2.4 Manipulation de types structurés

Compléments sur RPC en C

2.4.1 Ajout d'une procédure: récapitulatif

Procédure simple, retourne un int

a) Interface rdict.x

```
program RDICTPROG {        /* nom du programme                    */
version RDICTVERS {         /* version                */
int INITIALISE(void)  = 1;             /* premiére procédure du programme   */
int INSERTION(string) = 2;             /* seconde procédure                  */
int SUPPRESSION(string) = 3;           /* troisiéme procédure               */
int CHERCHE(string) = 4;               /* quatriéme procédure               */
int NOMBRE(void) = 5;                  /* 5 ième  procédure   NEW           */
} = 1;                     /* numéro de la version du programme */
} = 0x30090949;    /* numéro de programme               */
```

b) Client: rdict.c (le main)

```
...
CommandeSuivante(&Commande, Mot)
switch (Commande) {
...
            case 'n':/* "nombre" */
            printf("Nombre de mots actuels dans le Dico: %d \n", nombre ());

...
  if (*Commande!='q' && *Commande!='I' && *Commande!='n'){ ...
```

c) Client: rdict_cif.c (convention)

```
/*--------------------------------------------------------------------
 * nombre - procedure interface client qui appelle nombre_1
 *------------------------------------------------------------------- */
int nombre()
{
  return *nombre_1(handle);
}
```

d) Serveur: rdict_server.c (convention)

```
int* nombre_1_svc(struct svc_req *rqstp) {
   static int  resultat;          /* static Indispensable, pourquoi ?*/
   /*
    * insert server code here
    */
   resultat = nombre();
   return(&resultat);
}
```

e) Serveur: rdict_srp.c (code effectif)

```
/*------------------------------------------------------------------------
 * nombre : retourne le nombre de mots dans le Dico
 *------------------------------------------------------------------------
 */
int
nombre ()
{
   return CompteurMots;
}
```

f) Code généré

Client: rdict_clnt.c

```
int *
nombre_1(CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
         if (clnt_call (clnt, NOMBRE, (xdrproc_t) xdr_void, (caddr_t) NULL,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

Serveur: rdict_svc.c

```
static int *
_nombre_1 (void  *argp, struct svc_req *rqstp)
{
      return (nombre_1_svc(rqstp));       /* Appel nombre_1_svc */
                                           /* du fichier rdict_server.c */
}
```

Avec:

```
static void
rdictprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
...
switch (rqstp->rq_proc) {
...
...
case NOMBRE:
            _xdr_argument = (xdrproc_t) xdr_void;
            _xdr_result = (xdrproc_t) xdr_int;
            local = (char *(*)(char *, struct svc_req *)) _nombre_1;
            break;
...

result = (*local)((char *)&argument, rqstp);        /* Appel _nombre_1 */

                              /* Envoie du resultat s'il y en a un, */
                              /* signale éventuellement une erreur de la couche transport */
if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
            svcerr_systemerr (transp);
}
```

2.4.2 Procédure qui retourne un char*

char* ---> string en RPC

a) Interface rdict.x

```
string SUIVANT(string) = 6; /*  6eme procedure   NEW           */
```

b) Client: rdict.c (le main)

```
case 'S':   /* Mot suivant un mot */
        printf("Le mot suivant %s est : %s.\n", Mot, suivant(Mot));
        break;
```

c) Client: rdict_cif.c (convention)

```
/*------------------------------------------------------------------------
 * suivant  - procedure interface client qui appelle suivant_1
 *------------------------------------------------------------------------
 */
char * suivant(char* Mot)
{
  return *suivant_1(Mot, handle);
}
```

Note:

on a bien le mapping

string (.x) <----> char*

dans les .c

d) Serveur: rdict_server.c (convention)

```
char ** suivant_1_svc(char *argp, struct svc_req *rqstp)
{
	static char * resultat;

	/*
	 * insert server code here
	 */
	resultat = (char *) suivant(argp);
	/* Cast (char *) sinon: Warning: assignment makes pointer from integer
without a cast */
	return(&resultat);
}
```

Note:

On retourne un char ** car on retourne toujours un pointeur sur le type effectif de retour tel que déclaré dans le .x:

string --> char * --> dans rdict_server char**

e) Serveur: rdict_srp.c (code effectif)

