# Intergiciels

Introduction

Appels de procédures à distance (RPCs)
- RPC : présentation/motivation
- Transparence de la localisation : principe du mandataire
- Mise en œuvre des RPCs
- Etude de cas : les RPC de Sun

Objets répartis
- Bilan sur les RPCs
- Objets répartis : principes
- Etude de cas : Java RMI



## Sources, références, compléments

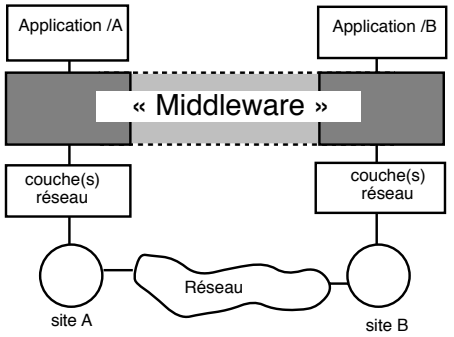
- Cours de systèmes répartis de Sacha Krakowiak (UJF), en ligne
- Cours de Gérard Padiou (N7), en ligne, et sur l'intranet du département informatique
  http://www.depinfo.enseeiht.fr/cours.html
- Ouvrage général sur les intergiciels de Sacha Krakowiak, (en anglais)
  http://sardes.inrialpes.fr/~krakowia/MW-Book/
- Ouvrage : Corba des concepts à la pratique (J-M Geib, Ph. Gransart, Ph. Merle)
- Le site www de l'ouvrage précédent
  http://corbaweb.lifl.fr/CORBA_des_concepts_a_la_pratique/index.html
- Le site de l'OMG : http://www.omg.org/
- Le site du W3C : http://www.w3.org
- Le site Java de Sun : http://java.sun.com, avec le tutoriel sur la projection CORBA/Java de Sun
  http://java.sun.com/docs/books/tutorial/idl/index.html



## 1 – Introduction

***Objectif : rendre la répartition transparente pour le programmeur***

→ simuler un environnement (virtuellement) centralisé
à partir d'un environnement (physiquement) réparti




→ gérer
- l'hétérogénéité des représentations
- les pannes
- la localisation
- l'homogénéité des accès

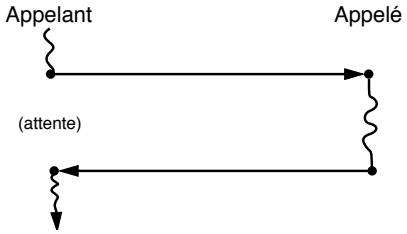


## 2 – Appels de procédures à distance (RPCs : Remote Procedure Calls)

### *1) Principe/motivation*

*But* : intégrer l'interaction C/S aux langages de programmation

*Idée* : similitude du flot de contrôle entre
- l'interaction C/S
- l'appel procédural




→ possibilité d'intégrer l'appel de services distants aux langages de programmation de manière
- naturelle
- transparente

appel de service ≈ appel de procédure

## Mise en œuvre de la transparence : mandataires

***Objectif :***

- le service distant doit être vu/appelé comme un service local
- le serveur doit pouvoir traiter des requêtes provenant d'environnement divers

***Difficultés (~ C/S)***

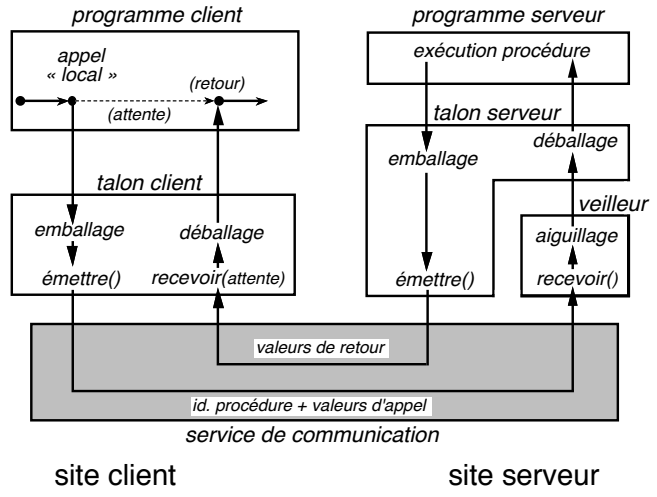
- hétérogénéité des communicants
- pannes
- masquage (± complet) de la localisation

***Solution***

- introduire sur le site client un service local (*mandataire*), appelé *talon* (*stub*) dans la terminologie des RPC, qui :
  - ◊ présente la même interface que le service distant (transparence d'accès)
  - ◊ assure la communication, la gestion des pannes et de l'hétérogénéité vis-à-vis du serveur
- de même, côté serveur, un talon assure la liaison avec les clients.



## 2)Protocole de principe






## Limites à la transparence

***Espaces d'adressage séparés***

- Avantage : encapsulation des données du serveur
  - ◊ sécurité
  - ◊ conception
- Difficulté : passage de paramètres au moyen de messages
  - ◊ passage par valeurs (copie) : OK
  - ◊ passage par références : KO (espaces d'adressage distincts)
    Remèdes :
    - *interdire* le passage de références
    - simuler le passage de références par *copie/restauration* des valeurs référencées (envoi d'une copie des données référencées, puis recopie des valeurs de retour)

      Difficulté possible : *aliasing* (passage de références distinctes à une même donnée)

      → *programmation explicite de l'emballage/déballage* des paramètres
    - utiliser des *références (globales) opaques* (cookies)
      * mémoire virtuelle répartie
      * service de nommage global



***Espaces de contrôle séparés***

**Erreurs possibles au cours d'un RPC :**

- défaillance du médium/du serveur. Protocoles associés aux sémantiques :
  - ◊ au plus une fois (*at most once*)
  - ◊ une fois et une seule (*exactly once*)
  - ◊ au moins une fois (*at least once*)
- erreurs en cours d'exécution du service appelé
  - ◊ soit zéro, soit une fois → mécanisme d'atomicité (→ cf transactions)
  - ◊ service exactement une fois → mécanisme de fiabilité : redondance, mémoire stable

→ signaler l'erreur au client (≈ lever une exception)

**Comment ?**

≈ envoyer un message/paramètre de retour du RPC

**Interprétation par le client**

Selon les possibilités du langage/du service de RPC :

- Service minimal (cf UNIX) : message d'erreur traduit par un *code retour* conventionnel, accompagné éventuellement d'un code d'erreur.
- Service évolué : *mécanisme d'exception intégré* au langage, ou offert par le service de RPC
  → message d'erreur traduit par la levée d'une exception au niveau du programme client

## 3) Désignation et liaison

Gestion de la correspondance : nom symbolique (externe) ↔ nom interne (identifiant, adresse réseau)

Cas des RPC :

- noms de services ↔ ports et adresses serveurs
- noms d'opérations d'un service ↔ références code sur le serveur correspondant

***Instant(s) d'évaluation de la liaison***

**Liaison statique** (précoce) :

localisation du serveur fixée au moment de la compilation du programme client

→ localisation fixée a priori par le programmeur, ou appel d'un serveur de nom à la compilation

