RPC, RMI et JoCaml

Louis Mandel
louis.mandel@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11

IFIPS
Cycle ingénieur de la filière étudiant
année 2008/2009

Appels de procédures à distance

- RPC = Remote Procedure Call
  - appel d'une procédure sur une machine A
  - exécution du code de la procédure sur une machine B
  - récupération des résultats sur la machine A

- Modèle de concurrence de haut niveau
  - concevoir des systèmes répartis "comme" des systèmes centralisés

- Une alternative aux communications basées sur les entrées/sorties

- Exemple d'utilisation : NFS

Principe des RPC

- Le client et le serveur se mettent d'accord sur l'interface des fonctions qui vont être exécutées sur le serveur
- Serveur :
  - attend que des clients appellent une fonction
  - reçoit l'appel sous forme d'un message
  - réalise l'exécution de la procédure appelée
  - transmet le résultat par un message
- Client :
  - réalise un appel de procédure
  - le transforme en un appel de procédure distant en envoyant un message au serveur
  - attend le résultat sous forme d'un message
  - retourne les paramètres résultats comme dans un appel de procédure

Transmission de données

- La représentation des données dépend des machines
  - transmission scalaires
  - transmission des tableaux
  - transmission des valeurs structurées
- Protocole XDR
  - normalisation des données : sérialisation/désérialisation
  - type `xdrproc_t` : pointeur vers une fonction de traitement des données
  - exemple de fonctions :

```
bool_t xdr_char(XDR *, char *);
bool_t xdr_int(XDR *, int *);
bool_t xdr_float(XDR *, float *);
bool_t xdr_array(XDR *xdr, char *tableau, u_int *pTaille,
                 u_int tailleMax, u_int tailleElem, xdrproc_t fct);
boot_t xdr_reference(XDR *xdr, void **ptr, u_int taille, xdrproc_t fct);
```

- Utilisation de langages sans manipulation de pointeurs

Structure d'une application

cf. schéma

Utilisation d'un IDL

IDL = Interface Description Language

- spécification commune entre le client et le serveur
- définition des fonctions : type et nature des paramètres

Utilisation de l'IDL pour générer automatiquement :

- le talon client (ou proxy, ou stub)
- le talon serveur (ou squelette, ou skeleton, ou stub)

RMI : Remote Method Invocation

- Application des RPC en Java
- Le serveur rend disponible des objets à des processus distants
- Un objet distribué se manipule comme tout autre objet Java
- IDL : les interfaces (contrats) sont des interfaces Java

Définition de l'interface (Compteur.java)

```
import java.rmi.*;
public interface Compteur extends Remote {
    public void incr() throws RemoteException;
    public int get() throws RemoteException;
}
```

- Définitions des méthodes accessibles aux clients
- Hérite de `java.rml.Remote`
- Les méthodes peuvent lever l'exception `RemoteException`
- Compilation : `javac Compteur.java`

Serveur : implantation de l'interface (CompteurImpl.java)

```
import java.rmi.*;
import java.rmi.server.*;
public class CompteurImpl extends UnicastRemoteObject implements Compteur {
    private int cpt;
    public CompteurImpl(int n) throws RemoteException { cpt = n; }
    public synchronized void incr() throws RemoteException{ cpt++; }
    public int get() throws RemoteException{ return cpt; }
}
```

- Implantation de l'objet qui sera exécuté sur le serveur
- Hérite de `UnicastRemoteObject` : définition d'objet distant et exportation automatique
- Compilation :

  ```
  javac CompteurImpl.java
  rmic CompteurImpl
  ```

- Génération de `CompteurImpl_Stub.class`

Serveur de noms et gestionnaire de sécurité

- Serveur de noms
  - initialisation de la communication entre le client et le serveur
  - association d'un nom avec un objet :

```
void Naming.rebind(String nom, Remote obj)
void Naming.unbind(String nom)
Remote Naming.lookup(String nom)
```

  - exécution du serveur de nom : `--> rmiregistry &`
- Gestionnaire de sécurité
  - lancement

```
if(System.getSecurityManager() == null) {
        System.setSecurityManager (new RMISecurityManager());
}
```

  - configuration des droits (`java.policy`) :

```
grant { permission java.security.AllPermission; };
```

Serveur : programme principal (CompteurServeur.java)

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
public class CompteurServeur{
    public static void main(String[] args) throws Exception {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        CompteurImpl compteur = new CompteurImpl(0);
        Naming.rebind("cpt", compteur);
    }
}
```

- Compilation : `javac CompteurServeur.java`
- Exécution :

  `java -Djava.security.policy=java.policy CompteurServeur`

Client (CompteurClient.java)

```
import java.rmi.*;
public class CompteurClient{
    public static void main(String[] args) throws Exception{
        if(args.length == 0){
            System.err.println("bad usage"); System.exit(2);
        }
        Compteur c = (Compteur) Naming.lookup("rmi://"+args[0]+"/cpt");
        c.incr();
        System.out.println(c.get());
    }
}
```

- L'objet c est distant
- Compilation : `javac CompteurClient.java`
- Exécution :

  `java -Djava.security.policy=java.policy CompteurClient`

RPCL

Appels de Procédures à Distance en C

RPCL : Mini-NFS (mnfs.x)

