (T x, THolder y, THolder z) {...};
```

```
int x, a, b;
```

```
THolder y = new THolder(1024);
```

```
THolder z = new THolder();
```

```
O.M (x, y, z);
```

```
a = y.value;
```

```
b = z.value;
```

- Pour satisfaire les modes inout et out, tous les types disposent de Holder
 - Définis par omg.org.CORBA pour les types de base
 - Générés en plus des souches et squelettes pour les types de l'utilisateur

// Java

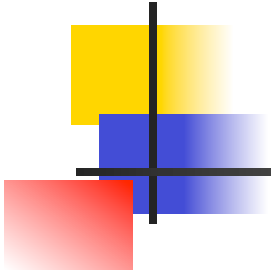
```
public final class TYPEHolder {
```

```
    public TYPE value;
```

```
    public TYPEHolder();
```

```
    public TYPEHolder(TYPE v){value =  
        v};
```

```
};
```



Java

Helper et Client

- Pour em/déballer une donnée vers un Any ou pour convertir vers un type, tous les types disposent de Helper
 - Définis par omg.org.CORBA pour les types de base
 - Générés en plus des souches et squelettes pour les types de l'utilisateur

// Java

```
public final class TYPEHelper {  
    public static void insert  
        (Any a; TYPE v) {...};  
    public static TYPE extract (Any a) {...};  
    ...  
    public TYPE narrow (Object ob) {...};  
};
```

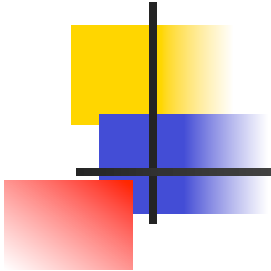
```
public class Client {  
    public static void main (String args) {  
        // Créer un orb  
        ORB orb = ORB.init (args, null);  
        // Retrouver la référence du client  
        CORBA.Object object  
            = orb.string_to_object (args[0]);  
        // Construction d'une référence objet  
        // typée  
        myInterface myObject =  
            myInterfaceHelper.narrow (Object);  
  
        // Invocation d'une méthode  
        myObject.myOperation();  
    };  
};
```

Java - côté serveur

Héritage et Délégation

```
public class Server {
    public static void main (String args) {
        // Créer un orb puis un adaptateur
        ORB orb = ORB.init (args, null);
        POA rootpoa = POAHelper.narrow
            (orb.resolve_initial_references
             ("RootPOA"));
        // Lance l'adaptateur
        rootpoa.the_POAManager().activate();
        // Créer un servent chez l'adaptateur
        myInterfaceImpl myImpl =
            new myInterfaceImpl ();
        org.omg.CORBA.Object myObject =
            rootpoa.servant_to_reference(myImpl);
        // ou directement auprès du rootpoa
        // org.omg.CORBA.Object o = myImpl._this(orb);
        // Affiche l'IOR pour les clients
        System.out.println
            (orb.object_to_string (myObject));
        // Lancer l'orb pour traiter les requêtes
        orb.run();
    };
};
```

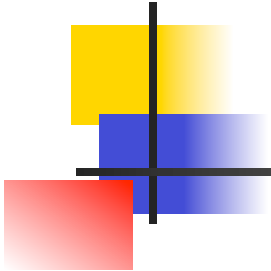
```
public class Server {
    public static void main (String args)
    {
        ORB orb = ORB.init (args, null);
        POA rootpoa = POAHelper.narrow
            (orb.resolve_initial_references
             ("RootPOA"));
        rootpoa.the_POAManager().activate();
        // Créer un objet implémentation
        myInterfaceImpl myImpl =
            new myInterfaceImpl;
        // Créer un servent relié à l'objet
        // implémentation et à l'adaptateur
        myInterfacePOATie tie =
            new myInterfacePOATie
                (myImpl, rootpoa);
        orb.run();
    };
};
```



Java : côté serveur

Objets CORBA et servants

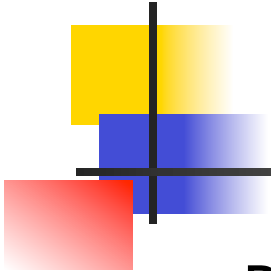
- Découplage entre la référence d'un Objet CORBA et le servant associé a cette référence
 - Le servant fournit les services décrits par l'Objet CORBA
- Un Serveur doit créer et exporter la référence d'un Objet CORBA afin de permettre aux clients d'accéder à cet Object CORBA
- Une référence doit indiquer l'Objet CORBA, qui doit être activé via une association à un servant (implémentation effective, en Java, C++, ...)



Java : côté serveur

l'activation d'Objets CORBA

- Deux actions sont nécessaires :
 - Instancier le servant (à associer avec l'Objet CORBA)
 - Ex: `HelloImpl myHelloServant = new HelloImpl();`
 - Inscrire le servant et l'Identifiant de l'Object CORBA auprès d'un POA (Portable Object Adapter)
 - Ceci active l'Objet CORBA de ce servant et le rend disponible aux clients



Java : côté serveur

l'enregistrement de servants

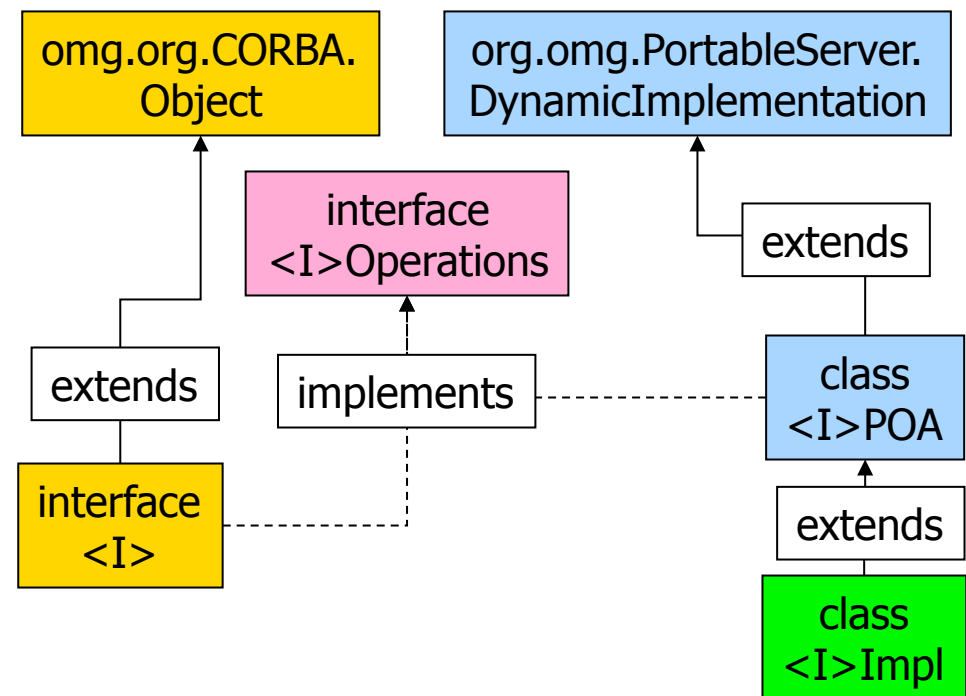
- Plusieurs façons possibles:
 - Appeler la méthode `_this()` sur le servant
 - Ex: `org.omg.CORBA.Object obj = myHelloServant._this(orb);`
 - Enregistre le servant avec le POA par défaut (le rootPOA), en lui associant un ID Objet unique
 - Crée et retourne l'Objet CORBA de ce servant
 - Appeler la méthode `servant_to_reference()` sur le POA ciblé
 - Ex: `org.omg.CORBA.Object obj =
rootPOA.servant_to_reference(myHelloServant);`
- NOTE: afin de convertir la référence de l'Objet CORBA vers une référence du type du servant il faut appeler la méthode `narrow()` de la classe **Helper** du type concerné:
 - Ex: `Hello myHello =>HelloHelper.narrow(obj);`

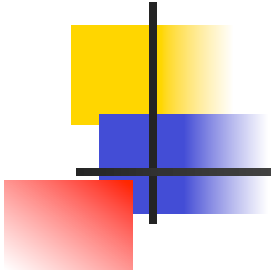
Java - Implémentation: Héritage

- L'objet implémentation hérite du squelette
- Dès lors il ne peut hériter d'une autre classe