**Liaison dynamique** (tardive) : localisation non fixée à la compilation

⇒ désignation symbolique des services (non liée à un site d'exécution)

→ possibilité d'implémentation ou de sélection retardée

→ possibilité de s'adapter à une reconfiguration du système : régulation, pannes, évolutions

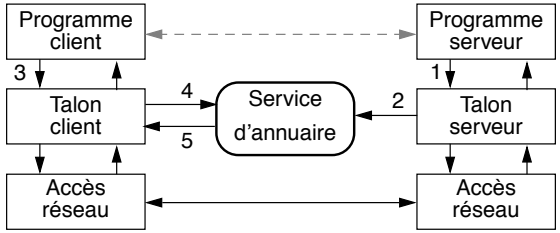
***Remarque***

La liaison dynamique peut évaluer la localisation du service :

- au premier appel seulement
- à chaque appel



***Mise en œuvre de la liaison dynamique : serveur d'annuaire (ou serveur de noms)***




**Protocole**

- `1,2` : le service est enregistré auprès de l'annuaire sous : `<nom, adr. serveur, n° port>`
- `3,4,5` : le client consulte l'annuaire pour trouver `<adr. serveur, n° port>` à partir de `<nom>`
- L'appel peut alors avoir lieu
- Schémas plus élaborés : attributs (critères de choix)

**Tolérance aux pannes (service critique)**

- recours à une mémoire stable
- duplication des tables, des serveurs

**Localisation du serveur de noms par le client**

- diffusion de la requête par le client, ou
- chargement à l'exécution (variable d'environnement fournie par le système), ou
- utilisation d'une adresse conventionnelle

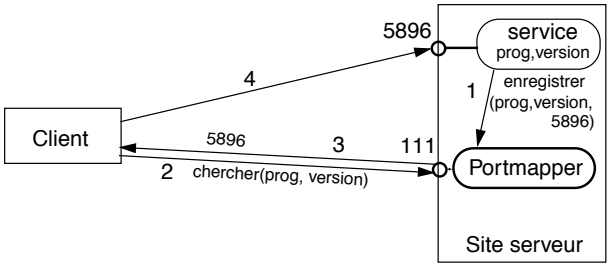


**« Serveur de noms » local à un site : *portmapper***

Cas (fréquent) où :

- le site hébergeant le service est connu,
- mais où le port correspondant au service n'est pas connu

→ utiliser un service de nommage local au serveur, le ***portmapper*****,** sans passer par un annuaire global



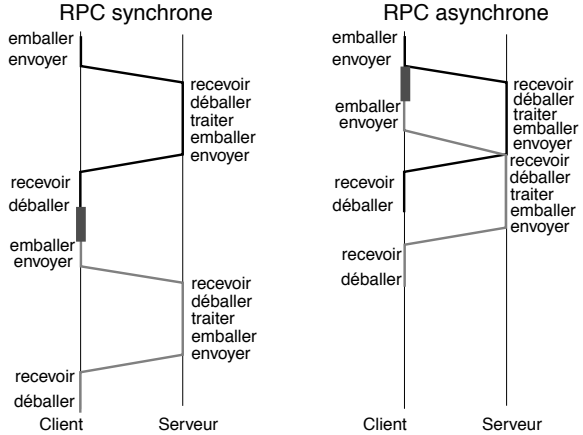

- Le portmapper a un n° de port fixé par convention (111)
- Un service enregistre le numéro de port de son veilleur auprès du portmapper
- Le veilleur se met en attente sur ce port



## 4) Extensions des RPC

***RPC asynchrones : réponses (et requêtes) différées***

- pas d'attente dans le cas où une réponse n'est pas attendue
- possibilité de grouper les requêtes avant envoi (utile si aucune réponse n'est attendue)
- découplage clients/serveurs : un client peut faire traiter plusieurs requêtes en parallèle




Exemple : X11 (requêtes d'affichage

***Appels de méthodes à distance (RMI : Remote Method Invocation)***

- Extension du RPC aux langages/environnements à objets.
- Caractéristiques :
  - ◊ intégration au langage : les services sont définis et utilisés comme des objets standard.
  - ◊ un objet a un état rémanent, conservé indépendamment de l'exécution des requêtes. Cet état peut être modifié par et influer sur le traitement des requêtes : deux appels identiques d'une même méthode sur un même objet n'auront pas forcément même effet.
- Le modèle objet facilite en outre l'importation de fonctionnalités/services (héritage)
  → « plateformes » réparties (objets système: CORBA, DCOM, ou objets langage : JAVA, ...) proposant, sous forme d'objets héritables (« mixins ») des services non fonctionnels, traitant les problèmes propres à la répartition : pannes, sécurité, MOM, courtage de ressources…
- Service critique : gestion d'un espace de noms vaste et évolutif

***C/S à composants***

*But* : permettre la construction d'applications

- par intégration de composants préexistants
- sans connaissance préalable sur le code des composants utilisés (boîte noire)
  → définition de protocoles « réflexifs », d'« introspection », permettant à un client de découvrir et/ou adapter dynamiquement un service nouveau ou inconnu a priori.

*Exemples* : Java Beans (graphique), OLE/COM/DCOM/.Net (applications intégrées), OpenDoc (documents composites), Enterprise Java Beans et CORBA CCM (applications réparties).



# *5) Mise en œuvre des RPCs*

## Architecture d'un serveur

Organisation et choix classiques du modèle C/S :

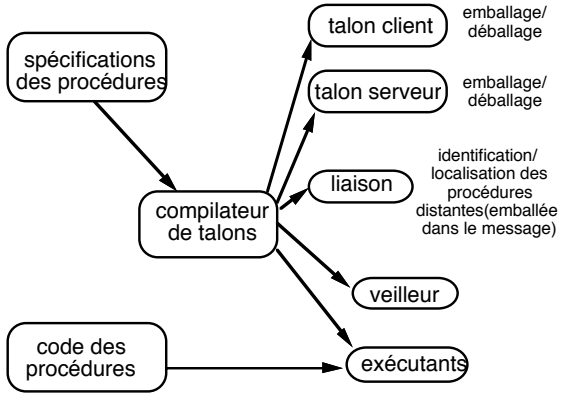
- veilleur + exécutants
- création des exécutants statique ou dynamique
- communication en mode
  - ◊ connecté (interactions longues et fiabilité requise)
  - ◊ ou datagramme (interactions courtes) ; généralement associé à l'absence d'exécutants
- serveur mono ou multi-activités
  si multiactivités
  - exécutants polyvalents
  - ou spécialisés par service, par requête ou par client.