```
*-----------------------------------------------------------------------
 * suivant : retourne le mot suivant un mot donne dans le dictionnaire
 *----------------------------------------------------------------------- */
char *
suivant  (Mot)
char       *Mot;
{
  int i;
  for (i=0 ; i<CompteurMots; i++)
    if (strcmp(Mot, Dictionnaire[i]) == 0) {
      if (i+1 < CompteurMots)
         return Dictionnaire[i+1];
      else
         return "Aucun, dernier mot";
    }
  return "Aucun, Mot non trouve";
}
```

2.4.3 Procédure qui retourne un type complexe

On doit déclarer dans le .x les structures nécessaires.

Dans ce cas. le protocole va se charger du “marshalling/un-marshalling

Exemple d’une liste chaînée de char*

a) Interface rdict.x

```
const MAXMOTLONG = 255; /* Taille maxi mot */

typedef string chaine_mot<MAXMOTLONG>;
                    /* Un mot pour chainage, indispensable */
typedef struct  list_mots* suivant_list;
                    /* Suivant dans la liste, et aussi premier de la liste */
struct list_mots {
   chaine_mot mot; /* Le mot */
   suivant_list mot_suivant;  /* Pointeur vers le suivant */
};
...
      suivant_list TOUS () = 7;     /*  7eme procedure   NEW            */
...
```

Le .h généré:

```
typedef char *chaine_mot;

typedef struct list_mots *suivant_list;
struct list_mots {
      chaine_mot mot;
      suivant_list mot_suivant;
};
typedef struct list_mots list_mots;

#define TOUS 7
extern  suivant_list * tous_1(CLIENT *);
extern  suivant_list * tous_1_svc(struct svc_req *);
```

MAXMOTLONG est utilisé, pas de façon explicite dans le .h, mais dans la gestion des buffer, marshalling/un-marshalling.

Fichier rdict_xdr.c :

```
bool_t
xdr_chaine_mot (XDR *xdrs, chaine_mot *objp)
{
       register int32_t *buf;

        if (!xdr_string (xdrs, objp, MAXMOTLONG))
               return FALSE;
       return TRUE;
}
```

Vérification qu'une donnée est bien du type

chaine_mot<MAXMOTLONG>

b) Client: rdict.c (le main)

c) Client: rdict_cif.c (convention)

d) Serveur: rdict_server.c (convention)

e) Serveur: rdict_srp.c (code effectif)

.... A regarder en TD.

2.4.4 Gestion mémoire

Nous sommes en C, pas de GC !

Le protocole gère/utilise des buffers pour transfert vers la couche transport.

Pb.:
Le protocole ne doit pas avoir de fuite de mémoire à chaque appel !

Le protocole donne la possibilité de gérer la mémoire, et en particulier de la libérer.

Pour ce faire, il génère des fonctions pour libérer les résultats complexes des fonctions déclarée dans le .x :