```
const TAILLEMAX=1024;
struct ecriture {
  string fichier<256>;
  unsigned long offset;
  char tampon<TAILLEMAX>;
  int nombre; };
struct lecture {
  string fichier<256>;
  unsigned long offset;
  int nombre; };
struct reponseLecture {
  int nombreLu;
  char tampon<TAILLEMAX>; };
program MNFS_PROG {
  version MNFS_V3 {
    int  MNFSCREATE(string) = 1;
    reponseLecture MNFSREAD(lecture) = 2;
    int MNFSWRITE(ecriture) = 3;
  } = 3;
} = 0x222222ff;
```

rpcgen

- La compilation d'un fichier RPCL génère des fichiers C
- Dans notre exemple (rpcgen -C -a mnfs.x) :
  - `mnfs.h` : définition des types et déclarations des filtres XDR, des prototypes des fonctions, etc.
  - `mnfs_xdr.c` : filtres XDR
  - `mnfs_svc_c` : programme principal du serveur
  - `mnfs_clnt.c` : talon du client
  - `mnfs_server.c` : code à trous pour les fonctions du serveur
  - `mnfs_client.c` : exemple de client

JoCaml

JoCaml

- Langage de programmation avec concurrence et distribution
- Extension de OCaml : langage fonctionnel, d'ordre supérieur, fortement typé avec inférence de type
- Exemple de programme Caml :

```
let f x =
  x + 1

let hello nom =
  "Bonjour " ^ nom

let main =
  print_endline (bonjour "Toto")
```

RPC en JoCaml : serveur (somme_s.ml)

```
let wait =
  def x () & y () = reply to x
  in x

def somme (x, y) =
  reply x+y to somme

let main =
  Join.Ns.register Join.Ns.here "somme" (somme: int * int -> int);
  Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
  wait ()
```

- Le mot clé def introduit un canal : fonction qui attend un argument pour être calculée sur son site de définition
- Compilation `jocamlc -o s somme_s.ml`

RPC en JoCaml : client (somme_c.ml)

```
let server =
  let server_addr = Unix.gethostbyname Sys.argv.(1) in
  Join.Site.there (Unix.ADDR_INET(server_addr.Unix.h_addr_list.(0), 12345))

let ns = Join.Ns.of_site server

let somme = (Join.Ns.lookup ns "somme": int * int -> int)

let main =
  let x = somme (1, 2) in
  Printf.printf "%d\n" x
```

- Compilation `jocamlc -o c somme_c.ml`

Exécution concurrente (ex1.ml)

```
def print_message(m) =
  while (true) do print_endline m done; 0

def print_par() =
  print_message "Hello"
  &
  print_message "Bye"

let main =
  spawn (print_par());
  Thread.delay(2.0)
```

- Un canal sans `reply/to` est un canal asynchrone
- L'opérateur `&` représente la composition parallèle
- `spwan` crée un nouveau thread

Synchronisation (ex2.ml)

- Les join-patterns : attente de plusieurs messages

```
def print1(m1) & print2(m2) =
  print_endline (m1^m2);
  0

let main =
  spawn print1("<");
  Thread.delay(1.0);
  spawn print2(">");
  Thread.delay(1.0)
```

Synchronisation (ex3.ml)

- Les join-patterns et les cannaux synchrones

```
def print1(m1) & print2(m2) & get() =
  print_endline (m1^m2);
  reply m1^m2 to get

let main =
  spawn print1("<");
  Thread.delay(1.0);
  spawn print2(">");
  print_endline (get())
```

Compteur : serveur (compteur_s.ml)

```
def incr () & cpt (x) =
  cpt (x + 1)
  & reply () to incr

or  get () & cpt (x) =
  cpt (x)
  & reply x to get

let main =
  spawn cpt (0);
  Join.Ns.register Join.Ns.here "incr" (incr: unit -> unit);
  Join.Ns.register Join.Ns.here "get" (get: unit -> int);
  Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
  wait ()
```

Compteur : client (compteur_c.ml)

```
let server =
  let server_addr = Unix.gethostbyname Sys.argv.(1) in
  Join.Site.there (Unix.ADDR_INET(server_addr.Unix.h_addr_list.(0), 12345))

let ns = Join.Ns.of_site server

let incr = (Join.Ns.lookup ns "incr": unit -> unit)
let get  = (Join.Ns.lookup ns "get": unit -> int)

let main =
  incr ();
  Printf.printf "%d\n" (get ())
```

Mémoire partagée (ex4.ml)

```
let new_cell v_init =
  def state(v) & get() = state(v) & reply v to get
   or state(v) & set(v') = state(v') & reply () to set
  in
  spawn state(v_init);
  (set, get)

let main =
  let set, get = new_cell(0) in
  spawn (set(4012); 0);
  print_int(get())
```

Pile partagée (ex5.ml)

```
let new_lifo () =
  def state(v::l) & get() = state(l) & reply v to get
   or state(l) & put(v) = state(v::l) & reply () to put
  in
  spawn state([]);
  (put, get)

let main =
  let put, get = new_lifo() in
  spawn (put(4012); 0);
  print_int(get())
```

Buffer (ex6.ml)

```
let new_fifo () =
  def state(xs, y::ys) & get() = state(xs, ys) & reply y to get
   or state(_::_ as xs, []) & get() =
     state([], List.rev xs) & reply get() to get
   or state(xs, ys) & put(v) = state(v::xs, ys) & reply () to put
  in
  spawn state([],[]);
  (put, get)

let main =
  let put, get = new_fifo() in
  spawn (put(4012); 0);
  print_int(get())
```

Traitement des pannes

Exemple du ray tracer