## Compilateur de talons

Spécification de procédure appelable à distance

- nom de la procédure
- type et direction des paramètres





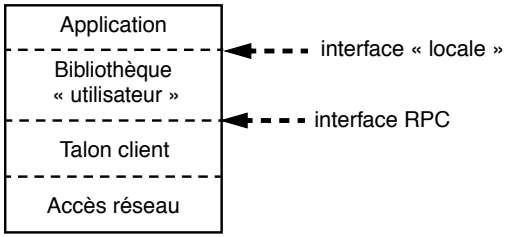

## (Non) transparence des RPCs

***Spécification des procédures appelables à distance***

- au moyen d'un langage spécifique (IDL : Interface Definition Language) (CORBA, RPC Sun)
  → indépendance vis-à-vis du langage
- ou bien, au sein d'un langage de programmation (Argus, Java (RMI))
  → économie de concepts, et intégration au langage

***Utilisation des RPC***

- utilisation explicite d'une bibliothèque de RPC
- ou bien RPC enveloppés dans une bibliothèque locale (préparant et réalisant l'appel distant)




*Remarque* : même principe/motivation que les bibliothèques d'E/S langage
(possibilité d'introduire des optimisations au niveau de la bibliothèque « utilisateur » : caches…)

# 6)Etude de cas : les RPC de Sun

## Langage de définition des interfaces (RPCL)

***Déclaration et liaison d'un service***

*Exemple* : service d'affichage de message à distance

```
/* message.x: protocole d'impression de messages à distance */
program MESSAGEPROG {
     version PRINTMESSAGEVERS {
        int PRINTMESSAGE(string) = 1;
   } = 5;
} = 0x20000001;
```

Une procédure à distance est identifiée de manière unique *sur une machine* par un triplet :

- numéro de programme (`0x20000001`),
  qui identifie un service, c.-à-d. un ensemble de procédures reliées « logiquement ».
- numéro de version (`5`) : un programme peut être disponible en plusieurs *versions*
  (en particulier plusieurs implantations).
- numéro de procédure (`1`), qui identifie chaque procédure dans la version du programme.

***Remarques***

- `/etc/rpc` est une liste des numéros de programmes utilisés
- `rpcinfo` fournit des informations sur les services RPC disponibles sur le réseau.



## Serveur de liaison

- Un serveur, appelé *rpcbind* (ou *portmapper* ) assure la correspondance entre
  - ◊ le triplet <n° de programme, n° de version, n° de procédure>
  - ◊ et l'adresse du serveur (sur IP, <adr machine, n° de port>).
- Quand un serveur RPC démarre, il indique au serveur de liaison l'adresse à laquelle il écoute (procédure `svc_reg`).
- Quand un client veut utiliser une *procédure* d'un *programme* de *version* donnée,
  il obtient l'adresse du serveur en contactant le serveur de liaison de la machine concernée.



## Définition des types : XDR

```
/* Service d'accès à un fichier distant, fichier AccFich.x */
const MAX = 1000;
typedef int IdFich;
typedef int PFich;
typedef int Lg;

struct Donnees {
  int lg;
  char tampon[MAX];
};

struct argsecr {
  IdFich f;
  PFich position;
  Donnees texte;
};

struct argslec {
  IdFich f;
  PFich position;
  Lg lg;
};

program ACCFICH {
  version VERSION {
      void ECRIRE(argsecr)=1
      Donnees LIRE(argslec)=2;
  }=4;
} = 8976;
```



## Processus de production de l'application (rpcgen)

A partir de l'interface RPCL (suffixe `.x`), le *compilateur de talons* `rpcgen` crée :

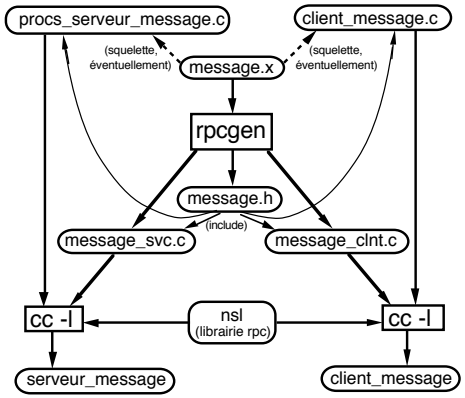
- *le source du talon client* (radical du fichier RPCL, suffixé par `_clnt.c`). Ce programme fournit l'interface locale du service distant, interface qui sera appelée par le programme client
- *le source du talon serveur* (radical du fichier RPCL, suffixé par `_svc.c`). Ce programme assure l'aiguillage vers les procédures définies dans le programme serveur, ainsi que l'emballage et le déballage des paramètres.
- un fichier de définitions ("*headers* ", suffixe `.h`), contenant notamment des macro-définitions (directive `#define`) pour les numéros de version, de programme, et de procédures, utilisées dans les autres modules.
- le source des *procédures de conversion* depuis et vers XDR (suffixe `_xdr.c`). Ce source n'est créé que si des types XDR sont définis dans le fichier RPCL. Il est aussi possible de programmer ses propres routines de conversion.
- optionnellement, `rpcgen` peut créer un *squelette* pour les procédures du service et/ou pour le programme client (options `-Ss` et `-Sc`).

Typiquement, lorsque `rpcgen` est utilisé, le programmeur doit uniquement écrire :

- la description du service RPCL
- le code du client
- le code des procedures serveur

## Exemple : affichage de messages à distance.

Service défini dans le fichier `message.x` vu précédemment.




### *commandes invoquées*

- `rpcgen message.x`
- `cc client_message.c message_clnt.c -o client_message -lnsl`
- `cc procs_serveur_message.c message_svc.c -o serveur_message -lnsl`
- `serveur_message` : lance le serveur (par exemple sur la machine brams)
- `client_message brams "bonjour"` : affiche bonjour sur la machine du serveur (brams)



### *Remarques*

- `x.message` ne contenant aucune définition de type, `rpcgen` ne crée pas `message_xdr.c`
- `rpcgen` peut créer un squelette pour le client, et/ou pour le serveur (options `-Sc` et `-Ss`). Ces squelettes contiennent les directives `#include` nécessaires, ainsi que:
  - ◊ pour la partie serveur, les déclarations
    - des paramètres des procédures du service,
    - des procédures du service, avec un corps vide et le nom approprié (nom de la procédure RPCL en minuscules, suffixé par `_n`, où n est le numéro de la version)
  - ◊ pour le code du client, l'appel
    - aux procédures de la librairie RPC (`nsl`) permettant la liaison avec le serveur, et la gestion des erreurs (`cln_create, clnt_destroy, clnt_perror...`)
    - l'appel aux procédures du service, sous le nom approprié, ainsi que la déclaration des paramètres correspondants.
- enfin, diverses options permettent de contrôler la forme des paramètres, la mise en œuvre du serveur sous forme d'activités (threads)…



### *programme centralisé* *(Adapté de : ONC+ Developer's Guide, Sun)*

```
/* message.c : affiche un message au terminal */
 #include <stdio.h>
 main(argc, argv)
 int argc; char *argv[];
 {char *message;
  if (argc != 2) {
   fprintf(stderr, "usage: %s <message>\n",argv[0]); exit(1);
  }
  message = argv[1];
  if (!printmessage(message)) {
   fprintf(stderr,"%s: echec affichage\n",argv[0]); exit(1);
  }
  printf("Message affiche!\n"); exit(0);
 }

 /* Affiche un message sur le terminal. Le booléen renvoyé indique le succès */
 printmessage(msg)
 char *msg;
 {FILE *f;
  f = fopen("/dev/tty", "w");
  if (f == (FILE *)NULL) {return (0);}
  fprintf(f, "%s\n", msg);
  fclose(f);
  return(1);
 }
```