```
import org.omg.CORBA.*;
import java.lang.*;

public class <I>Impl extends <I>POA
{
    public Op (...);
}
```

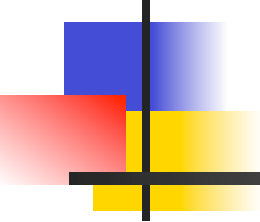




Java

Connexions avec la JVM

- Le standard Java inclut le support d'un ORB CORBA
 - Les mécanismes de configuration de la JVM, via l'introspection, permettent de modifier la bibliothèque d'implantation utilisée
- Mécanismes standard de configuration
 - Orb.properties -> fichier de configuration standard
 - Portabilité du code des stubs/skels
 - Implantation portable du cœur de l'intergiciel: org.omg.CORBA.portable, org.omg.CORBA.portable_2_3, and org.omg.PortableServer.portable
- Possibilité de spécifier lors de l'exécution du nœud la classe Java implantant l'ORB: org.omg.CORBA.ORBClass
- Note: le JDK de Sun dispose d'une implantation de CORBA, mais elle est incomplète et non standard, lui préférer une autre implantation, telle que JacORB.

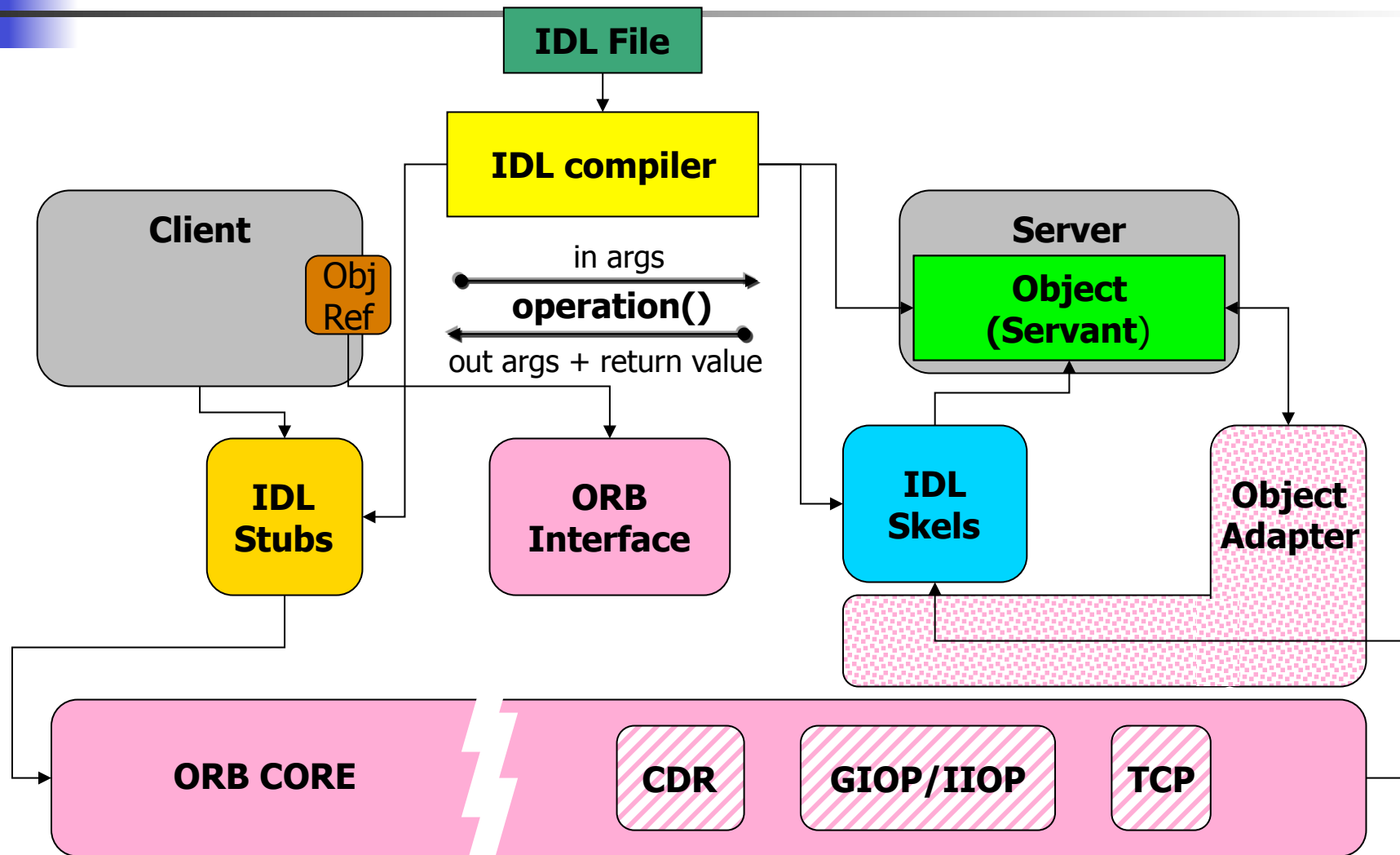


OMG CORBA

Cœur de l'ORB

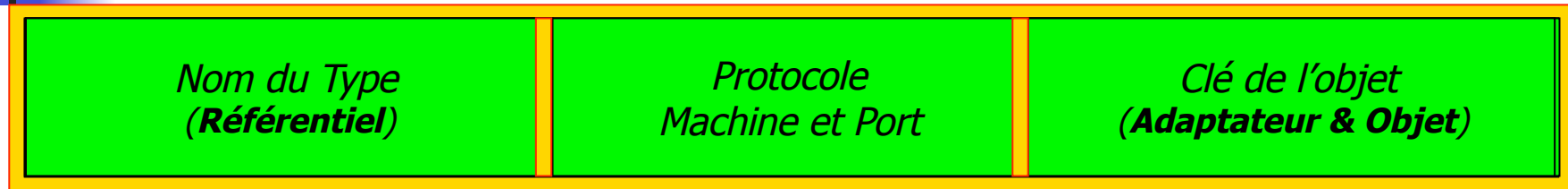
CORBA

Rôle de l'ORB



ORB

Interoperable Object Reference (IOR)



Référentiel **ex:** *"IDL:myModule::myInterfaceDef:1.0"*

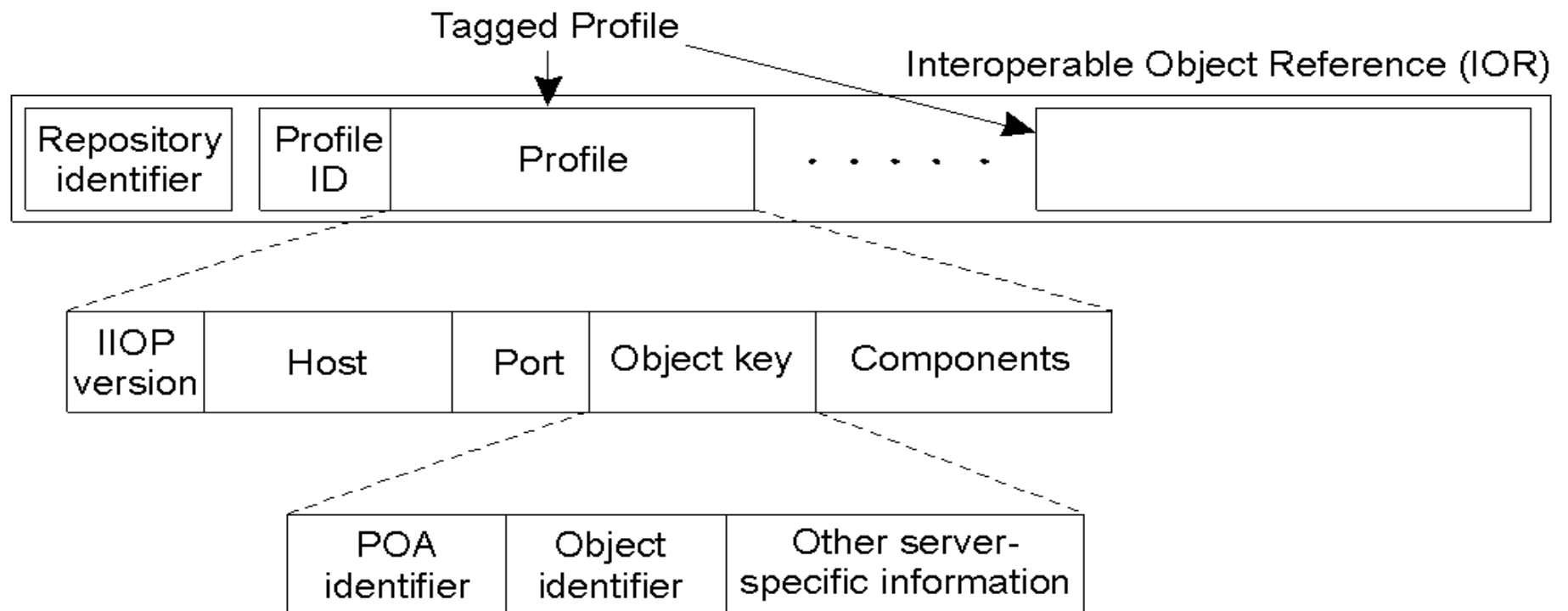
GIOP, adresse, port **ex:** *"IIOP v1.0","antigone.enst.fr", 5555*

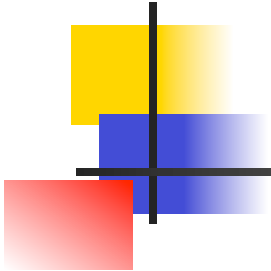
Chemin vers l'adaptateur objet et clé de l'objet

ex: *"OA007/OA009", "_obj"*

ORB

IOR: Profil, Référence, Clé





ORB (Java) – convertir l'IOR

- Afin d'obtenir la référence (IOR) d'un Objet CORBA, appelez la méthode

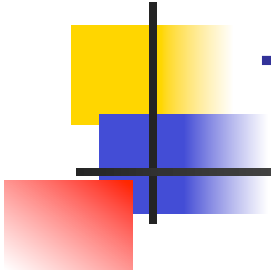
String object_to_string(Object obj)

- Ex: **String reference =
orb.object_to_string(obj);**

- Afin d'obtenir un Objet CORBA à partir de sa référence (IOR), appelez la méthode

Object string_to_object(String ior)

- Ex: **Object obj = orb.string_to_object(ior);**



