

Troisième partie

Appel de procédure et de méthode à distance

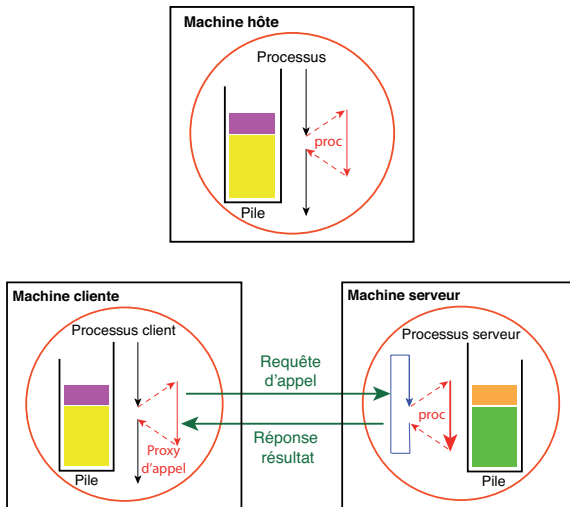


Plan

- 1 L'appel procédural à distance
 - Introduction
 - Transparence
 - Sémantiques
 - Paramètres
 - Désignation et liaison
 - Mise en œuvre
- 2 L'appel de méthode à distance
 - Sémantique et propriétés
 - Sérialisation
 - RMI de Java
 - Exemple basique
 - Exemple : callback



Communication par appel procédural à distance alias Remote Procedure Call (RPC)



Communication par appel procédural à distance

Extension répartie du mécanisme d'appel procédural

- Procédure appelée exécutée dans un espace **différent** de celui de l'appelant
- Synchronisation appelant-appelé
- Transaction de messages
(question - réponse - [acquiescement])
- Fiabilité bien moindre qu'en centralisé
- Comment transmettre les paramètres ?
- Problème de l'hétérogénéité
 - du matériel
 - du système d'exploitation
 - de la représentation des données (paramètres)
 - des langages de programmation



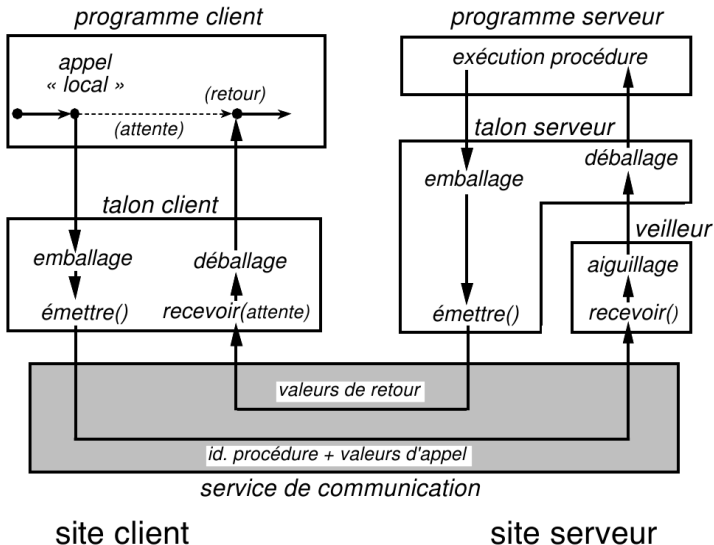
Transparence

But : rendre l'utilisation de l'appel à distance aussi conforme (*transparent*) que l'appel local de procédure

- passage des paramètres
- liaison (nommage)
- protocole de transport
- exceptions
- sémantique de l'appel
- représentation des données
- performance
- sécurité

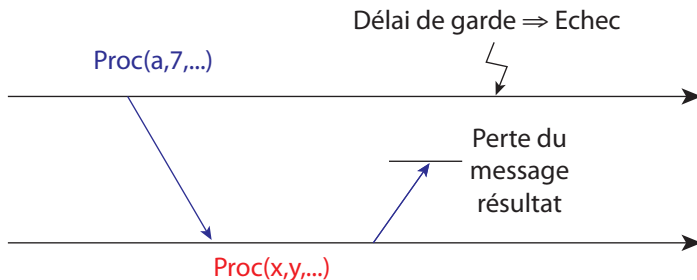


Principe général



Sémantique de l'appel procédural à distance

Quelques problèmes. . .

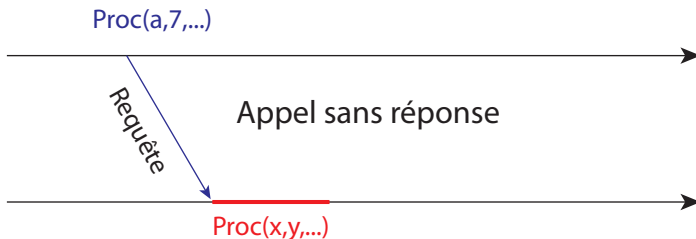


Plusieurs sémantiques possibles !

- « Sans garantie » (Maybe / Best effort)
- « Au moins une fois » (At-least-once)
- « Au plus une fois » (At-most-once)
- « Exactement une fois » (Exactly-once).