### *programme client* *(Adapté de : ONC+ Developer's Guide, Sun)*

```
/* client_message.c:version répartie (client) de message.c */
#include <stdio.h>
#include "message.h" /*message.h cree par rpcgen*/
main(argc, argv)
  int argc; char *argv[];
 {CLIENT *clnt; int *result; char *server; char *message;

  if (argc != 3) {fprintf(stderr,"usage:%s host message\n",argv[0]; exit(1);}
  server = argv[1]; message = argv[2];
/*Créer le handle client utilisé pour appeler le serveur argv[1]*/
  clnt = clnt_create(server, MESSAGEPROG,PRINTMESSAGEVERS,"visible");
                                /*"visible": protocole de transport "générique"*/
  if (clnt == (CLIENT *)NULL) { /* Echec liaison. Signaler et terminer */
     clnt_pcreateerror(server); exit(1); }
/* Appel de la procédure distante "printmessage" */
  result = printmessage_1(&message, clnt);
  if (result == (int *)NULL) {/* Echec appel. Signaler et terminer */
     clnt_perror(clnt, server); exit(1); }
/* Appel reussi*/
  if (*result == 0) {/*Echec affichage par le serveur. Signaler, terminer*/
     fprintf(stderr,"%s: echec affichage\n",argv[0]); exit(1);}
/* Affichage reussi */
  printf("Message affiche surt %s\n",server);
  clnt_destroy(clnt);
  exit(0);}
```

### *procédure serveur* (Adapté de : ONC+ Developer's Guide, Sun)

```
/* procs_serveur_message.c : implementation de "printmessage" */
#include <stdio.h>
#include "message.h" /*message.h cree par rpcgen*/

int * printmessage_1(msg, req)
  char **msg;
  struct svc_req *req;  /* details de l'appel */
 {static int result; /* doit survivre à l'appel ! */
  FILE *f;

  f = fopen("/dev/tty", "w");
  if (f == (FILE *)NULL) {
   result = 0;
   return (&result);
  }
  fprintf(f, "%s\n", *msg);
  fclose(f);
  result = 1;
  return (&result);
 }
```

***Remarque***

Il peut *parfois* être utile (efficacité, adaptation...) de retoucher le code produit par `rpcgen` ou de programmer directement avec la librairie RPC, qui fournit des modalités d'interaction plus fines : authentification, diffusion d'un RPC, RPCs asynchrones, rappel des clients ("callbacks")…



## Eléments de mise en œuvre

***Choix du protocole de transport***

Datagrammes (UDP/IP) si

- les procédures sont idempotentes
- la taille des arguments et des résultats est plus petite que la taille d'un paquet du protocole de transport (UDP : 8Ko)
- le serveur doit accepter beaucoup de clients simultanés

Circuit virtuel (TCP/IP) si

- les procédures ne sont pas idempotentes
- l'application nécessite une communication fiable
- les arguments ou résultats ne tiennent pas dans un paquet

***Remarque : incidence sur la sémantique des RPCs***

Communication fiable (TCP) : au plus une fois

Communication non fiable (UDP)

- Classique :au plus une fois
- Moderne : au moins une fois, jusqu'à un délai de garde… → aucune garantie (maybe)



### *Procédures RPC de haut niveau*

**Côté client**

```
enum clnt_stat rpc_call (const char *host,
                  u_long prognum, u_long versnum, u_long procnum,
                  xdrproc_t inproc, char *in,
                  xdrproc_t outproc, char *out,
                  const char *nettype);
```

Appel d'une procédure <prognum,versnum,procnum> sur la machine `host`.

La procédure reçoit en paramètre `in` codé avec `inproc` et retourne `out` décodé avec `outproc`.

```
enum clnt_stat rpc_broadcast (u_long prognum, u_long versnum, u_long procnum,
            xdrproc_t inproc, char *in, xdrproc_t outproc, char *out,
            bool_t eachresult (char *resp, ...),
            const char *nettype);
```

Diffusion de l'appel à <prognum,versnum,procnum>.

Pour chaque réponse, `eachresult` est appelé avec le résultat reçu.

**Côté serveur**

```
bool_t rpc_reg (u_long prognum, u_long versnum, u_long procnum,
                  char * (*procname) (char *arg),
                  xdrproc_t inproc, xdrproc_t outproc,
                  const char *nettype);
```

Enregistre `procname` sous le nom <prognum,versnum,procnum>. Le paramètre est décodé avec `inproc`. Le résultat est encodé avec `outproc`. Le désenregistrement peut être réalisé par appel à :

```
void svc_unreg (u_long prognum, u_long versnum);
```

`void svc_run(void)`: code du veilleur/aiguilleur.



### *procédures RPC de niveau intermédiaire*

```
CLIENT * clnt_create (const char *host,
                  u_long prognum, u_long versnum,
                  char *nettype);

bool_t clnt_control (CLIENT *clnt,
                     u_int req, char *info);
```

Création et contrôle d'un *handle* client vers un programme donné. Si `nettype` correspond à un protocole connecté, `clnt_create` établit la connexion. `clnt_control` permet de changer les délais de garde, de manipuler le numéro de version…

```
enum clnt_stat clnt_call (CLIENT *clnt,
                    u_long procnum,
                    xdrproc_t inproc, char *in,
                    xdrproc_t outproc, char *out,
                    struct timeval tout);
```

`rpc_call`, avec délai de garde avant abandon.

```
int svc_create(void(*dispatch)(struct svc_req *,…),
                  u_long prognum, u_long versnum,
                  char *nettype);
```

Enregistrement d'un programme. `dispatch` est la procédure d'aiguillage.

# 3 – Objets répartis

## 1) Bilan sur les RPCs

***Apport***

- Transparence (désignation, localisation)
- Intégration au modèle de programmation classique (appel procédural ↔ interaction C/S )
- Adéquation et concordance avec les démarches de conception modulaire
- Mise en œuvre facilitée pour le programmeur (compilation automatique des talons)

***Limitations***

- Caractère monolithique du processus de développement traditionnel :
  un utilisateur donné n'utilise qu'une faible fraction des fonctions d'un progiciel
- Conséquence : l'application est construite est déployée de manière statique
  - ◊ lourdeur d'évolution, d'adaptation au niveau fonctionnel
  - ◊ peu d'adaptation possible à l'exécution : équilibrage de charge, tolérance aux pannes…

  → inadapté à la diversité et variabilité des environnements d'exécution matériels et logiciels
- Structure d'exécution asymétrique, centralisée sur le serveur
  - ◊ point de contention et de défaillance unique
  - ◊ lourdeur de gestion des données propres à chaque client : état d'une session, transaction
  - ◊ → inadapté au passage à l'échelle, et à un fonctionnement ouvert
- Peu de services extrafonctionnels
  supervision, équilibrage de charge, tolérance aux pannes, transactions, données rémanentes…



## 2) Objets répartis : principes

Recherche de possibilités de développement plus incrémentales, adaptées et équilibrées

**Intergiciels à messages (Message Oriented Middleware)**

souple, ouvert, services extrafonctionnels, mais interactions et conception de bas niveau