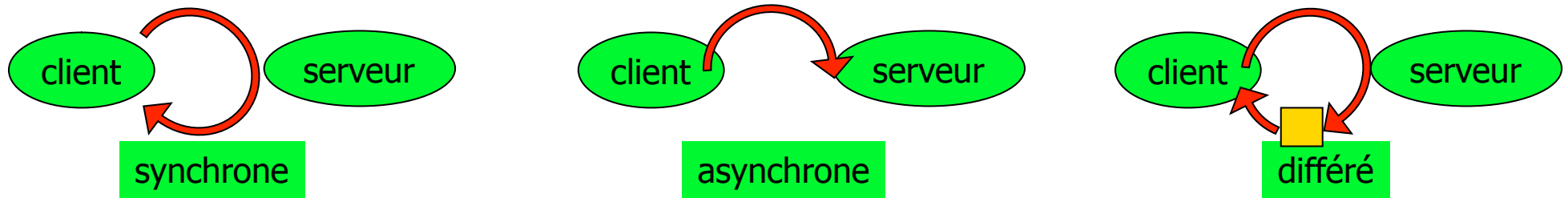
Transmission de la référence d'Objet

- Transmission directe de la référence en format String
 - Ex: si le client et le serveur partagent un système de fichiers
 - Le serveur écrit la référence dans un fichier
 - Le client lit le même fichier afin de récupérer la référence
- Utilisation d'un service de Nommage ou de Trading
 - Publication dans un endroit bien connu, sous un nom unique, lisible par l'utilisateur

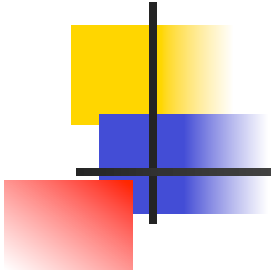
CORBA

Protocole de communication

- Modes d'invocation d'opérations



- Protocole d'invocation d'opérations : GIOP (General Inter-Orb Protocol)
 - Associé à un protocole de transport fiable
 - IIOP (Internet Inter-Orb Protocol) pour TCP/IP
 - Autres pour HTTP, RPC, ...
 - Associé à un format de représentation
 - CDR (Common Data Representation)
 - # de XDR (le récepteur s'adapte à l'émetteur)



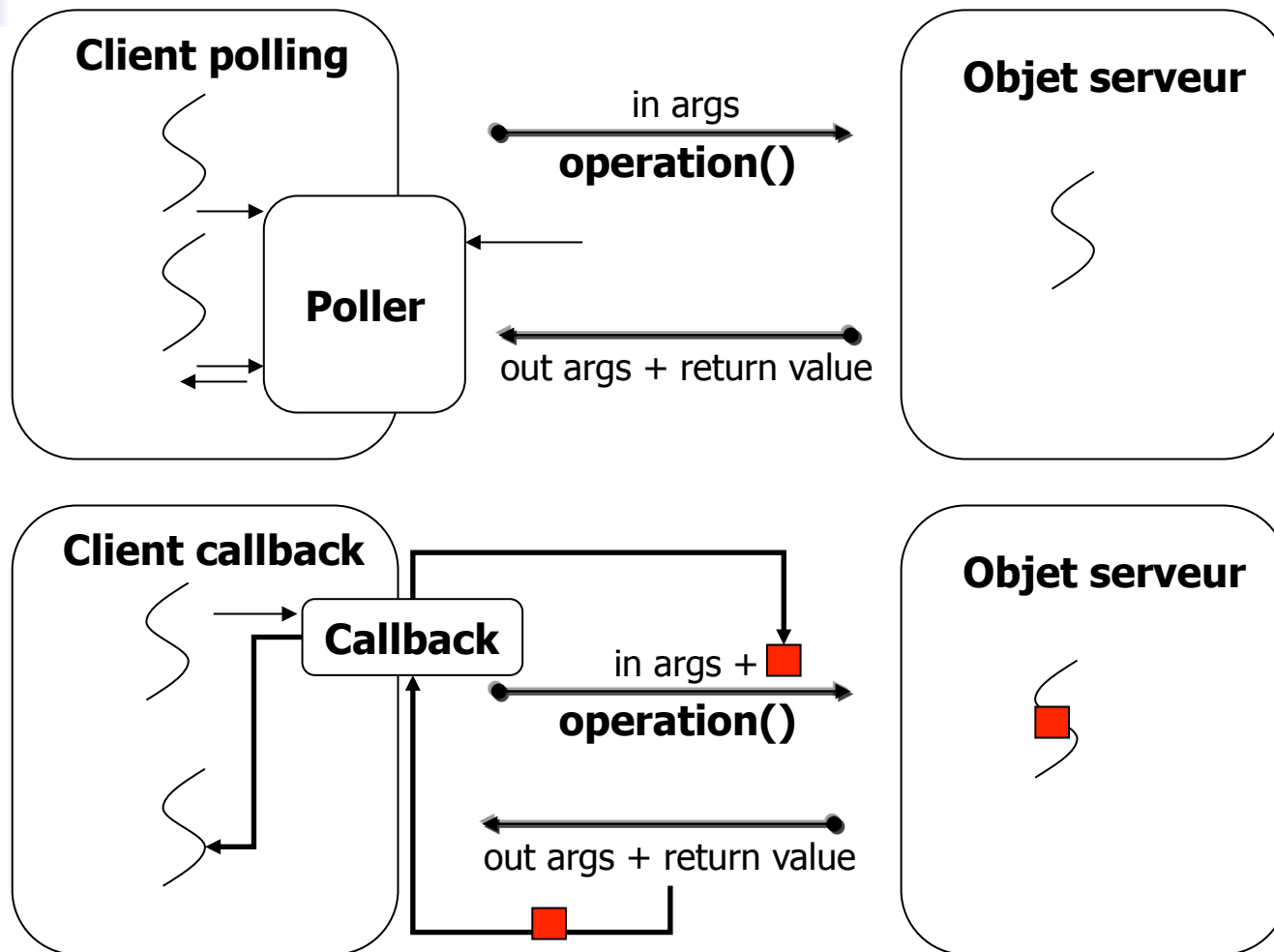
ORB

General Inter-ORB Protocol

- Protocole pour assurer les communications entre objets
 - Request: invocation d'une méthode (Client)
 - Reply: réponse à une invocation (Serveur)
 - CancelRequest: annulation d'une invocation (Client)
 - LocateRequest: localisation d'un objet (Client)
 - LocateReply: réponse de localisation (Serveur)
 - CloseConnection: fermeture de connexion (Serveur/Client)
 - MessageError: signalisation de message erroné (Serveur/Client)
 - Fragment: fragmentation de messages (Serveur/Client)
- GIOP (General Inter-ORB Protocol) se fonde plutôt sur un protocole de transport fiable, orienté connexion
 - **IIOP** (Internet IOP), implantation de GIOP au-dessus de **TCP/IP** (compulsory)
 - **MIOP** (Multicast IOP), implantation de GIOP au-dessus d'**UDP** (!)

AMI - Asynchronous Message Interface

Callback et Polling

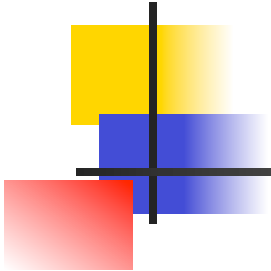