Sémantique de l'appel procédural à distance

Sémantique minimaliste : « Sans garantie »

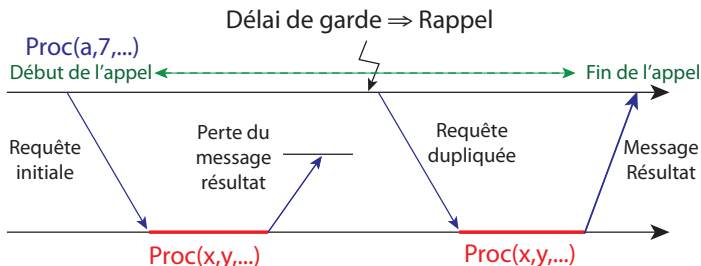


Avantages et inconvénients

- Simple à implanter : envoyer un message
- Pas de réponse donc pas de garantie d'exécution
- Utile dans certains cas (logging)

Sémantique de l'appel procédural à distance

Sémantique « Au moins une fois »



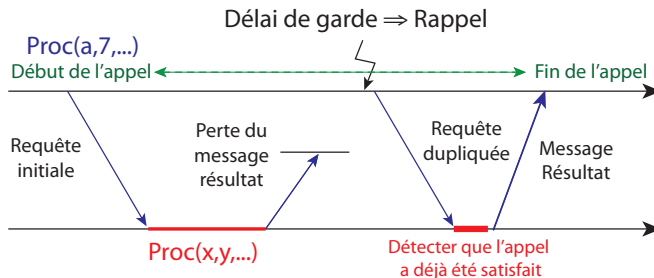
Avantages et inconvénients

- Robuste face aux pertes et lenteurs
- Si terminaison correcte \rightarrow garantie d'une exécution au moins
- Si terminaison incorrecte (après plusieurs rappels), pas de garantie sur ce qui s'est passé à distance
- Risque de plusieurs exécutions pour un **seul** appel logique

27

Sémantique de l'appel procédural à distance

Sémantique « Au plus une fois »



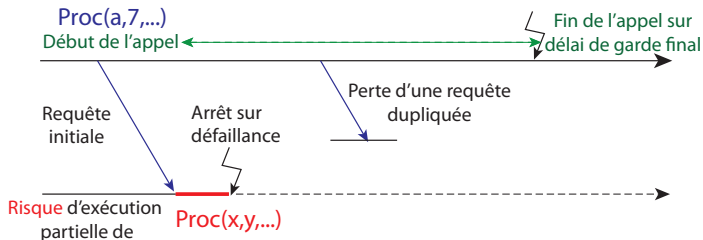
Avantages et inconvénients

- Plus proche de l'appel procédural centralisé
- Terminaison correcte \Rightarrow garantie d'une **seule** exécution à distance
- Terminaison incorrecte : pas de garantie sur ce qui s'est passé à distance \Rightarrow entre autre, risque d'exécution partielle à distance



Sémantique de l'appel procédural à distance

Sémantique « Exactly once »



Avantages et inconvénients

- Si terminaison correcte \Rightarrow équivalent à « au + une fois »
- Si terminaison incorrecte \Rightarrow garantie d'atomicité à distance



Espace d'adressage séparés

+ Isolation des données du serveur

- Sécurité
- Conception modulaire

– Passage des paramètres au moyen de messages

- pas de variables globales implicites
- passage **par valeur** (copie)
- Références ?
 - Interdire le passage de références \Rightarrow expressivité faible
 - Sérialisation : transfert et reconstruction du *graphe* (structure de données)
 - Références globales opaques (objets répartis)



Représentation des données

Problèmes

- **hétérogénéité** du matériel
- taille des types élémentaires (booléen, entier...)
- ordre des octets pour les entiers
- nombres réels
- codage des caractères
- données composites : structures, tableaux

Approches

- 1 définir une représentation standard
 - typage implicite : seules les valeurs sont transmises
 - typage explicite : information de type + valeur
- 2 préfixer par le type local, et laisser le destinataire convertir si nécessaire

Désignation et liaison

Correspondance : nom symbolique (externe) → nom interne
(adresse réseau, identifiant local)

Cas des RPC :

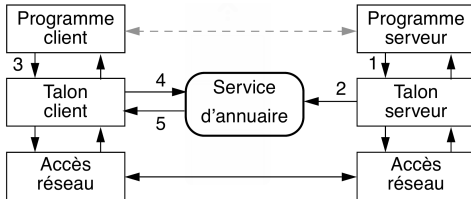
- nom de service → port et adresse serveur
- nom d'opération → procédure sur le serveur correspondant

Instant(s) d'évaluation de la liaison

- Liaison statique (précoce) : localisation du serveur fixée au moment de la compilation du programme client
- Liaison dynamique (tardive) : localisation à l'exécution
 - désignation symbolique des services
 - choix retardé de l'implémentation
 - localisation du service au premier appel seulement, ou à chaque appel
 - adaptation à une reconfiguration du système : régulation, pannes, évolutions



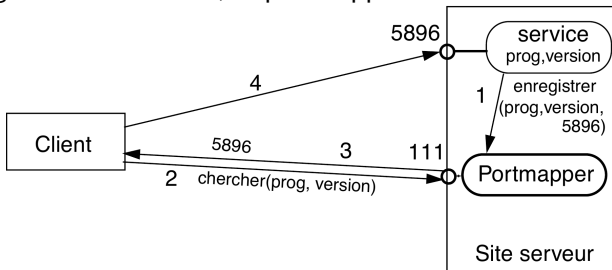
Liaison dynamique : serveur de noms



- 1,2 : Enregistrement du service dans l'annuaire sous :
 $\langle \text{nom} \rangle \rightarrow \langle \text{adr. serv.}, n^{\circ} \text{ port} \rangle$
- 3,4,5 : Consultation de l'annuaire pour trouver $\langle \text{adr. serv.}, n^{\circ} \text{ port} \rangle$ à partir de $\langle \text{nom} \rangle$
- L'appel peut alors avoir lieu
- Tolérance aux pannes (service critique) : mémoire stable, duplication des tables, des serveurs
- Localisation du serveur de noms :
 - diffusion de la requête par le client
 - ou variable de configuration du système
 - ou utilisation d'une adresse conventionnelle