**Objets répartis (ex : Java RMI, CORBA)**

- introduction de l'abstraction apportée par la modularité dans les langages à objets
- structure d'exécution abstraite et équilibrée (application = ensemble de pairs, en interaction)
  - ◊ chaque objet gère les instances de données qu'il encapsule
  - ◊ le mécanisme d'instanciation d'objets d'une même classe permet de gérer simplement et systématiquement les données propres à chaque interaction.
- les mécanismes des langages à objets (héritage, polymorphisme) facilitent l'adaptation et l'évolution lors de la conception, du déploiement, et de l'exécution
- combinaison avec la liaison dynamique et la mobilité pour faciliter l'adaptation à l'exécution
  - ◊ l'accès aux objets distants via des références globales permet de déterminer l'instance adaptée à chaque utilisation
  - ◊ le chargement dynamique de code distant fournit un moyen réparti de contrôler et adapter à la volée le code exécuté par l'application

**Composants répartis (ex : EJB, Corba CCM, .Net, Web Services)**

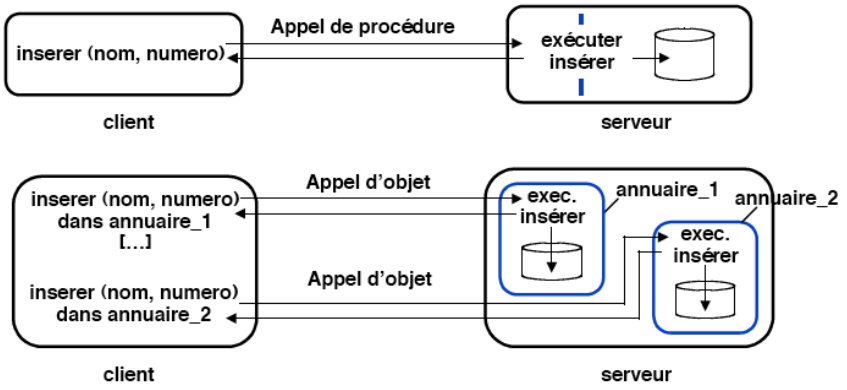
- conception à gros grain:applications définies par assemblage de *composants* logiciels (objets)
- faciliter la prise en compte des aspects extrafonctionnels (transactions, rémanence…)



***Le code reste partagé, mais les données sont spécifiques à chaque client***

Chaque client interagit avec une instance spécifique (un objet) de la classe réalisant une fonction

- les méthodes d'une classe sont utilisées par tous les clients de l'application (≈ RPC)
- les données utilisées par une méthode sont spécifique à chaque instance (objet) (≠ RPC)l




→ chaque client doit obtenir et désigner des *instances* (indépendammment des classes)



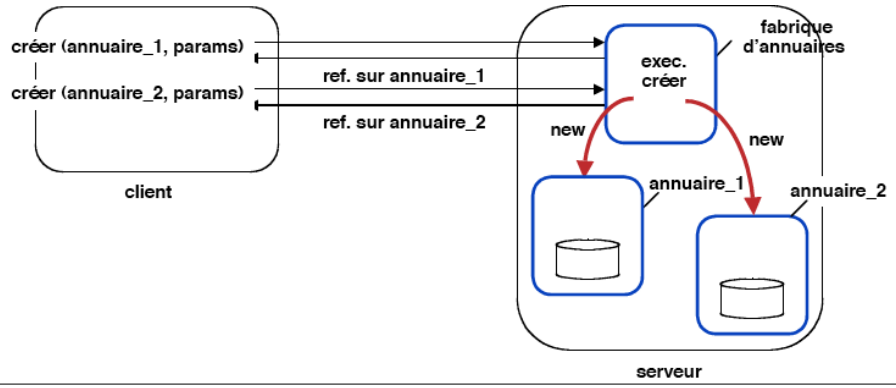
***Désignation des objets : notion de référence d'objet***

Contient ce qui est nécessaire pour atteindre l'objet distant

- Localisation de l'objet
  - ◊ Localisation réseau : machine, port d'accès utilisé pour les appels de méthodes distant
  - ◊ Localisation interne au serveur : référence mémoire de la classe, de l'instance
  - ◊ Protocole d'accès (opéations, sérialisation…)
- En général une référence est "opaque" (son contenu ne peut pas être directement exploité) ; elle n'est utilisable que via un mécanisme d'appel à distance
- Exemples
  - ◊ Java RMI : la référence est celle du talon (client avant java 5, serveur ensuite))
  - ◊ CORBA : format de référence spécifié (IOR : Interoperable Object Reference)
  - ◊ DCOM : format propriétaire

***Obtention d'instances d'une classe d'objets***

- Statiquement : le serveur crée une instance d'objet appelable à distance et
  - ◊ conserve sa référence pour la fournir à un (ou des) client(s)
  - ◊ ou enregistre (la référence de) cet objet auprès d'un service de nommage
- Dynamiquement : motif(patron) de conception de la fabrique (*factory*)
  - ◊ le serveur crée (statiquement) un objet `fabrique_de_X` accessible à distance
  - ◊ cet objet fournit une méthode `créer_X`, qui crée à chaque invocation une nouvelle instance d'objet de la classe `X` sur le serveur, et en fournit la référence à l'appelant.






## 3) Java RMI (Remote Method Invocation)

***Utilisation : schéma des RPC.***

Le programmeur doit fournir

- Les interfaces des classes d'objets appelables à distance
- Pas d'IDL spécifique : les interfaces sont définies comme des interrfaces Java
- Le programme du serveur
  - ◊ Objets implémentant les interfaces
  - ◊ méthode `main` instanciant et publiant les objets appelables à distance
- Le programme du client



***Réalisation***

L'environnement Java fournit

- la génération des talons : Java permet la génération dynamique des talons, en s'appuyant sur l'API d'introspection de Java pour transmettre les requêtes, et en utilisant du code générique pour gérer l'interaction requête/réponse
  - ◊ les talons clients sont générés à l'exécution, lors de la création des objets distants, à partir de l'environnement d'exécution du serveur
  - ◊ les fonctions du talon serveur sont fournies et intégrées à l'objet distant, par héritage

  Un compilateur de talons explicite et statique (`rmic`) est aussi fourni, pour assurer la compatibilité avec les versions de Java antérieures à la version 5
- un service de nommage (package `registry`, classe `Naming`, application `rmiregistry`), qui permet aux clients de désigner (par des *références distantes*) les objets qu'exportent les serveurs
- un mécanisme de chargement de code dynamique, qui permet aux clients de charger le code des objets fournis en paramètre lorsqu'il n'est pas disponible localement, et en particulier le code des talons correspondant aux références distantes obtenues.