CDR - Common Data Representation

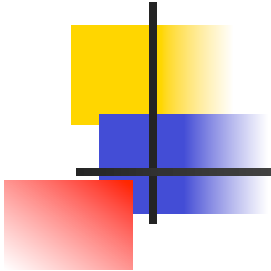
- GIOP spécifie la représentation des données entre env. hétérogènes au travers de CDR - **spécifie les règles de transformation entre les formats IDL et binaire**
 - L'émetteur utilise sa représentation par défaut
 - Le récepteur transforme les données si nécessaire
 - CDR diffère de XDR qui force la représentation big endian
- GIOP spécifie également l'alignement des données
 - Les structures complexes sont alignées sur les membres
 - Les chaînes de caractères (« strings ») contiennent la longueur plus la chaîne terminée par un caractère nul

Alignement [byte]	Dimension [byte]	Type
pas besoins	1	char, octet, boolean
2	2	(unsigned) short
4	4	(unsigned) long, float, enum
8	8	(unsigned) long long, double
8	16	long double



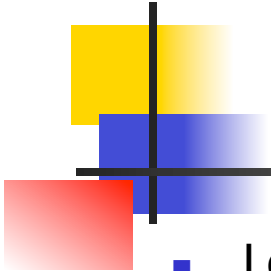
ORB - Adaptateurs d'Objets (Object Adapters – OA)

- Enregistre les objets d'implémentation (servants)
- Produit et interprète les références d'objets (CORBA Objects)
- Retrouve les objets en fonction des références
- Active et désactive les objets d'implémentation
- Aiguille les requêtes vers les objets d'implémentation
- Invoque les méthodes à travers les squelettes statiques ou dynamiques
- CORBA propose plusieurs adaptateurs qui diffèrent dans la manière de traiter les étapes précédentes
 - BOA ou le Basic Object Adapter avec 4 modes d'activation (déprécié)
 - POA ou Portable Object Adapter avec 7 étapes dans la chaîne d'exécution chacune disposant de 2, 3 ou 4 modes



ORB - Basic Object Adapter (BOA)

- Première définition d'un adaptateur d'objets et de ses politiques pour CORBA
 - Serveur partagé
 - Un processus pour tous les objets du serveur
 - Serveur non partagé
 - Un processus par objet du serveur
 - Serveur par méthode
 - Un processus par méthode
 - Serveur persistant
