

Programmation Répartie et Architecture n 1/3

Denis Caromel

Université de Nice Sophia Antipolis

UFR Sciences

Département Informatique

Master 1 Informatique,

ALR: Architectures Logicielles Réparties

ALR 1

Programmation Répartie

caromel@unice.fr

<http://www.inria.fr/oasis/Denis.Caromel/>

Version 06

Préambule

Volume:

- 8 x 2H00 de cours, 8 x 2H00 de TD machine
mini-projet sur machine

Plan du Cours

- Programmation répartie de type RPC:

Principes, Modèle et Méthode

Architectures clients-serveurs avec RPC

- RPC dans les langages classiques

C, et rpcgen

- RPC dans les langages objets (Java RMI)

Clients et serveurs, Callbacks,

RMI et les applets,

Interactions avec les threads

Chargement dynamique de code

Polymorphisme : client-serveurs génériques

- Composants et Architecture 3 tiers:

Principes des architecture 3 tiers:

1. 1er tiers: client
2. 2ème tiers: Objets métiers, etc.

3. 3ème tiers: Bases de données, facturation

Architectures à base de composants:

1. rôle des JavaBeans, JSP, servlet
2. principes des EJB, container et transactions
3. aspects concurrence et transactions transparentes

Utilisation et rôle dans les applications web:

1. Commerce Électronique,
2. Liens avec HTTP, HTTPS, CGI, JDBC

- Principes et comparaisons avec d'autres systèmes:

CORBA,

DCOM/ActiveX,

- Conception, paternes pour la program. répartie:

Acceptor et Connector,

Thread par session ou par requête,

Objets actifs, Services asynchrones, migration (mobilité), etc.

- Les web Services:

Idées et Principes

Rôle de WSDL, UDDI, SOAP, XML, (C#)

Comparaison avec différentes plate-formes

Notation

- Examen + Mini-Projet

Examen: dates à voir, en Mai ?

Mini-projet :

Structure du mini-projet:

Une application simple 3 1/3,
par groupe de 4 étudiants (éventuelle-
ment 2 ou 3).

En Java. A voir

(éventuellement coordonné avec
d'autres enseignements)

Dates pour 2005-2006:

du 27 février au 31 mars
(Semaines 5 à 8 du cours)

Voir web:

<http://www.inria.fr/oasis/Denis.Caromel/ProgRpt/>

CHAPITRE 1	Introduction aux RPC	14
1.1	Architecture actuelles: cluster de SMPs.	15
1.2	Problème général et RPC	17
1.2.1	Propriétés souhaitables des langages/modèles	17
1.2.2	Schéma de principe.	18
1.2.3	Communications + Exécution: RPC	19
1.2.4	Différentes vues d'un RPC	21
1.2.5	Opérations successives d'un RPC	23
1.2.6	Traitement des RPC par le serveur	26
1.2.7	Interactions Client-Serveur	28
1.3	Composantes et variations d'un RPC	31
1.3.1	Forme, syntaxe	31
1.3.2	Passage des paramètres	31
1.3.3	Représentation des données	31
1.3.4	Sémantique de l'appel	32
1.3.5	Gestion des erreurs	32
1.3.6	Code sur machine distante	32
1.3.7	Découverte et nommage	33
1.3.8	Terminaison et GC distribué	33
1.3.9	Persistance des "objets", Tolérance aux fautes	33
1.3.10	Sécurité	34
1.3.11	Autres aspects et concepts	35
1.4	Conclusion et Evolutions en cours.	36
CHAPITRE 2	RPC en C	38
2.1	Principes	39
2.1.1	Généralités	39
2.1.2	Serveur de noms RPC : portmapper	40
2.2	Mise en oeuvre	43
2.2.1	Fichiers de base	43
2.2.2	Exemple d'utilisation	45
2.2.3	Démarrage et configuration	46
2.3	Exemple	49
2.3.1	Principes	49
2.3.2	Interface .x	52
2.3.3	Exemple de client	52
2.3.4	Fichier intermédiaire client: cif.c	54
2.3.5	Exemple de serveur	55
2.3.6	Conversions XDR	58
2.3.7	Header file: rdict.h	59
2.3.8	Stubs client	60
2.3.9	Stubs serveur	61
2.4	Manipulation de types structurés	62
2.4.1	Ajout d'une procédure: récapitulatif	62
	a) Interface rdict.x	62