```
#define TOUS 7
extern  suivant_list * tous_1(CLIENT *);
extern  suivant_list * tous_1_svc(struct svc_req *);
extern int rdictprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);
```

Une technique standard utilisée:

libérer la mémoire construite à l'appel N, lors de l'appel N+1

Voir en TD/TP.

CHAPITRE 3 Introduction à Java

RMI

3.1 Principes de base

RPC en Java

3.1.1 Fonctionnalités

RPC synchrone

Notation pointé pour l'appel distant:

remote_obj.foo (param1, param2);

Possibilité de retourner un résultat:

res = remote_obj.bar (param);

--> • - “Language centric"

Basé sur une interface marqueur

Passage par copie de graphes d'objets (DAG et Graphes avec cycles)

--> • Sérialisation, copie profonde

Sécurité, applets, et persistance (activable)

Génération de stub et de skeleton

Transport: TCP,
mais on peut utiliser un autre tranport (UDP, etc. avec Socket Factory).

3.1.2 Architecture:

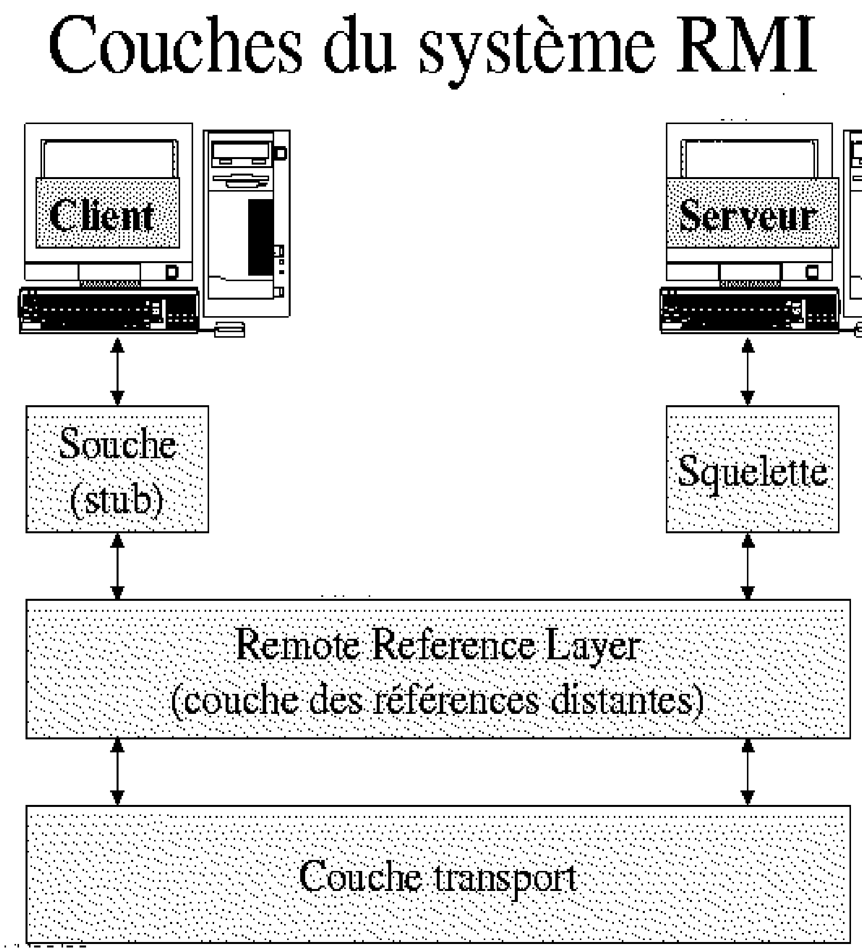

FIGURE 12 Couches du système RMI

Schéma classique, en pratique qq plus :

- La sérialisation Java
- Le byte code pour la portabilité
- Le protocol JRMP pour l'interopérabilité

RMI a son propre "directory" local (rpcbind)
rmiregistry

RMI a son propre "directory" global (LDAP, CDS):
Jini (Note: +/- utilisable aussi: JNDI)

3.1.3 Fonctionnement

Principes de fonctionnement à l'exécution

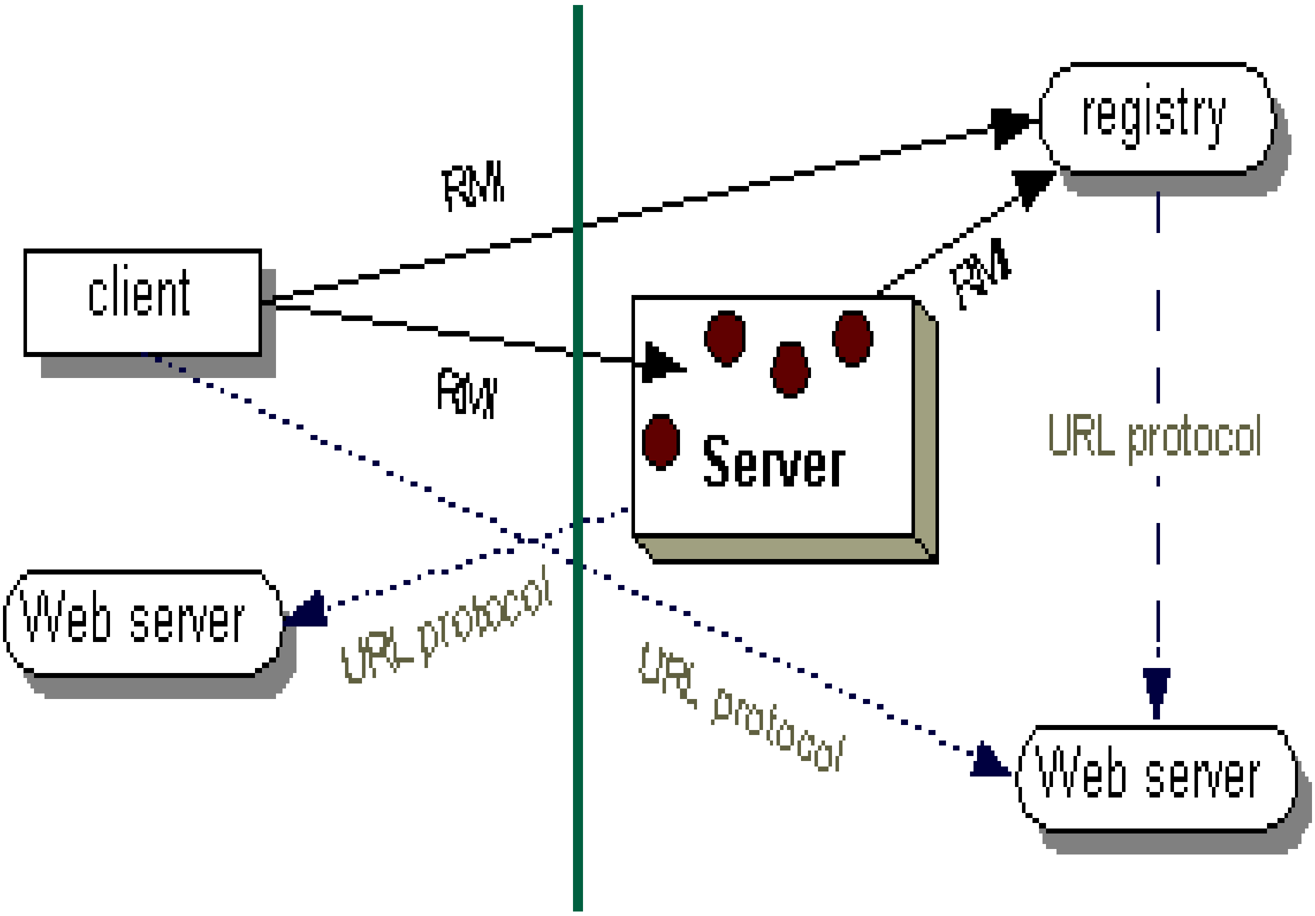

FIGURE 13 Entités du système RMI

Utilisation d'un “registry“ pour associer un nom à une référence vers un objet distant

Utilisation de “Web Serveur”
(en pratique juste un serveur HTTP)
pour charger si nécessaire le byte code d'un serveur vers un client, ou d'un client vers un serveur.

3.1.4 Différences avec RPC

Différences principales:

- syntaxe (notation pointé vs. procédure avec “handle“
- Uniquement TCP (pas vraiment UDP, pour l’instant)
- mais on peut utiliser des “secured sockets“

3.1.5 Modèle de la communication

Sémantique du passage des paramètres:

- les types primitifs (int, double, ...) sont toujours passés par valeur