 - Serveur partagé qui ne se charge pas de la création et de la terminaison des processus
- Le BOA est maintenant « deprecated »

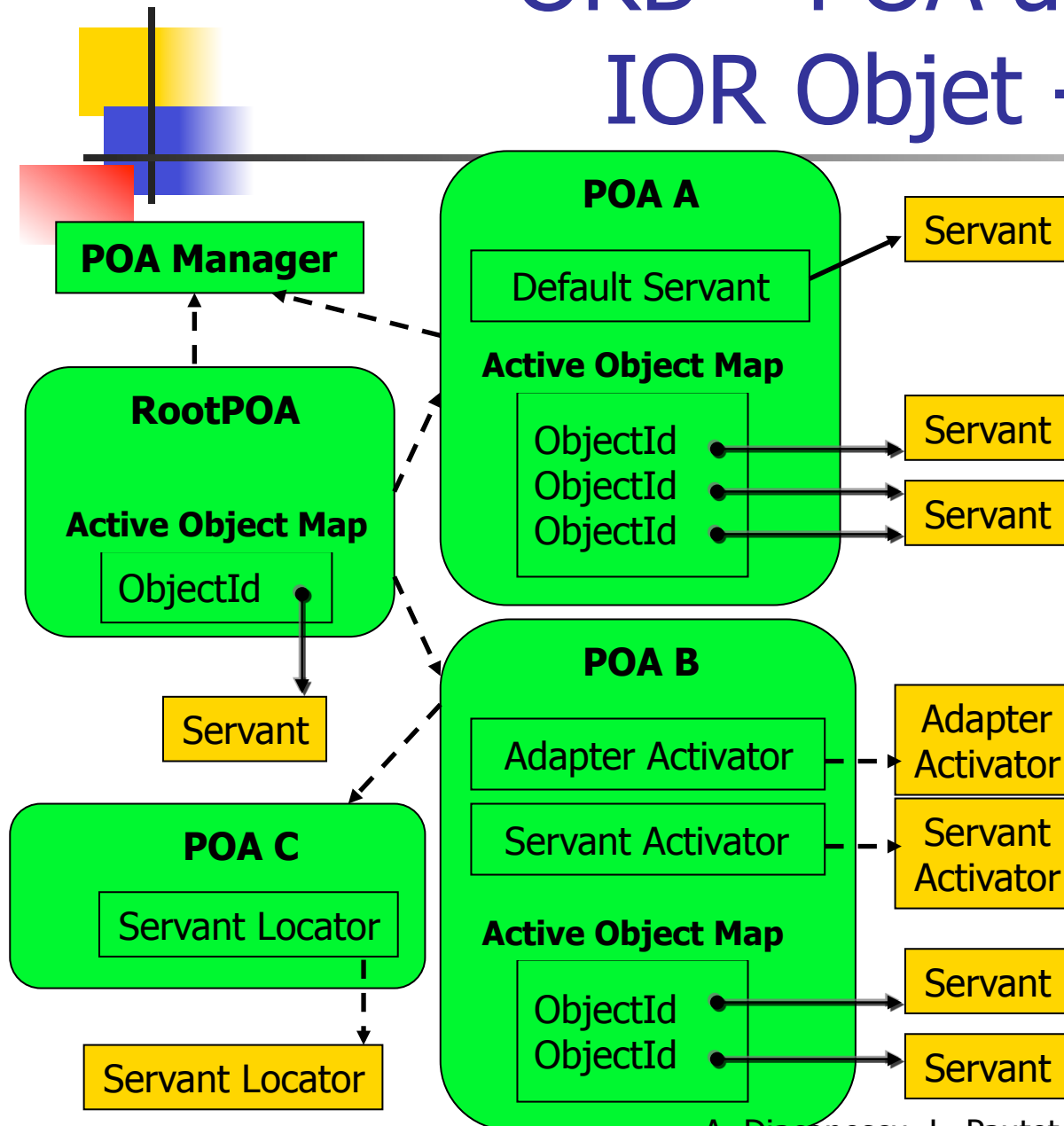


ORB - Portable Object Adaptor (POA)

- Le POA enrichit et corrige les parties mal spécifiées du BOA
- Un ORB peut accueillir plusieurs POAs, organisés de façon hiérarchique:
 - Chaque ORB dispose d'un POA racine (rootPOA)
 - Un POA est un objet composé contenant objets et POAs
- La clé de l'objet d'une référence d'objet (IOR) contient:
 - Les identificateurs des POA imbriqués qui mènent à l'objet
 - L'identificateur d'objet relatif au POA qui le contient
- Le BOA propose une association statique : lors de la création d'une IOR, le servant est instancié en fonction de l'objet
- Le POA propose une association dynamique : la création de l'IOR et l'instanciation du servant sont dé-corrélées.
- L'association de l'objet avec le servant peut se faire à chaque requête (via Servant Locator) ou non (via Servant Activator et l'Active Object Mapping - AOM).

ORB – POA associations

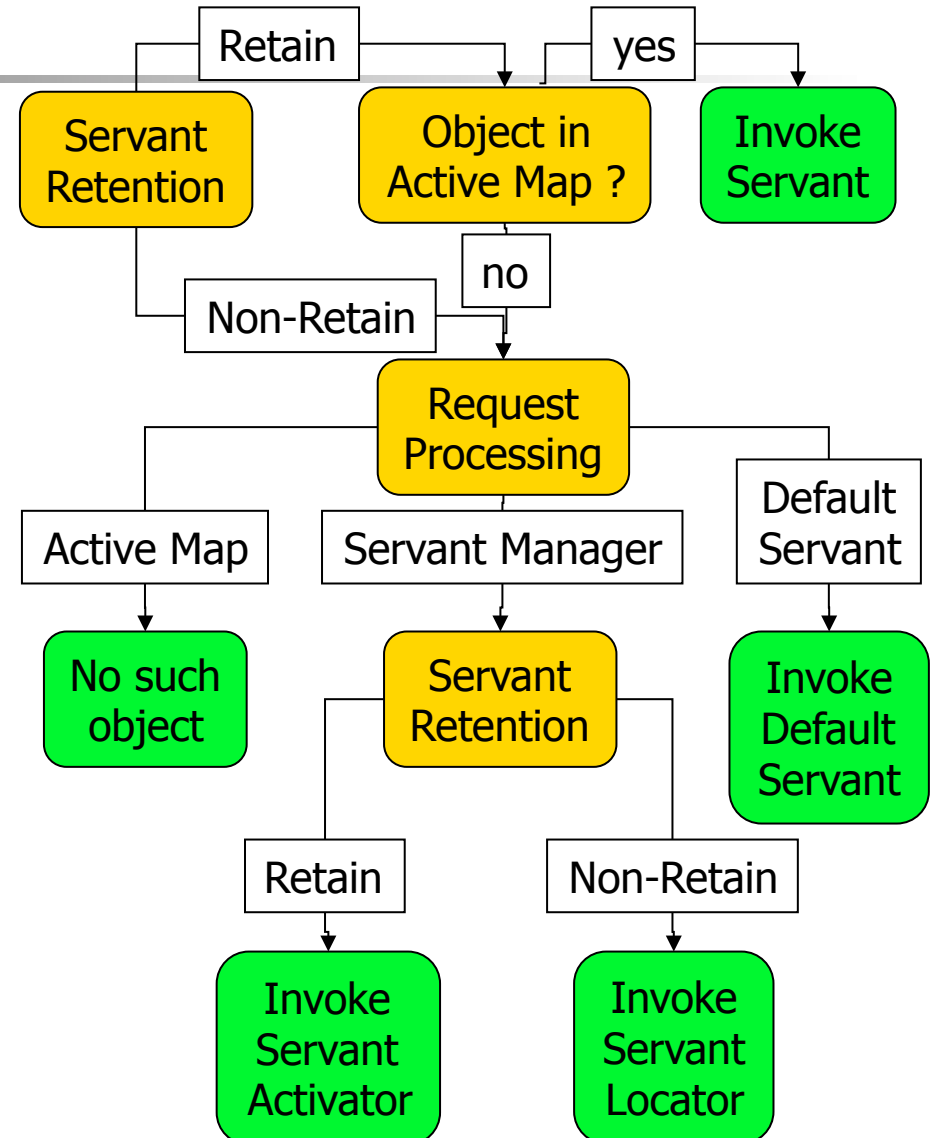
IOR Objet - servant



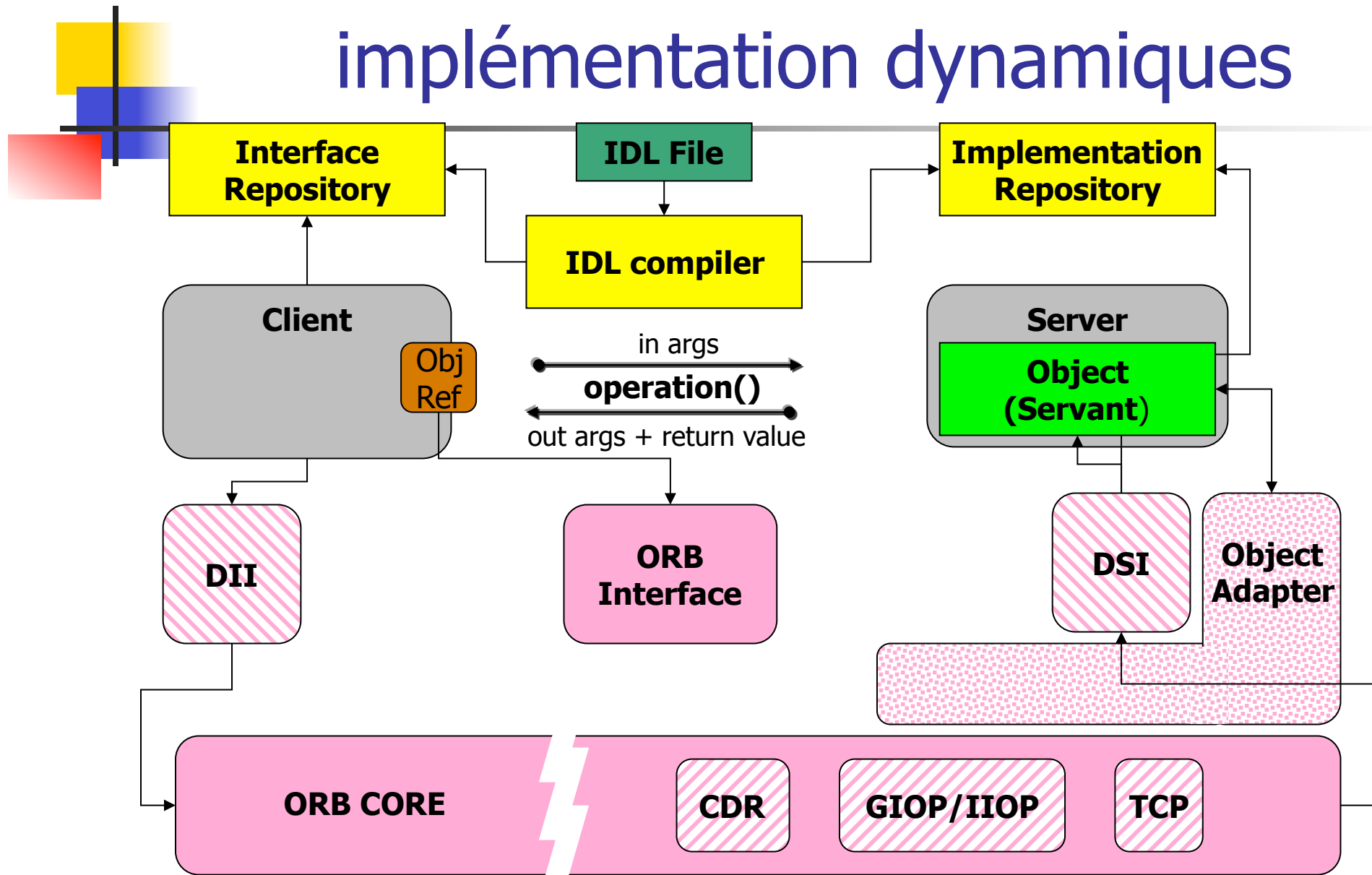
- POA Manager contrôle l'exécution des POAs
- Adapter Activator crée les POAs absents
- Default Servant si actif traite les requêtes
- Servant Manager gère les Servants
 - Servant Activator les crée (~ lazy loading)
 - Servant Locator les trouve (~ cash)

ORB - politiques du POA

- Thread Policy
 - Création et nombre des processus exécutant les requêtes
- Lifespan Policy
 - Durée de vie de l'objet en fonction du POA et du serveur (Transient ou Persistent)
- Object Id Uniqueness Policy
 - Association d'un servent avec un ou plusieurs objets
- Id Assignment Policy
 - Création de l'identifiant d'objet
- Servant Retention Policy