Voir `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html` pour la spécification détaillée des classes et des services disponibles



## Définition des objets appelables à distance

- L'interface d'un objet distant est celle d'un objet Java, avec les contraintes suivantes :
  - ◊ L'interface distante doit être **publique**
  - ◊ L'interface distante doit étendre l'interface **`java.rmi.Remote`**
  - ◊ Chaque méthode de l'interface doit lever au moins l'exception **`java.rmi.RemoteException`**
- Passage d'objets en paramètre
  - ◊ Les objets locaux sont passés par valeur (copie) et doivent être **sérialisables**
  - ◊ Les objets distants sont passés par référence et sont désignés par leur interface
- Réalisation des classes distantes. Une classe distante
  - ◊ doit implémenter une interface elle-même distante
  - ◊ doit spécifier le protocole utilisé pour créer, gérer et exporter les objets distants, souvent en étendant **`java.rmi.server.UnicastRemoteObject`** (ou encore : **`java.rmi.server.Activatable`**, qui permet de suspendre/reprendre un objet distant, selon son utilisation en cours)
  - ◊ peut définir des méthodes appelables localement, en sus des méthodes distantes
- Le serveur doit instancier l'objet distant
  - ◊ Définir le constructeur de l'objet distant
  - ◊ Fournir la réalisation des méthodes appelables à distance
  - ◊ Définir et installer un contrôle d'accès pour les objets chargés (gestionnaire de sécurité)
  - ◊ Créer au moins une instance de la classe serveur
  - ◊ Enregistrer au moins une instance dans le serveur de noms

## Java RMI : exemple (Hello world)

***Définition de l'interface d'un objet appelable à distance***

```
import java.rmi.*;
public interface Hello extends Remote {
  public String sayHello() throws java.rmi.RemoteException;
     /* fournit un message */
}
```

***Classe réalisant l'interface***

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends java.rmi.server.UnicastRemoteObject implements Hello {
  private String texte;

  public HelloImpl(String s) throws RemoteException {
     texte = s ;
  };

  public String sayHello() throws RemoteException {
     return texte ;
  };
}
```



***Programme du serveur***

```
import java.rmi.*;
public class ServeurHello {
  public static void main (String [] argv) {
           /* fixer une politique de contrôle d'accès */
     System.setSecurityManager(new RMISecurityManager());
     try {/*enregistrer une instance de HelloImpl dans le serveur de noms, situé sur yoda.n7.fr*/
        Naming.rebind ("rmi://yoda.n7.fr/Hi1", new HelloImpl("Hello world !"));
        System.out.println ("Le serveur est prêt.") ;
     } catch (Exception e) { System.out.println("Erreur serveur : " + e); }
  }
}
```

***Programme du client***

```
import java.rmi.*;
public class ClientHello {
  public static void main (String [] argv) {
     System.setSecurityManager(new RMISecurityManager());
     try {     /* trouver une référence vers l'objet distant */
        Hello h = (Hello) Naming.lookup("rmi://yoda.n7.fr/Hi1");
              /* appel de méthode à distance */
        System.out.println(h.sayHello());
     } catch (Exception e) { System.out.println("Erreur client : " + e); }
  }
}
```



## Java RMI : Étapes de la réalisation

***Compilation***

- Sur la machine serveur : compiler les interfaces et les programmes du serveur

  `javac Hello.java HelloImpl.java ServeurHello.java`

- Sur la machine client : compiler les interfaces et le programme client

  `javac Hello.java ClientHello.java`

***Exécution***

- Lancer le serveur de noms (sur la machine `yoda.n7.fr`)

  `rmiregistry &`

  `rmiregistry` écoute par défaut sur le port 1099. On peut préciser un autre port mais il faut aussi modifier les URL en conséquence (`rmi://`**`<serveur>:<port>`**`/<nom objet distant>`)

- Lancer le serveur