Serveurs de noms local à un site : portmapper

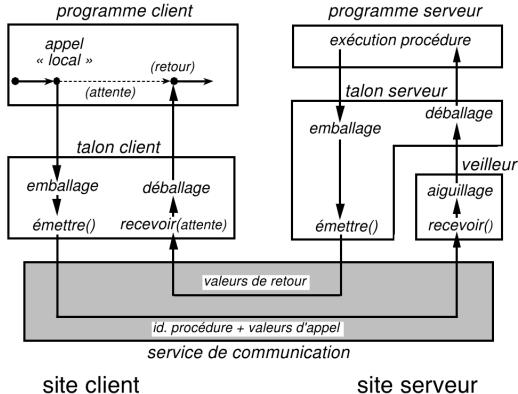
Cas où le site hébergeant le service est connu, mais où le port correspondant au service n'est pas connu \Rightarrow utiliser un service de nommage local au serveur, le portmapper.



- Le portmapper a un numéro 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



Mise en œuvre



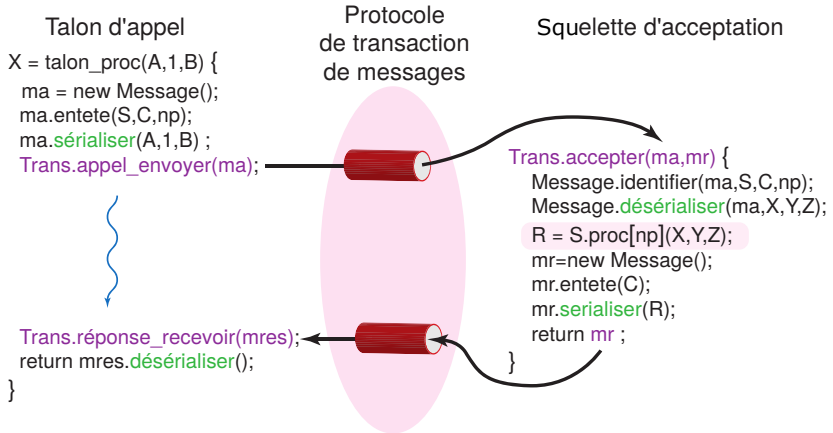
- Définition d'un protocole de transaction de messages
- Sérialisation/Désérialisation des paramètres
- Génération de talon d'appel (stub) et d'acceptation (skeleton)
- Assistance par génération automatique de code

Langage de description d'interface (IDL)



Mise en œuvre

Niveaux de protocole



Description d'interface en IDL

Exemple minimaliste (SUN)

Les langages IDL (Interface Definition Language)

- Langage commun de description d'interface
- Purement déclaratif : types de données, interfaces
- Base pour la génération des talons et squelettes

```
struct arg { int a1; int a2; };  
program MONSERVICE {  
    version MAVERSION {  
        int PROC1 (arg) = 1;  
        int PROC2 (int, int) = 2;  
    } = 1;  
} = 0x30000050;
```



eXternal Data Representation

- Description et codage des données indépendantes du matériel
- Structures de données arbitraires (structures, tableaux, séquences)
- RPC convertit les données machine en XDR avant de les envoyer sur le réseau (*sérialisation*)
- RPC convertit les données XDR en données machine après lecture sur le réseau (*désérialisation*)
- Génération automatique des routines de sérialisation et de désérialisation à partir d'une description proche du langage C
- ou codage manuel de ces routines à partir d'une librairie pour les types élémentaires



Programmation par appel explicite

```
int callrpc (const char *host,  
            u_long prognum, u_long versnum, u_long procnum,  
            xdrproc_t inproc, char *in,  
            xdrproc_t outproc, char *out);
```

Appel d'une procédure $\langle \text{prognum}, \text{versnum}, \text{procnum} \rangle$ sur la machine `host`. La procédure reçoit en paramètre `in` codé avec `inproc` et retourne `out` décodé avec `outproc`.

```
int registerrpc (u_long prognum, u_long versnum,  
                u_long procnum,  
                char * (*procname) (char *arg),  
                xdrproc_t inproc, xdrproc_t outproc);
```

Enregistrement de la procédure `procname` sous le nom $\langle \text{prognum}, \text{versnum}, \text{procnum} \rangle$. Le paramètre est décodé avec `inproc` et le résultat est encodé avec `outproc`.