 - Association servent-objet dans une table (Active Object Map - AOM)
- Request Processing Policy
 - Création des servents en cas d'absence
- Implicit Activation Policy
 - Activation implicite ou non du servent lors de l'enregistrement de l'objet



ORB : invocation et implémentation dynamiques



Interface d'Invocation Dynamique

- DII : Dynamic Interface Invocation
 - Invocation de méthodes sur des objets dont on ne connaît pas l'interface au moment de la compilation
 - Utilisation d'un référentiel d'interfaces (IFR) pour découvrir les informations relative aux interfaces IDL
 - Recherche et interprétation de l'interface par le référentiel
 - Construction d'une requête
 - Spécification de l'objet distant et de l'opération
 - Ajout des paramètres et envoi de la requête
- Interfaces stockées dans le référentiel d'interfaces (IFR)
 - base de données des descriptions d'interfaces des objets serveurs
 - organisé de façon hiérarchique (ensemble d'objets dont la structure reflète celle des interfaces qui y sont stockées)
 - chaque entité stockée possède un identifiant unique

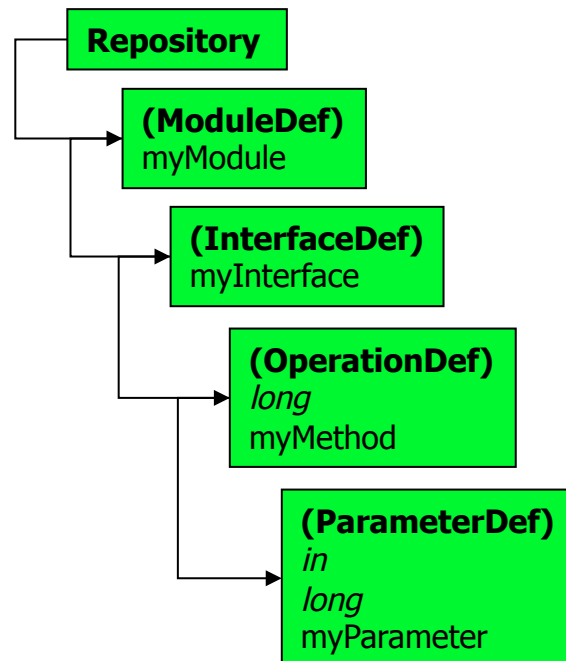
ORB

Interface de Squelette Dynamique

- DSI : Dynamic Skeleton Interface
 - Intégration d'un squelette générique pour les objets serveurs dont on ne connaît pas l'interface au moment de la compilation
 - Equivalent côté serveur de la DII
- Idée du fonctionnement
 - Implante une classe et une méthode qui aiguille les appels entrants
 - Interprète les requêtes et leurs paramètres
 - Travaille sur des objets request identiques à ceux de la DII
- Inconvénients
 - Difficiles à manipuler
 - Pas de vérification
 - Plus lent à l'exécution
- Avantages
 - Plus flexible
 - Ponts génériques entre ORB
 - Intégration d'applications existantes non-CORBA

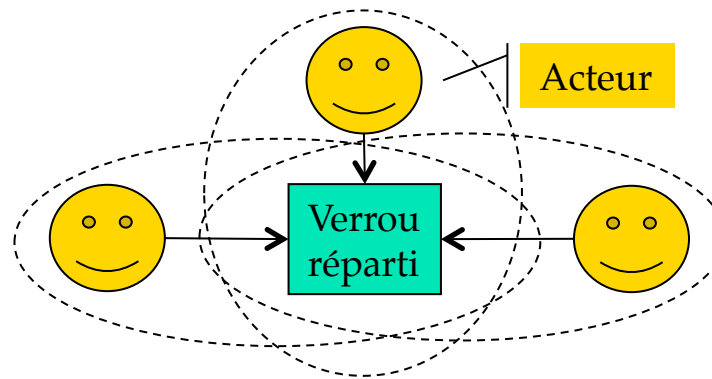
ORB - invocation dynamique sur un exemple