  ```
  java -Djava.rmi.server.codebase=http://yoda.n7.fr/<répertoire des classes> \
       -Djava.security.policy=serveurHello.policy ServeurHello &
  ```

  Signification des propriétés (options `-D`) :

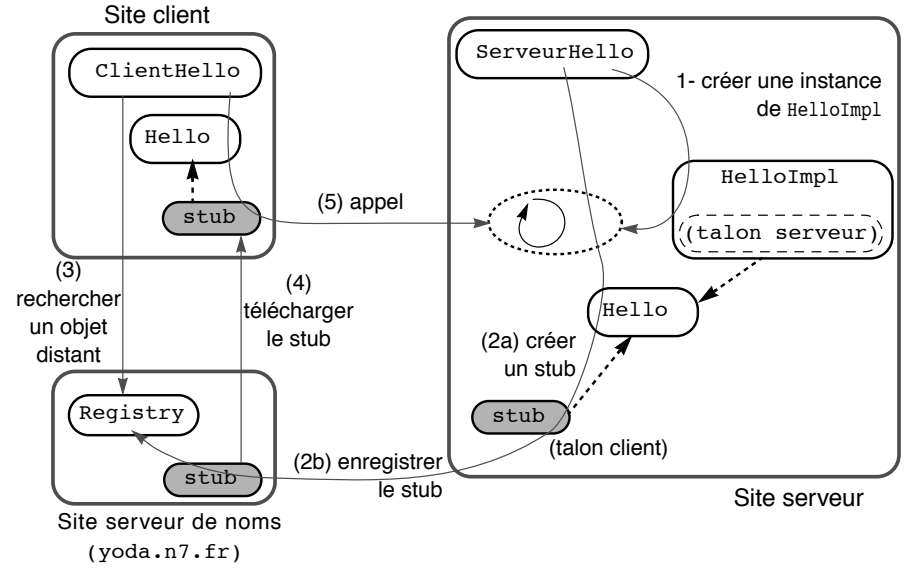
  - Le contenu du fichier **`serveurHello.policy`** spécifie la politique de contrôle d'accès (cf infra)
  - L'URL donnée par `codebase` permet le chargement dynamique de classes par le client

- Lancer le client

  `java -Djava.security.policy=helloC.policy ClientHello`



## Fonctionnement d'ensemble de Java RMI

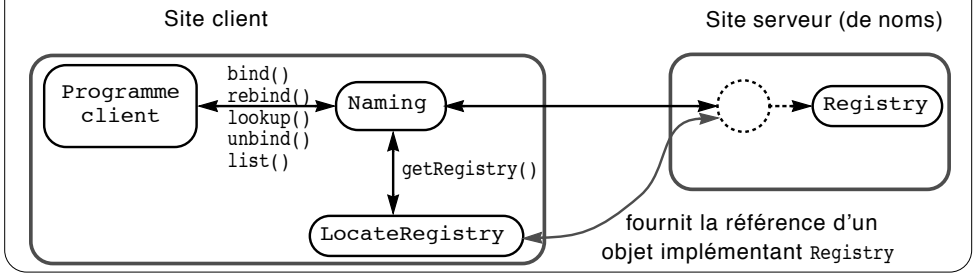


## Principaux éléments du service de RMI

***Le serveur de noms (Registry)***

Classes utiles (fournies par java.rmi)

- `Naming` : sert de représentant local du serveur de noms.
  Fournit les méthodes `bind(), rebind(), lookup(), unbind(), list()`
- `LocateRegistry` : permet de créer (`createRegistry`) ou d'obtenir (`getRegistry`) la référence d'un serveur de noms (`Registry`). Dans le cas d'une création, le serveur de noms est un thread de l'application (généralement le serveur) ayant appelé `LocateRegistry.createRegistry()`
  - ◊ `Naming` fournit les opérations de `Registry`
  - ◊ En général invisible au client (appelé en interne par `Naming`)





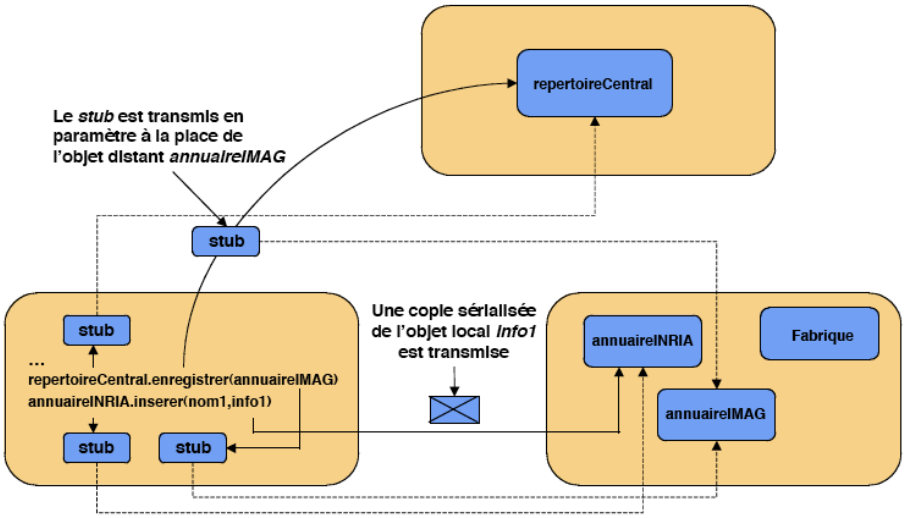

## Passage d'objets en paramètre

Deux cas possibles. L'objet transmis en paramètre peut être

- local (sur la JVM de l'appelant)
  → passage par valeur : on envoie l'état de l'objet
  (plus précisément : la valeur de l'ensemble des attributs « visibles » pour les clients distants)
  Pour cela l'objet doit être sérialisable (i.e. implémenter l'interface `java.io.Serializable`)
- non-local (hors de la JVM de l'appelant, par ex. site distant)
  → passage par référence : on transmet une référence sur l'objet.
  La référence est, de fait un talon client (*stub*) de l'objet
- Un stub contient en particulier une variable `ref` de type `RemoteRef` qui contient la localisation de l'objet (adresse IP, port)
- Un appel de méthode se fait par appel de `ref.invoke(…)`, qui utilise les sockets



## Passage d'objets en paramètre : illustration






## Objets sérialisables

- Un objet sérialisable (transmissible par valeur hors de sa JVM) doit implémenter l'interface `java.io.Serializable`. Celle-ci est réduite à un marqueur (interface vide).
- Les objets référencés dans un objet sérialisable doivent aussi être sérialisables
- Comment rendre effectivement un objet sérialisable ?
  - ◊ Les types primitifs (`int, boolean, …`) sont implicitement sérialisables
  - ◊ Les objets dont les attributs sont des types primitifs sont implicitement sérialisables
  - ◊ On peut éliminer un attribut de la représentation sérialisée en le déclarant `transient`
  - ◊ Un objet sérialisable est (dé)sérialisé en appelant readObject() (resp. writeObject())
    Ces méthodes peuvent être redéfinies pour adapter la (dé)sérialisation
  - ◊ Le support de sérialisation est un stream (flot) : classes `java.io.ObjectOutputStream` et `java.io.ObjectInputStream`.
- Exemples de sérialisation : passage de paramètres, écriture sur un fichier.
- Le chargement dynamique permet qu'une classe soit modifiée en cours d'exécution, et donc que deux instances d'une même classe ne puissent correspondre.
  - ◊ Afin de permettre au récepteur de contrôler que l'instance reçue correspond à la classe qu'il a chargée, un numéro de version de classe est transmis avec chaque instance sérialisée.
  - ◊ Ce numéro de version peut être défini par la programmeur, sous la forme d'un attribut prédéfini de la classe : **`static final long serialVersionUID`**

### *Sécurité*

- La politique de sécurité spécifie les actions autorisées, en particulier sur les sockets
- Cette politique est implantée par un gestionnaire de sécurité (`SecurityManager`)
- Des gestionnaires de sécurité sont prédéfinis (`AppletSecurityManager, RMISecurityManager…`)
- La politique mise en œuvre par le gestionnaire de sécurité installé peut être amendée/ complétée, en listant des autorisations/interdictions dans un fichier `policy`. Par exemple

```
grant {
  permission java.net.SocketPermission "*:1024-65535", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
};
grant codeBase "file:/home/gertrude/java/" {
  permission java.security.AllPermission;
};
```

Permet d'utiliser les sockets comme indiqué, et permet toutes les opérations aux programmes du répertoire `/home/gertrude/java/` de la machine locale. Toute autre utilisation est interdite

### *Compléments*

**Parallélisme sur le serveur**

Le développeur du serveur doit synchroniser (p. ex. en spécifiant des méthodes `synchronized`) les threads créés pour réaliser les objets distants, afin de garantir la cohérence de l'état du serveur.

**Activation automatique de serveurs**

Un veilleur (démon) spécialisé, `rmid`, peut assurer l'activation/le lancement d'objets distants.



## Schémas d'interaction utiles

Source : cours de Mastère Université Joseph Fourier de Sacha Krakowiak (en ligne)

### *Fabrique d'objets (Factory)*

**Motivation : Permettre au client de construire des instances multiples d'une classe C sur le site serveur**

- Le `new` n'est pas utilisable tel quel (car il ne gère que la mémoire locale, celle du client)
  → appel d'un objet `FabriqueC`, qui crée sur le serveur les instances de C (en utilisant new C)

**Exemple**

```
import java.rmi.*
public interface FabAnnuaire extends Remote {
  public Annuaire newAnnuaire(String titre) throws RemoteException;
}
public interface Annuaire extends Remote {
  public String titre;
  public boolean inserer(String nom, Info info) throws RemoteException, ExisteDeja;
  public boolean supprimer(String nom) throws RemoteException, PasTrouve;
  public Info rechercher(String nom) throws RemoteException, PasTrouve;
}
public class Info implements Serializable {
  public String adresse;
  public int num_tel;
}
public class ExisteDeja extends Exception {};
public class PasTrouve extends Exception {};
```



**Mise en oeuvre d'une fabrique**

```
import java.rmi.*;
public class AnnuaireImpl implements Annuaire extends UnicastRemoteObject {
  private String letitre;
  public Annuaire(String titre) throws RemoteException { this.letitre = titre };
  public String getTitre() { return letitre };
  public boolean inserer(String nom, Info info) throws RemoteException,ExisteDeja {…};
  public boolean supprimer(String nom) throws RemoteException, PasTrouve {…};
  public Info rechercher(String nom) throws RemoteException, PasTrouve {…};
}

public class FabAnnuaireImpl implements FabAnnuaire extends UnicastRemoteObject {
  public FabAnnuaireImpl {};
  public Annuaire newAnnuaire(String titre) throws RemoteException {
     return new AnnuaireImpl(titre)};
  }

public class Server {
  public static void main (String [ ] argv) {
     System.setSecurityManager (new RMISecurityManager ()) ;
     try {
        Naming.rebind ("rmi://yoda.n7.fr/Fab", new (FabAnnuaireImpl));
        System.out.println ("Serveur prêt.") ;
     } catch (Exception e) {System.out.println("Erreur serveur : " + e);}
  }
}
```



**Utilisation de la fabrique**