- les objets, qui ne sont pas des objets “Remote“ (accessible à distance), sont passés par copie profonde.
  Sérialisation, des arbres, DAGs, graphes qq avec cycles.
- les objets ‘Remote“ sont passés par référence
  Cela signifie que l'on transmet en faite un stub sur l'objet distant.

--> exemple au tableau

3.1.6 Autres caractéristiques

+

Ramasse miettes réparti, mais ne gère pas les cycles

Intégration avec les exceptions

Chargement dynamique de code

-

Non-intégration, uniformisation avec les threads

Pas de politique de service simplement programmable pour un objet serveur RMI (FIFO, buffer, lecteur-rédacteur, etc.)

Il faut rendre l'objet synchronize si cela est nécessaire.

En pratique, ce sont les routines visibles par les interfaces Remote que l'on doit éventuellement rendre synchronize.

3.2 Classes et Interfaces

Pour l'utilisateur standard, l'API comporte essentiellement:

- une interface: java.rmi.Remote
- une classe: java.rmi.server.UnicastRemoteObject

Packages:

java.rmi

Package de base, avec Interface Remote, classe Naming, RMISecurityManager, et RemoteException

java.rmi.server

Implémentation: côté serveur, stub et skeletons, Transport et HTTP tunneling

java.rmi.registry

Repository. Mapping Nom --> ref. remote

java.rmi.dgc

Accès au GC réparti

java.rmi.activation

Persistance et activation (réveille) automatique des objets distants

3.2.1 Interface Remote

public interface Remote {}

Une interface de type marqueur

Sert juste à identifier les interfaces de type Remote:

implémentent directement ou indirectement cette interface

Une “interface distante“ doit obligatoirement respecter les contraintes suivantes:

- implémente l'interface java.rmi.Remote
- chaque routine doit déclarer l’exception java.rmi.RemoteException dans sa clause throw (ou un ancêtre: java.io.IOException, java.lang.Exception)
- Arguments et résultats des méthodes doivent être Serializable (implémente l'interface java.io.Serializable).
  Récursivement, sauf static or transient

Exemple d’interface distante (accessible à) :