```
module myModule {  
  interface myInterface {  
    long myMethod  
    (in long myParameter);  
  };  
};
```



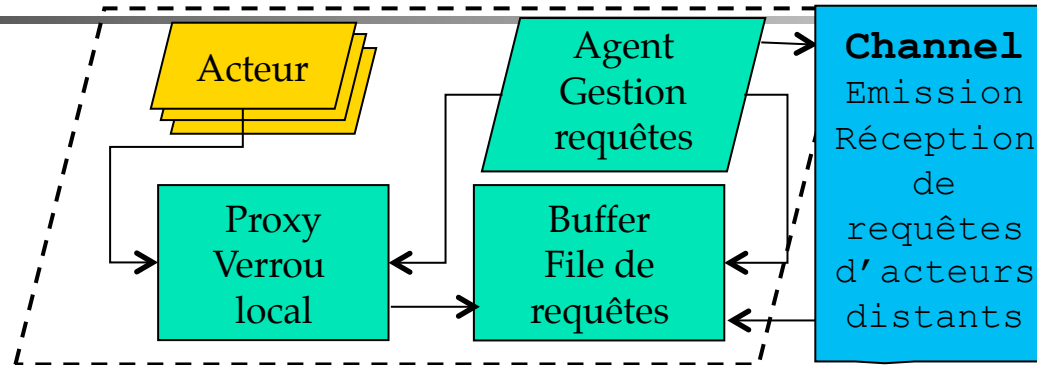
```
Object o = ...  
// Récupère un objet  
InterfaceDef i = o._get_interface();  
// Explore la description de l'interface i  
Request r = o._request ("myMethod");  
// Crée une requête pour l'invocation  
r.set_return_type(tk_long);  
Any setVal = r.add_in_arg()  
setVal.insert_long(999);  
// Remplit la liste des paramètres  
r.invoke();  
// Invoke la requête et bloque ...  
int l = r.return_value().extract_long();  
// Extrait les paramètres de retour  
// des valeurs de retour de la requête
```

Retour sur le cas d'étude

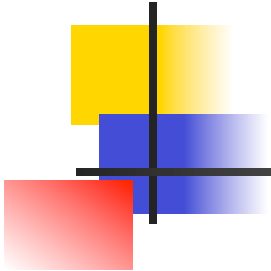


Etude de cas (Java CORBA)

Modélisation des Acteurs



- Chaque acteur est modélisé par un processus léger et se trouve regroupé avec proxy et agent sur un même processus lourd
- Chaque agent A_n modélisé par un processus léger dispose d'un tampon circulaire T_n de requêtes
- A l'aide du servant propre au nœud n , Channel reçoit les requêtes émises vers le nœud et les dépose dans le tampon T_n
- L'agent A_n un processus cyclique lit les requêtes de T_n , interagit avec le proxy P_n et peut envoyer des requêtes au travers de Channel en utilisant CORBA

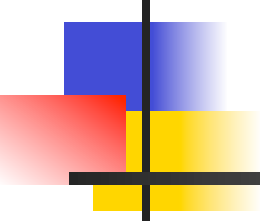


Etude de cas (Java CORBA)

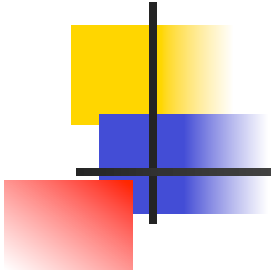
Réutilisation de la version Java threads

- Grace à l'architecture adoptée, nous pouvons reprendre les codes du Proxy et de l'Agent de la version Java threads
- Comme pour la version Java sockets, **seule l'implantation de Channel doit être revue dans le cas de la version Java CORBA**
- Channel crée un servant CORBA pour recevoir les requêtes des autres nœuds et les déposer dans le tampon de l'agent
- Channel.initialize stocke dans un fichier l'IOR du servant
- Il charge les IORs de tous les nœuds pour produire des stubs
- Après l'opération Channel.initialize, le sous-programme principale Main peut créer Actor, Proxy et Agent
- La dernière opération Channel.activate démarre l'exécution de l'ORB et donc la boucle d'événements bloquante ORB.run

OMG CORBA



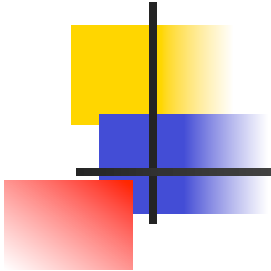
Common Object Services (COS)



CORBA COS

Common Object Services (COS)

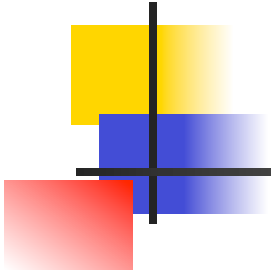
- Serveur de noms, cycle de vie, évènements
- Transactions, concurrence, externalisation, relations
- Sécurité, serveur de temps
- Persistance de l'état
- Propriétés, licences, serveur de requête
- Annuaire par fonctionnalité, collection, gestionnaire de versions



COS Naming

- Le service de nommage fournit
 - La **notion d'association** entre nom et référence d'objet
 - La **notion de contexte** contenant associations et contextes
 - Les **méthodes de manipulation**
 - d'association
 - de contexte

- Le service de nommage s'organise autour
 - D'une hiérarchie comme celle des fichiers Unix
 - Association = Fichier (référence d'objet = vers données)
 - Contexte = Répertoire
 - De méthodes de
 - Création et destruction d'association et de contexte
 - Exploration et découverte d'association et de contexte

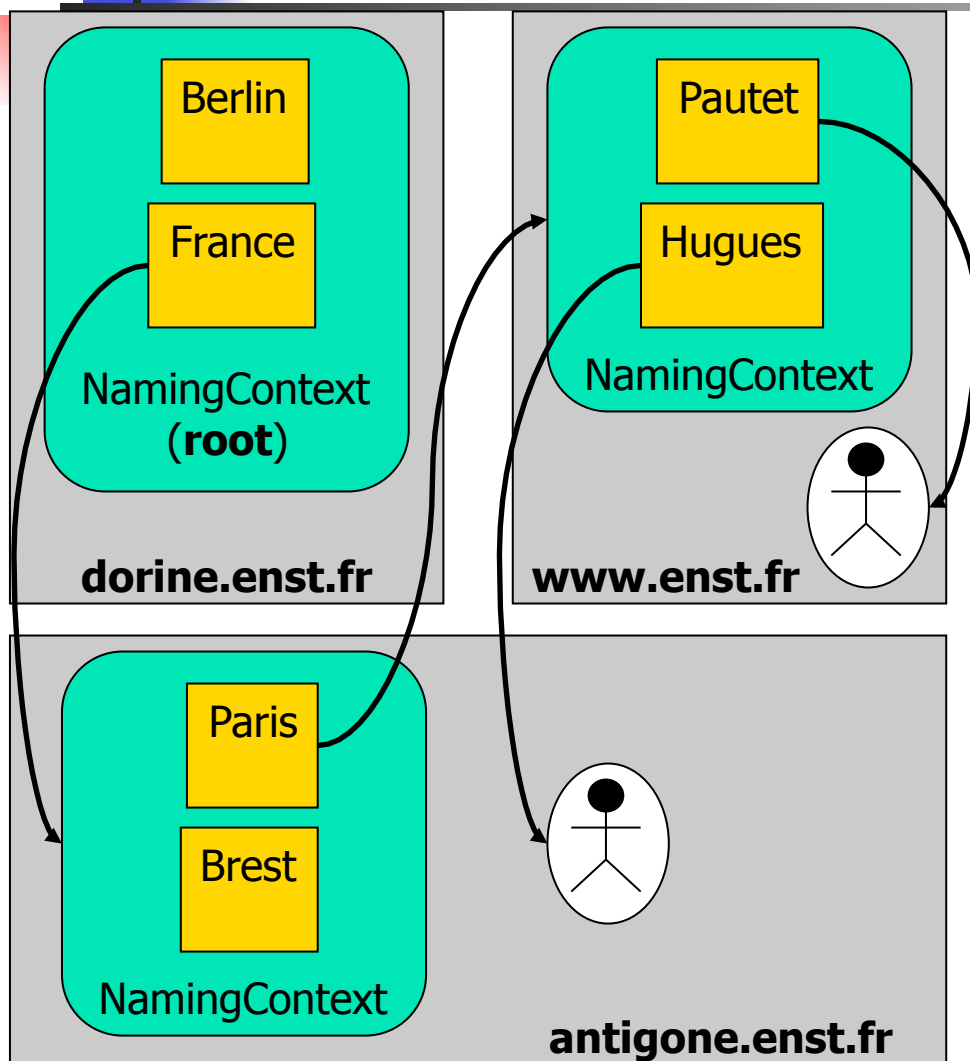


COS Naming – NamingContext

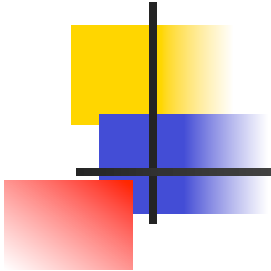
- Administration
 - Gestion d'une hiérarchie de NamingContext
- Utilisation
 - Export et import de références d'Objets CORBA