```
import java.rmi.*;
public class HelloClient {
  public static void main (String [ ] argv) {
     System.setSecurityManager (new RMISecurityManager ()) ;
     try { /* trouver une référence vers la fabrique */
       FabAnnuaire fabrique = (FabAnnuaire) Naming.lookup("rmi://yoda.n7.fr/Fab");
        /* créer un annuaire */
        annuaireIMAG = fabrique.newAnnuaire("IMAG");
        /* créer un autre annuaire */
        annuaireINRIA = fabrique.newAnnuaire("INRIA");
        /* utiliser les annuaires */
        annuaireIMAG.inserer(…, …);
        annuaireINRIA.inserer(…, …);
        …
     } catch (Exception e) { System.out.println("Erreur client: " + e);}
  }
}
```

**Fonctionnement d'ensemble de la fabrique**

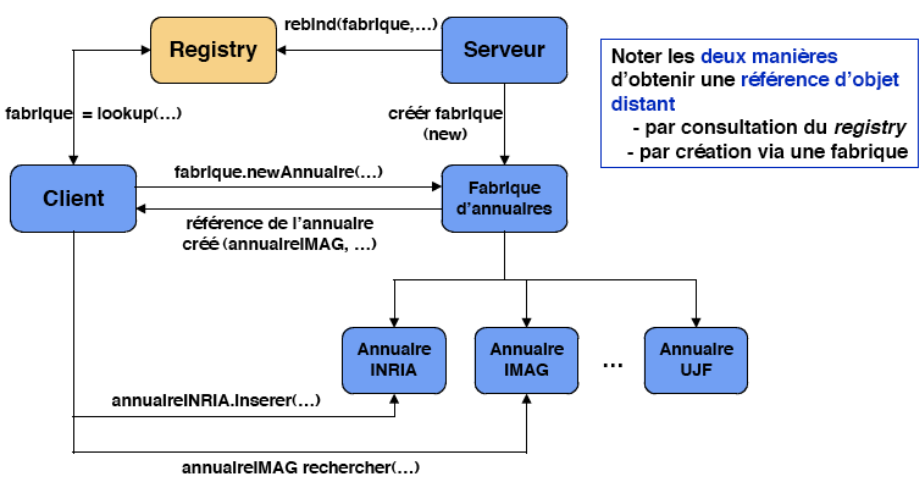





***Schéma de rappel (callback)***

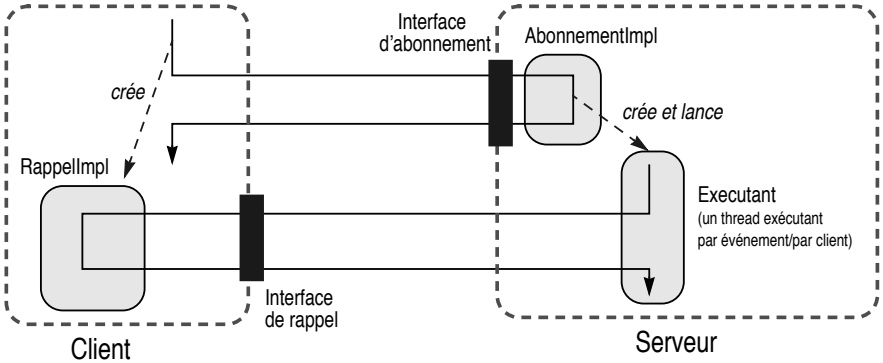
Permettre au serveur de rappeler un client l'ayant contacté auparavant

***Asynchronisme*** : schéma publier/s'abonner

***Dynamisme*** : à l'exécution, le serveur peut demander au client des données/calculs complémentaires

**Principe**

- Le client passe en paramètre au serveur l'objet à rappeler
- Le serveur exécute un appel asynchrone sur cet objet






**Rappel : réalisation des interfaces des objets distants** (doivent étendre Remote)

```
// les deux interfaces des objets appelés à distance
import java.rmi.*;
public interface Rappel extends Remote { // interface de l'objet callback
  public void signaler(String message) throws RemoteException;
}
import java.rmi.*; // interface du serveur
public interface Abonnement extends Remote {
 public void sAbonner(int time,String param, Rappel rap) throws RemoteException;
}
```

**Rappel : réalisation (le servant)**

```
import java.rmi.*;
public class Executant extends Thread {
  private int time; private String param; private Rappel rap;
  public Executant(int time, String param, Rappel rap) {
     this.time = time; this.param = param; this.rap = rap;
  }
  public void run() { // exécution du servant comme thread séparé
     try { Thread.sleep(1000*time); // attend time secondes)
     } catch(InterruptedException e) { }
     try { rap.signaler(param); // exécute l'appel en retour
     } catch(RemoteException e) { System.err.println("Echec appel en retour : "+e); }
  }
}
```



**Rappel: réalisation (le serveur)**

```
import java.rmi.*;
import java.rmi.server.*;
public class AbonnementImpl extends UnicastRemoteObject implements Abonnement {
  public AbonnementImpl() throws RemoteException { super(); }
  public void sAbonner(int time, String param,Rappel rap) throws RemoteException {
     Executant ex = new Executant(time, param,rap);
     ex.start(); // lancement de l'exécutant
  }
  public static void main(String[] args) throws Exception {
     AbonnementImpl serveur = new AbonnementImpl(); //création et démarrage du serveur
     Naming.rebind("Serveur",serveur);
     System.out.println("Serveur pret");
  }
}
```

**Rappel : réalisation (le rappel)**

```
import java.rmi.*;
import java.rmi.server.*;
public class RappelImpl extends UnicastRemoteObject implements Rappel {
  public Rappel() throws RemoteException { super(); }
  public void signaler(String message) throws RemoteException {
     System.out.println(message);
  }
}
```

**Rappel : réalisation (le client)**

```
import java.rmi.*;
public class Client {
  public static void main(String[] args) throws Exception {
     RappelImpl rap = new RappelImpl(); //création d'un objet de rappel
     Abonnement abo = (Abonnement) Naming.lookup("Serveur");
     System.out.println("démarrage du client");
     abo.sAbonner(5, "coucou", rap); //demande de rappel
     for (int i=1; i<=5; i++) {
        System.out.println(i);
        try { Thread.sleep(2000);} catch (InterruptedException e) { }
     }
     System.out.println("fin du main");
  }
}
```



## Conclusion sur Java RMI

Extension du RPC aux objets

- Permet l'accès à des objets distants
- Permet d'étendre l'environnement local par chargement dynamique de code
- Pas de langage séparé de description d'interfaces (IDL fourni par Java)

Limitations

- Environnement restreint à un langage unique (Java)
  - ◊ Mais passerelles possibles, en particulier RMI/IIOP
- Services réduits au minimum
  - ◊ Service élémentaire de noms (sans attributs)
  - ◊ Services additionnels (duplication d'objets, transactions…) fournis par des plateformes s'appuyant sur les RMI (EJB…)