Appel explicite : client

```
#include <stdio.h>
#include <rpc/rpc.h>
int main()
{
    int stat;
    int result;
    char *argin = "546";
    stat = callrpc (/*host*/ "cotiella",
                   /*prognum*/ 87654321, /*versnum*/ 1, /*procnum*/ 12,
                   /*inproc*/ xdr_string, /*in*/ (char*) &argin,
                   /*outproc*/ xdr_int, /*out*/ (char*) &result);
    if (stat != 0) { fprintf (stderr, "Call failed: "); /*...*/ }
    printf ("Call succeeded: \"%s\" = %d\n", argin, result);
}
```



Appel explicite : serveur

```
#include <rpc/rpc.h>
#include <stdlib.h>
int resultat;
char *maproc (char *argin)
{
    char *argument = *(char**)argin;
    resultat = atoi(argument);
    return (char*) &resultat;
}
int main()
{
    int stat;
    stat = regerrpc (/*prognum*/ 87654321, /*versnum*/ 1, /*procnum*/ 12,
                    /*procname*/ maproc,
                    /*inproc*/ xdr_string, /*outproc*/ xdr_int);
    if (stat != 0) { fprintf (stderr, "Registering failed\n"); /* ... */ }
    svc_run(); /* veilleur/aiguilleur */
}
```

Diffusion & protocole

```
typedef bool_t (*resultproc_t) (char *out,  
                                struct sockaddr_in *addr);  
int clnt_broadcast(u_long prognum, u_long versnum,  
                  u_long procnum,  
                  xdrproc_t inproc, char *in,  
                  xdrproc_t outproc, char *out,  
                  resultproc_t eachresult);
```

Envoi diffusé sur le réseau, la fonction `eachresult` est appelée pour chaque réponse obtenue.

Protocole de transport

Par défaut pour `callrpc` et `clnt_broadcast`, le protocole de communication est UDP. Pour `callrpc`, possibilité d'utiliser TCP via `client_create` (à suivre).



Génération automatique : rpcgen

À partir d'une description des types et d'une déclaration des procédures, RPCGEN engendre :

- les routines XDR pour convertir les types de données
- le talon client masquant l'appel à distance
- le talon serveur gérant l'appel
- la procédure principale (`main`) et la procédure de sélection (`dispatch`) pour le serveur

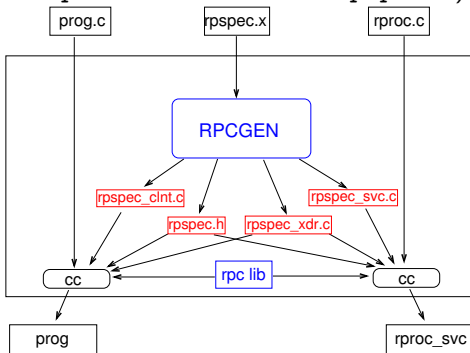
Le programmeur doit :

- décrire les types de données échangées
- écrire la programme principal client
- écrire le code des procédures coté serveur



RPCGEN : exemple

Un programme (fichier prog.c) appelle deux procédures distantes (implantées dans rproc.c, décrites dans rpspec.x) d'un serveur



```
$ rpcgen rpspec.x
$ cc -o server rproc.c rpspec_svc.c rpspec_xdr.c -lnsl
$ cc -o client prog.c rpspec_clnt.c rpspec_xdr.c -lnsl
```



RPCGEN : Fichier d'interface

`rpspec.x` contient :

- les noms, et les numéros de programme, de version et de procédures, des procédures distantes
- les types de données manipulés

```
/* rpspec.x */  
struct arga {  
    int arga1;  
    int arga2;  
};  
  
program MONPROG {  
    version MAVERS {  
        long RPROCA (string, int) = 1; /* proc #1 */  
        arga RPROCB (void) = 2;      /* proc #2 */  
    } = 1;                             /* version #1 */  
} = 0x1234567;                         /* prog num */
```



RPCGEN : un exemple de client

```
#include <rpc/rpc.h>
#include "rpspec.h"
main() {
    CLIENT *cl;
    long *res1;
    struct arga *res2;

    /* Mise en place du lien. cotiella est le nom de la machine distante. */
    cl = clnt_create ("cotiella", MONPROG, MAVERS, "tcp");
    if (cl == NULL) { clnt_pcreateerror ("cotiella"); exit(1); }

    res1 = rproca_1 ("45", 42, cl); /* appel de rproca version 1 */
    if (res1 == NULL) { clnt_perror (cl, "failed"); exit (1); }
    printf("Result: %ld\n", *res1);

    res2 = rprocb_1 (cl); /* appel de rprocb version 1 */

    clnt_destroy (cl); /* fin du lien */
}
```



RPCGEN : procédures du serveur

```
#include "rpspec.h"

/* Implantation de RPROCA version 1 */
/* Passage par pointeur du retour */
/* req contient des informations de contexte */
long *rproca_1_svc (char *arg1, int arg2, struct svc_req *req) {
    static long res;
    res = atoi(arg1) + arg2;
    return &res;
}

/* Implantation de RPROCB version 1 */
struct arga *rprocb_1_svc (struct svc_req *req) {
    /* ... */
}
```



RPC : bilan

Apports

- Transparence partielle (désignation, localisation)
- Modèle de programmation classique (appel procédural ↔ interaction C/S)
- Conception modulaire
- Outils de génération automatique des talons

Limitations

- Développement traditionnel monolithique
- Construction et déploiement statique
- Structure d'exécution asymétrique, centralisée sur le serveur
- Peu de services extra-fonctionnels : supervision, équilibrage de charge, tolérance aux pannes, données rémanentes...



Et variantes

XML-RPC

Un protocole RPC qui utilise XML pour encoder les données et HTTP comme mécanisme de transport.