```
interface NamingContext {  
  
    void bind (in Name n, in Object obj)  
        raises (AlreadyBound, ...);  
    void bind_context  
        (in Name n, in NamingContext nc)  
        raises (AlreadyBound, ...);  
  
    ...  
    NamingContext new_context ();  
    NamingContext bind_new_context(in Name n)  
        raises (AlreadyBound, ...);  
  
    ...  
    Object resolve (in Name n)  
        raises (NotFound, ...);  
  
    ...  
    void list  
        (in unsigned long how_many,  
         out BindingList bl,  
         out BindingIterator bi );  
  
};
```

COS Naming - Administration



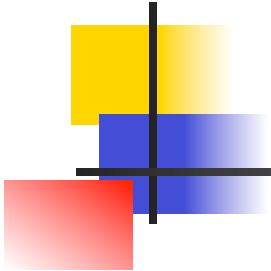
- L'administrateur utilise l'API de la classe `NamingContext` afin de créer, modifier ou effacer une hiérarchie de nommage
- Methodes:
`bind_context`,
`rebind_context`
...



COS Naming - Utilisation (1)

- Côté serveur :
 - Exporter les références Objet vers le Service de Nommage
 - Méthodes `bind` ou `rebind`
 - Ex: `ns.bind( myHelloName, myHelloObject );`

- Côté client :
 - Importer les références Objet depuis le Service de Nommage
 - Méthode `resolve`
 - Ex: `org.omg.CORBA.Object obj =
 ns.resolve(myHelloName);`



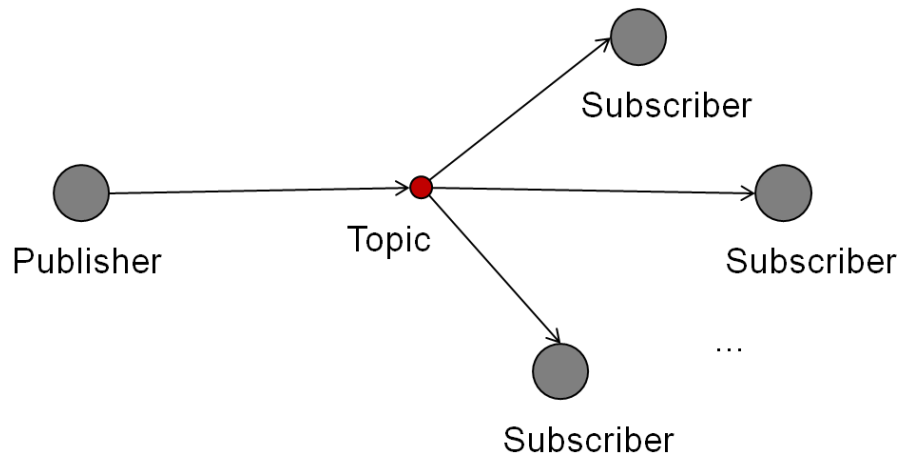
COS Naming - Utilisation (2)

- Comment obtenir une référence vers le Service de Nommage?
- Le service de nommage est un Objet CORBA comme tous les autres => appelez la méthode **resolve_initial_references** sur l'interface de l'ORB
 - `org.omg.CORBA.Object nsObject =
orb.resolve_initial_references(
"NameService");`
 - `NamingContext nc =
NamingContextHelper.narrow(nsObject);`

CORBA

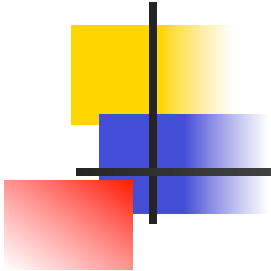
Les services Publish/Subscribe

- Modele de communication asynchrone, 1-n (one-to-many)
- Les services Publish/Subscribe de CORBA:
 - Event Service
 - Notification Service (étend l'Event service)
 - Telecom Log service (étend le Notification Service)



La terminologie CORBA :

- Publisher → Supplier
- Subscriber → Consumer
- Topic → Event Channel



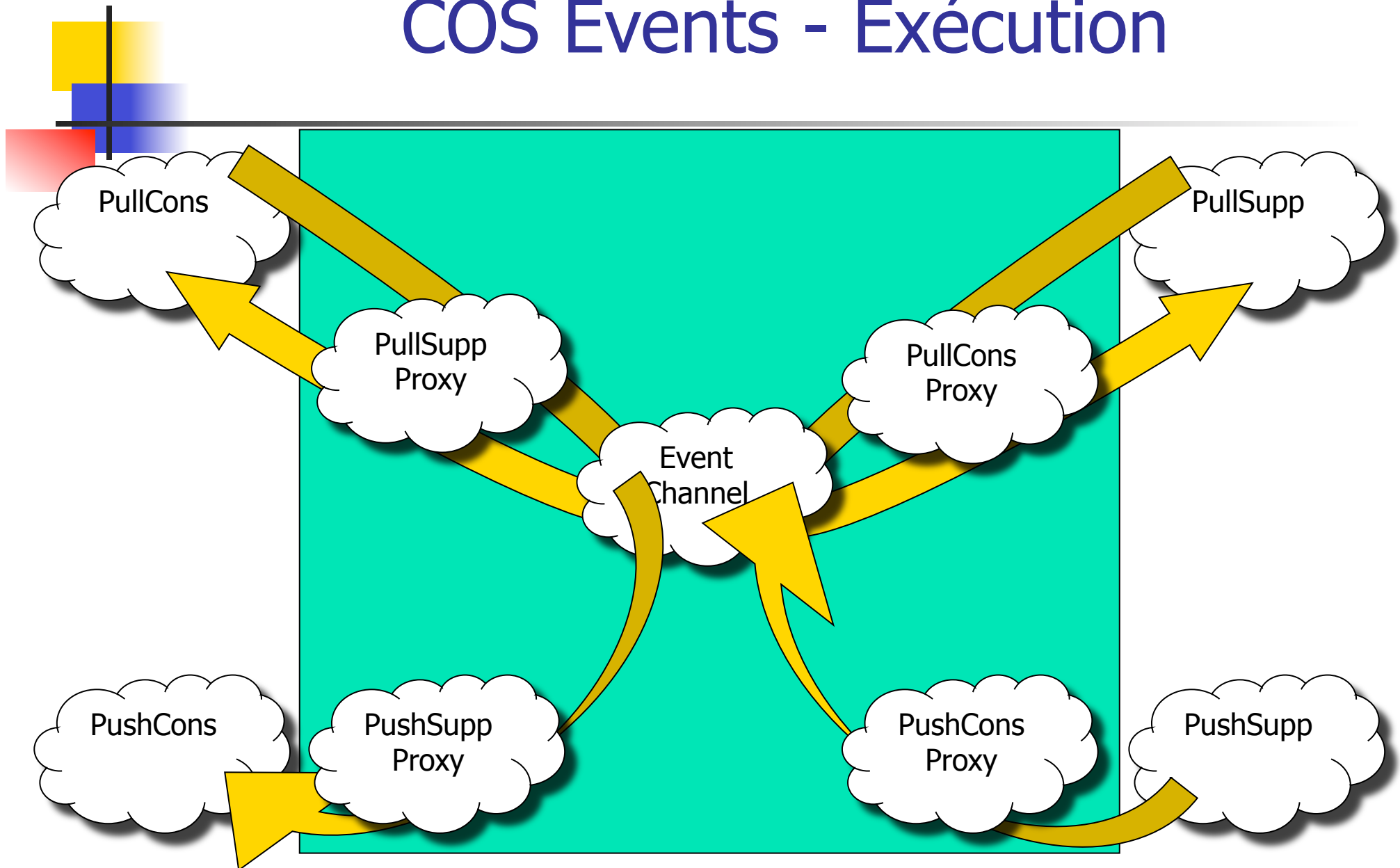
COS Events - Principes

- Le service d'évènements permet de s'abonner à des services 1-N de publications / souscriptions
 - Producteur d'évènements
 - Consommateur d'évènements
- ... selon deux modes ...
 - Push
 - Pull
- ... ce qui permet de découpler les interactions entre objets
 - Le client et le serveur ne se connaissent pas
 - Le client et le serveur ne sont pas actifs simultanément

COS Events - Organisation



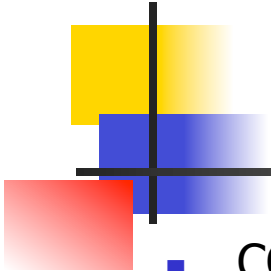
COS Events - Exécution



COS Events - Rôles

Rôle	Action	Description
PushSupplier <i>actif</i>	Invoque la méthode push du proxy	Le producteur transmet par la méthode push un événement au canal par le proxy
PullSupplier <i>passif</i>	Fournit la méthode pull pour le proxy	Le proxy attend que le producteur émette un événement et publie celui-ci dans le canal
PushConsumer <i>passif</i>	Fournit la méthode push pour le proxy	Le proxy invoque la méthode push du consommateur à l'arrivée d'un événement
PullConsumer <i>actif</i>	Invoque la méthode pull du proxy	Le proxy débloque l'appel à pull du consommateur à l'arrivée d'un événement

Consumer	Supplier	
	Push	Pull
Push	Notifier	Agent
Pull	Queue	Procurer



Conclusions et perspective

- CORBA répond à de nombreux besoins
 - Simplifie la réalisation d'une application répartie
 - Offre interopérabilité de langages et de systèmes
 - S'appuie sur de nombreuses technologies
- ... s'accompagne de nombreux inconvénients
 - Induit une forte complexité (multiples spécifications)
 - Induit une courbe d'apprentissage élevée
- ... et continue de s'enrichir (... de grossir 😊)
 - CORBA 3.0 et son modèle de composant ...
- ... mais se voit concurrencer par des solutions moins coûteuses
 - Message Oriented Middleware (JMS)
 - Intergiciels dépendants de la technologie
+ Web services pour l'interopérabilité...
 - ICE ? (Internet Communication Engine - www.zeroc.com)