```
public interface BankAccount extends java.rmi.Remote {
    public void deposit(float amount)
        throws java.rmi.RemoteException;
    public void withdraw(float amount)
        throws OverdrawnException, java.rmi.RemoteException;
    public float getBalance()
        throws java.rmi.RemoteException;    }
```

3.2.2 Classe UnicastRemoteObject

public class UnicastRemoteObject

extends RemoteServer

Permet d'implémenter une classe accessible à distante (un objet instance d'une ...).

Une classe accessible à distante implémente une ou plusieurs interfaces qui implémentent Remote

Exemple:

```
public class BankAccountImp
  extends UnicastRemoteObject
  implements BankAccount


  public BankAccountImp () throws RemoteException {
            super();
        }
public void deposit(float amount) throws RemoteException {
...
}
...
```

En fait, l'appel au constructeur par défaut de la super classe (super();) est fait automatiquement. Plus clair comme cela, mais non nécessaire.

La classe implémente les routines définies dans les interfaces remotes.

Une telle classe peut avoir des routines publiques autres que celles des interfaces remotes : elles ne sont accessibles que depuis la VM locale.

3.2.3 Schéma global

Schéma global des classes et interfaces RMI dans le package java.rmi :

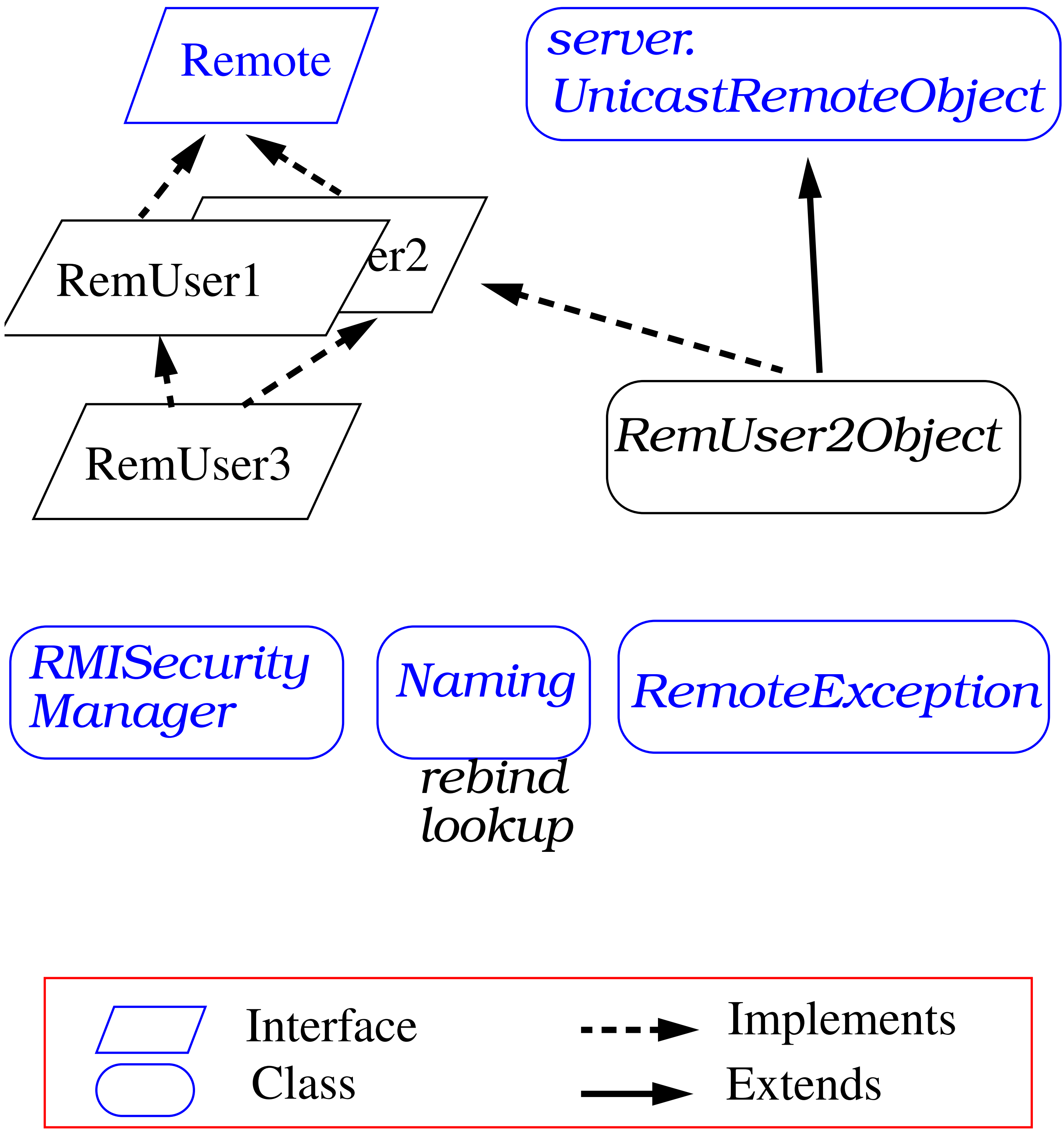

FIGURE 14 classes et interfaces RMI

3.2.4 Enregistrement d'un objet RMI

public class Naming

method rebind

ou

method bind

Permet d'enregistrer un objet accessible à distance dans un serveur (de nom) rmiregistry.

Exemple:

```
{
    // Create the server object
  HelloServer serverObject = new BankAccountImp ();
    // Register the server object
  Naming.rebind("BankAccount", serverObject );
    // Signal successful registration
  System.out.println("BankAccount Server bound in registry");
  }
```

Opération à réaliser en local uniquement.

Même si l'on peut écrire:

```
Naming.rebind("//clio.unice.fr/BankAccount", serverObject );
```

3.2.5 Lookup d'un objet RMI

public class Naming

method lookup

Permet, à distance, de récupérer une référence vers un objet RMI accessible à distance

A condition que celui-ci ait été enregistré dans un rmiregistry.

Exemple:

```
try {
        BankAccount remoteA = (BankAccount) Naming.lookup(
                                              "//clio.unice.fr/BankAccount");
        float r= remoteA.getBalance ();
}
catch (NotBoundException error) {
        error.printStackTrace();
    }
{
```

Note: attention!!!

On doit connaître la machine où se trouve l'objet RMI.

3.3 Outils et compilateurs

rmic, rmiregistry (rmid : plus tard)

3.3.1 rmic

Permet de générer les Stubs et les Skeletons pour les classes implémentant l'interface Remote

On doit donner le nom complet de la classe, avec le nom complet du package.

Example:

% rmic BankAccountImp

Crée les .class:

BankAccountImp_Skel.class
BankAccountImp_Stub.class

NAME
rmic - Java RMI stub compiler
SYNOPSIS
rmic [-classpath path] [-d directory] [-depend] [-g]
[-keepgenerated] [-nowarn] [-O] [-show]
[-verbose] [-v1.2] [-vcompat] package-qualified-class-names

-keepgenerated : Garde les sources
-v1.2 : La version 1.2 de Java ne nécessite plus les skeletons, mais par défaut elle les garde par compatibilité avec Java 1. -v1.2 ne les gardent pas.
-vcompat : permet de générer des stubs/skeletons compatible avec 1.1 et 1.2 protocols.

3.3.2 rmiregistry

Démarre le registry (service de nom) des objets distants Java RMI, sur la machine en cours, avec un certain numéro de port.

Par défaut: le port est le 1099.
Utiliser un port supérieur à 1024

Example:

% rmiregistry &

ou

% rmiregistry 2001 &

NAME

rmiregistry **- Java remote object registry**

SYNOPSIS

rmiregistry [port]

DESCRIPTION

The Java rmiregistry command creates and starts a remote object registry on the specified port on the current host. If port number is omitted, the registry is started on port 1099. The rmiregistry command produces no output and is typically run in the background. For example:

example% rmiregistry &