SOAP (Simple Object Access Protocol)

Architecture client-serveur via des échanges de messages décrits en XML. Neutralité du transport (HTTP, SMTP, TCP, JMS).

REST (Representational state transfer)

Règles architecturales de construction d'applications client-serveur, s'appuyant sur HTTP et axé sur les ressources (RPC est axé sur les actions). Une règle essentielle est que la relation client-serveur est sans état (pas de session).



Plan

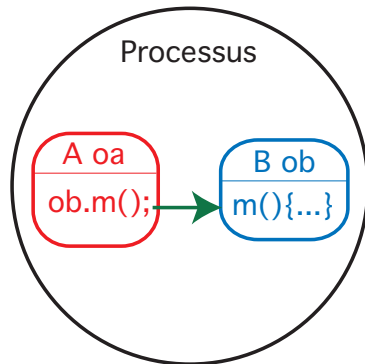
- ① L'appel procédural à distance
 - Introduction
 - Transparence
 - Sémantiques
 - Paramètres
 - Désignation et liaison
 - Mise en œuvre
- ② L'appel de méthode à distance
 - Sémantique et propriétés
 - Sérialisation
 - RMI de Java
 - Exemple basique
 - Exemple : callback



L'appel de méthode centralisé

Principe et propriétés

- Un seul espace d'exécution
- Point de contrôle unique
- Fort couplage
- Fiabilité
- Sécurité

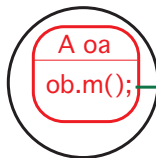


L'appel de méthode à distance

Principe et propriétés

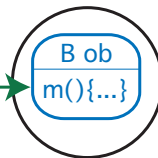
- Deux espaces d'exécution
- Deux points de contrôle
- Couplage plus faible
- Protocole de communication entre processus.

Processus client



Réseau

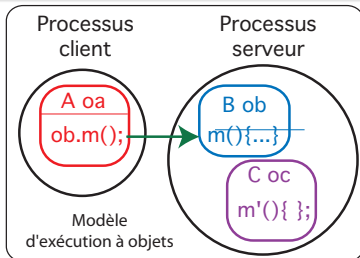
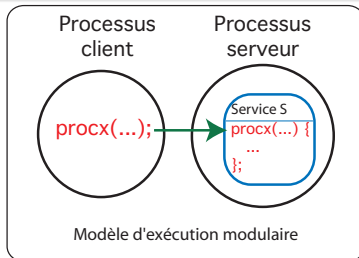
Processus serveur



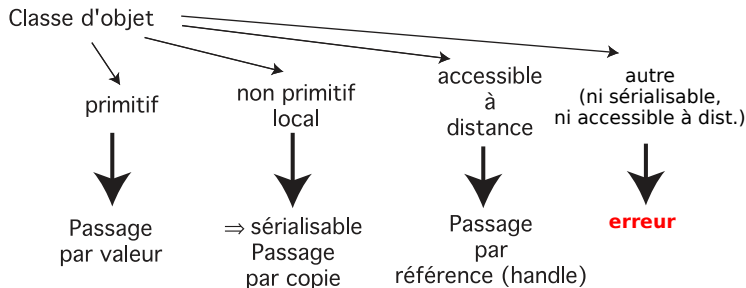
L'appel de méthode à distance

Différences avec l'appel de procédure

- Contexte d'exécution différent : l'un « module », l'autre « objet »
- Appel d'une méthode sur un objet
- Aspect dynamique : création de « services »
+ transmission de services à distance



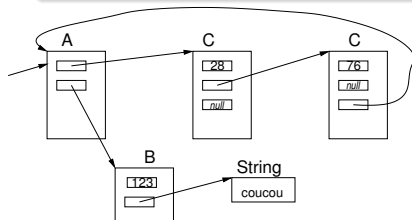
Le passage de paramètres



Sérialisation

Définition

La sérialisation d'un graphe d'objets consiste à obtenir une représentation linéaire inversible de ces objets et de leurs relations. La désérialisation reconstruit une forme interne et structurée du graphe d'objet.



- but : exportation vers un fichier ou un autre processus
- Difficulté : la présence de cycles



Sérialisation

Un objet est sérialisable, s'il appartient à une classe :

- qui implante l'interface `java.io.Serializable`
⇒ pas de code à fournir, mécanisme par défaut :
Sont récursivement sérialisés les attributs non statiques ni *transients* contenant :
 - des types primitifs (`int`, `bool...`)
 - ou des objets qui doivent être sérialisables.
- ou qui implante l'interface `java.io.Serializable` et fournit les méthodes (code utilisateur arbitraire) :

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException;
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```



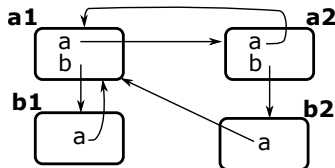
Exemple sérialisation

```
class A implements java.io.Serializable {  
    public B b;  
    public A a;  
}
```

```
class B implements java.io.Serializable {  
    public A a;  
}
```

```
A a1 = new A();    A a2 = new A();  
B b1 = new B();    B b2 = new B();  
a1.a = a2;  
a2.a = a1;  
b1.a = a1;  
b2.a = a1;
```

```
ObjectOutputStream oos = new ObjectOutputStream(  
    new FileOutputStream("/tmp/toto"));  
oos.writeObject(a1);
```



Compatibilité de versions

Comment s'assurer, à l'endroit et au moment de la désérialisation, que l'implantation de la classe est la même qu'à la sérialisation (mêmes attributs en particulier) ?

⇒ **gestionnaire de version des classes**

Solution élémentaire : un attribut statique serialVersionUID (type long) dans chaque classe :

```
private static final long serialVersionUID = 76428734L;
```

Par défaut si absent, le compilateur calcule un tel champ (à partir des attributs notamment), mais le calcul est sensible à son humeur

⇒ à gérer soi-même.



Le mécanisme RMI (Remote Method Invocation)

Proxy

Objet local « remplaçant » l'objet distant = objet ayant la même interface que l'objet distant, et sachant appeler l'objet distant.

Servant

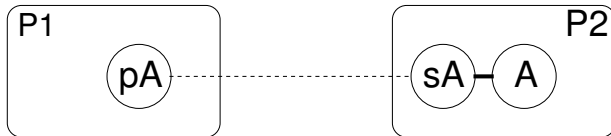
Objet interne sachant discuter à distance avec des proxys et localement avec l'objet applicatif.

Service de nommage

Désignation globale par serveurs de noms (Registry)



Talons : proxy/servant



Proxy = talon client = stub = a la même interface que l'objet applicatif distant.

Servant = talon serveur = squelette = reçoit les requêtes, appelle la méthode correspondant de l'objet applicatif, et gère les erreurs. Le servant peut être un objet distinct (association), ou être commun à l'objet applicatif (héritage).



Obtenir un proxy ?

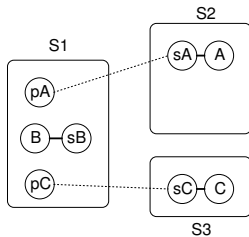
Comment obtenir un proxy sur un objet distant :

- Utiliser un **service de nommage**, qui conserve des associations entre objet accessible à distance et *nom externe* (une chaîne de caractères ou un URL) : le client demande au service de nommage de lui fournir un proxy correspondant à un nom externe donné.
- Avoir appelé une méthode (à distance) qui transmet/renvoie un (autre) objet accessible à distance : création implicite des proxys.

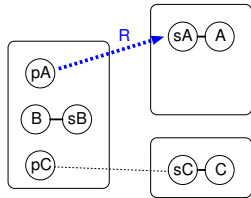


Le mécanisme RMI

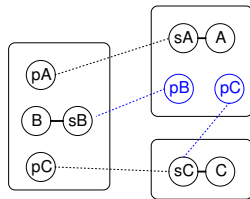
La gestion des proxys et squelettes



S1: Appel de pA.foo(B,4,pC)



S1 -> S2 : Envoi de la requête (A, foo, B, 4, C)



S2 : création du proxy pB
création du proxy pC
appel de A.foo(pB,4,pC)

Réalisation

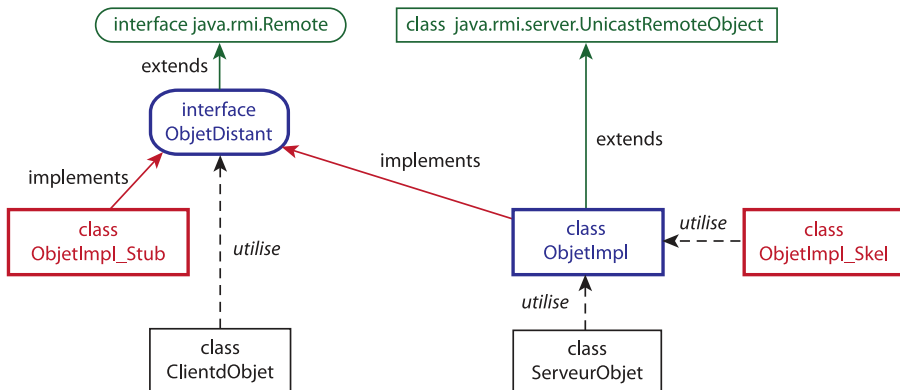
L'environnement Java fournit :

- la génération **dynamique** des talons, en s'appuyant sur l'API d'introspection
 - le proxy est généré à l'exécution, lors de la création d'une référence à un objet accessible à distance, à partir de l'environnement d'exécution du serveur ;
 - les fonctions du servant sont fournies et intégrées à l'objet accessible à distance, par héritage ;
 - historiquement, il existait un générateur statique de talons (`rmic`), analogue à `rpcgen`.
- un service de nommage (package `java.rmi.registry`), pour nommer symboliquement des objets accessibles à distance.
- 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 (en particulier le code des proxys).



Mise en œuvre du protocole

Composants d'une classe d'objet accessible à distance



La description d'une classe d'objet accessible à distance

Définition de l'interface d'appel

- Hérite de l'interface `Remote`
- Chaque méthode lève l'exception `RemoteException`
- Un objet local non primitif doit être sérialisable :
 class `ObjParam` implements `Serializable`
- Un objet accessible à distance peut être passé en paramètre

Exemple

```
interface ObjetAAD extends Remote {  
    void m(int i, ObjParam p) throws RemoteException;  
    String mm(OAADParam pr) throws RemoteException;  
    ...  
}
```



La description d'une classe d'objet accessible à distance

Définition de la classe d'implantation

- Hérite de la classe **UnicastRemoteObject**
- Implante l'interface du proxy correspondant

Exemple

```
class ObjetAADImpl extends UnicastRemoteObject
    implements ObjetAAD {
    void m(int i, ObjParam p) {...}
    String mm(OAADParam pr) {... }
    ...
}
```



Exemple : un agenda

Fait référence à l'interface `Serializable`.

```
public class RendezVous implements java.io.Serializable {  
    private String qui;  
    private java.util.Date date;           // est sérialisable  
    private java.time.Duration duree;      // est sérialisable  
    private String salle;  
  
    public String toString(){... }  
  
    public RendezVous (String qui, Date date, Duration duree,  
                        String salle) {  
        this.qui = qui;  
        this.date = date;  
        this.duree = duree;  
        this.salle = salle;  
    }  
}
```



Interface de l'agenda accessible à distance

```
import java.rmi.*;
import java.util.Date;
import java.time.Duration;
interface Agenda extends Remote {
    public void ajouter(RendezVous rdv)
                        throws RemoteException;
    public boolean déplacer(RendezVous rdv, Date date)
                        throws RemoteException;
    public boolean estLibre(Date date, Duration duree)
                        throws RemoteException;
    public void effacer(String nom, Date date)
                        throws RemoteException;
    public RendezVous[] lister(String nom)
                        throws RemoteException;
    public void importer(String nom, Agenda orig)
                        throws RemoteException;
}
```



Agenda : une implantation

```
import java.rmi.*;
import java.rmi.server.*;

public class AgendaImpl
    extends UnicastRemoteObject implements Agenda {
    private Set<RendezVous> table = new HashSet<>();

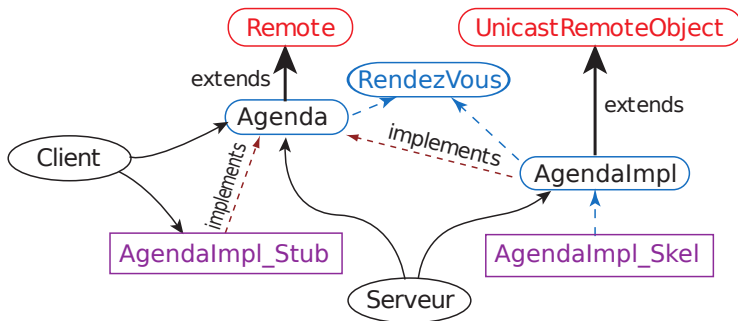
    AgendaImpl() throws RemoteException { }

    public void ajouter(RendezVous rdv) { table.add(rdv); }
    :
    public void importer(String nom, Agenda orig)
        throws RemoteException {
        RendezVous[] rr = orig.lister(nom);
        for (RendezVous r : rr) this.table.add(r);
    }
}
```



Structure des classes

- Génération statique ou dynamique des stubs et des skeletons.



⇒ Cas statique : utilisation du générateur `rmic` (obsolète)

Agenda : un programme serveur

Cas où le serveur de noms est créé en tant que thread interne

```
import java.rmi.registry.*;

public class Serveur {
    public static void main(String args[]) throws Exception {
        Agenda aa = new Agenda();
        Registry dns = LocateRegistry.createRegistry(1099);
        dns.bind("Travail",aa);
    }
}
```



Agenda : un programme client

```
import java.rmi.registry.*;

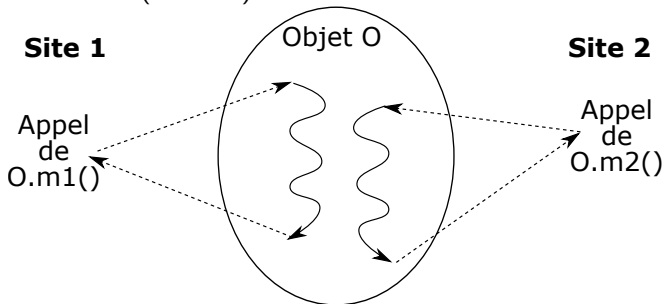
public class Client {
    // args[0] contient le site support du service de nommage
    public static void main(String args[]) throws Exception {
        Registry dns = LocateRegistry.getRegistry(args[0], 1099);
        Agenda proxy = (Agenda) dns.lookup("Travail");
        RendezVous rdv = new RendezVous("chef", new Date(...),
                                           Duration.ofMinutes(30), "F303"
        proxy.ajouter(rdv);
        RendezVous[] tous = proxy.lister("chef");
        for (RendezVous r : tous) System.out.println(r.date);
    }
}
```

Exemple d'appel : `java Client emeraude.enseeiht.fr`



Concurrence

Chaque invocation d'une méthode par un appel à distance se fait dans une activité (thread) distincte.




⇒ nécessité de gérer la protection des attributs et autres données partagées (p.e. **synchronized** pour toutes les méthodes).



Interface du service de nommage

`java.rmi.registry.Registry` : interface d'accès au nommage.
Les noms d'objets doivent obéir à la syntaxe générale des URL :
`rmi://host:port/name`, ou `name` si le contexte est non ambigu.

```
public class Registry {  
    public Remote lookup(String name)  
        throws NotBoundException, MalformedURLException,  
               UnknownHostException, RemoteException;  
    public void bind(String name, Remote obj)  
        throws AlreadyBoundException, MalformedURLException,  
               UnknownHostException, RemoteException;  
    public void unbind(String name)  
        throws RemoteException, NotBoundException,  
               MalformedURLException, UnknownHostException;  
    public String[] list(String name)  
        throws RemoteException, MalformedURLException,  
               UnknownHostException;  
}
```



Localisation et accès au service de nommage

La classe `java.rmi.registry.LocateRegistry` offre un ensemble de méthodes pour créer ou obtenir l'accès à un serveur de noms local ou distant :

```
public final class LocateRegistry {  
    public static Registry getRegistry(String host, int port)  
        throws RemoteException, UnknownHostException;  
    ...  
    public static Registry createRegistry()  
        throws RemoteException;  
}
```



Remarques sur le service de nommage

- Le service de nommage peut aussi être une application autonome (programme `rmiregistry`), et peut résider sur un site différent.
- Un objet accessible à distance n'a pas nécessairement à être enregistré dans le service de nommage : seuls les objets racines le doivent.



Implantation du service de nommage

Le service de nommage n'est lui-même qu'un objet accessible à distance. C'est un objet « notoire », avec une identité fixée : $\langle \text{adr machine}, n^{\circ} \text{ port} \rangle$ suffit à le trouver.

```
class RegistryImpl extends java.rmi.server.RemoteServer {
    private Map<String, Remote> bindings
        = new HashMap<String, Remote>();

    public Remote lookup(String name)
        throws RemoteException, NotBoundException
    {
        synchronized (bindings) {
            Remote obj = bindings.get(name);
            if (obj == null) throw new NotBoundException(name);
            return obj;
        }
    }
    ...
}
```



Schéma de callback (rappel)

But : permettre au serveur d'appeler un client l'ayant contacté auparavant

- Augmenter l'asynchronisme : schéma publier/s'abonner :
 - 1 appel client → serveur avec retour immédiat (s'abonner)
 - 2 rappel serveur → client quand le service est exécuté (publier)
- Augmenter les interactions : le serveur peut demander au client des données complémentaires
- Programmation événementielle

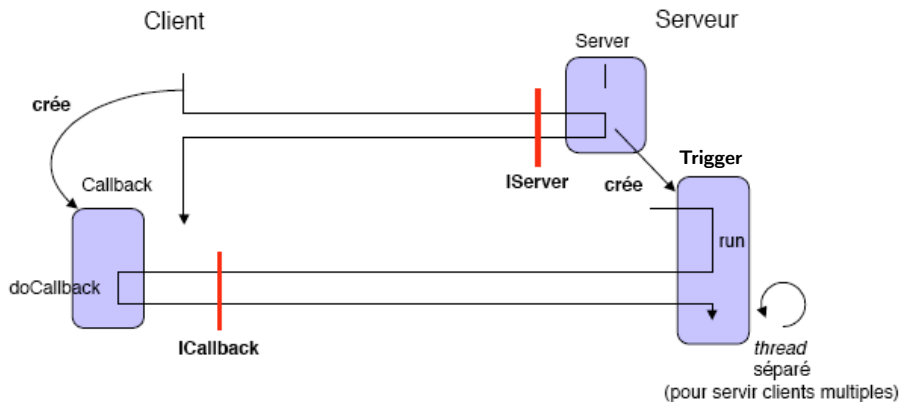
Principe

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

La relation client/serveur est conceptuelle, pas une relation d'usage !



Callback



Exemple 2 : callback

Interfaces

```
// les deux interfaces des objets appelés à distance

public interface ICallback extends Remote {
    public void wakeUp(String msg) throws RemoteException;
}

public interface IServer extends Remote {
    public void callMeBack(int time, String param,
                           ICallback callback) throws RemoteException;
}
```



Exemple 2 : callback

Trigger

```
public class Trigger extends Thread {  
    private int time;  
    private String param;  
    private ICallback callback;  
  
    // le callback cb sera appelé avec param dans time seconds  
    public Trigger(int time, String param, ICallback cb) {  
        this.time = time; this.param = param; this.callback = cb;  
    }  
  
    public void run() { // exécution comme thread  
        try {  
            Thread.sleep(1000*time); // attend time secondes  
            callback.wakeup(param);  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
}
```



Exemple 2 : callback

Serveur

```
import java.rmi.*;
import java.rmi.server.*;

public class Server extends UnicastRemoteObject
    implements IServer {
    public Server() throws RemoteException { }
    public void callMeBack(int time, String param, ICallback cb)
        throws RemoteException {
        Trigger action = new Trigger(time, param, cb);
        action.start();
    }

    public static void main(String[] args) throws Exception {
        Server server = new Server();
        Naming.rebind("ReveilMatin", server);
    }
}
```



Exemple 2 : callback

Callback & client

```
public class Callback extends UnicastRemoteObject
                                implements ICallback {
    public Callback() throws RemoteException { }
    public void wakeUp(String message) throws RemoteException {
        System.out.println(message);
    }
}

public class Client {
    public static void main(String[] args) throws Exception {
        Callback callback = new Callback();
        IServer serv = (IServer) Naming.lookup("ReveilMatin");
        serv.callMeBack(5, "coucou", callback);
        ...
    }
}
```



RMI : Conclusion

- Génération automatique des stubs et skeletons
- Sérialisation automatique (en général) par simple référence à l'interface `Serializable`
- Serveur multi-thread
- Sémantique au plus une fois (at-most-once)
- Problème : ramassage des objets accessibles à distance
- Attention à la fiabilité : `RemoteException`