b) Client: rdict.c (le main)	62
c) Client: rdict_cif.c (convention)	62
d) Serveur: rdict_server.c (convention)	63
e) Serveur: rdict_srp.c (code effectif)	63
f) Code généré	63
2.4.2 Procédure qui retourne un char*	65
a) Interface rdict.x	65
b) Client: rdict.c (le main)	65
c) Client: rdict_cif.c (convention)	65
d) Serveur: rdict_server.c (convention)	66
e) Serveur: rdict_srp.c (code effectif)	66
2.4.3 Procédure qui retourne un type complexe	67
a) Interface rdict.x	67
b) Client: rdict.c (le main)	68
c) Client: rdict_cif.c (convention)	68
d) Serveur: rdict_server.c (convention)	68
e) Serveur: rdict_srp.c (code effectif)	68
2.4.4 Gestion mémoire	69

CHAPITRE 3 Introduction à Java RMI 70

3.1 Principes de base	71
3.1.1 Fonctionnalités	71
3.1.2 Architecture:	72
3.1.3 Fonctionnement	73
3.1.4 Différences avec RPC	74
3.1.5 Modèle de la communication	74
3.1.6 Autres caractéristiques	75
3.2 Classes et Interfaces	76
3.2.1 Interface Remote	77
3.2.2 Classe UnicastRemoteObject	78
3.2.3 Schéma global	79
3.2.4 Enregistrement d'un objet RMI	80
3.2.5 Lookup d'un objet RMI	81
3.3 Outils et compilateurs	82
3.3.1 rmic	82
3.3.2 rmiregistry	83
3.4 Méthode type de développement	84
3.4.1 Interfaces distantes	84
3.4.2 Classes implémentant les Interfaces Distantes: serveur(s)	84
3.4.3 Clients utilisant les objets distants	85
3.4.4 Compiler les sources	85
3.4.5 Précompilation: création Stubs/Skeletons	85
3.4.6 Lancer le rmiregistry	85
3.4.7 Installer le serveur	86
3.4.8 Lancer le ou les clients	86
3.5 Exemple simple: Remote Hello	87

3.5.1	Remote Interface	87
3.5.2	Serveur HelloWorld	88
3.5.3	Enregistrement dans le rmiregistry	89
3.5.4	Un client de HelloWorld	91
3.5.5	Compilation, génération, registry	92
	a) Côté Serveur	92
	b) Côté Client	93
3.6	Implémentation	95
3.6.1	Stub	95
3.6.2	Skeleton	97
 CHAPITRE 4 Techniques et utilisations avancées de RMI		101
4.1	Sécurité et chargement dynamique de code	102
4.1.1	Rappels et cadre	102
4.1.2	Security Manager	103
4.1.3	Security file et RMI	104
4.1.4	Applets signés et RMI	104
4.1.5	codebase et RMI	105
4.1.6	CLASSPATH et rmiregistry	109
4.2	Applets et RMI	111
4.2.1	Principes	111
4.2.2	Code et principes des classes	113
	a) Remote interface Hello	113
	b) Serveur classe (HelloImpl) et main	114
	c) Applet cliente: HelloApplet	115
	d) Page HTML: hello.html	116
4.2.3	Compilation et déploiement	117
	a) Placement des fichiers, serveurs HTTP	117
	b) Compilation	118
	c) Placement de l'applet et CLASSPATH	119
	d) RMI registry, serveur et applet	119
	e) Exécution de l'applet	121
4.3	Polymorphisme et RMI	122
4.3.1	Principe et implications	122
4.3.2	Retour de résultat	123
4.3.3	Passage de paramètres	124
4.4	Exemple RMI avec polymorphisme	125
4.4.1	Principe des 2 interfaces	125
4.4.2	Implémentation du serveur	126
4.4.3	Un client: compute Pi	127
4.5	Callback, synchro, multithread, et RMI	131
4.5.1	Model	131
4.5.2	Appel asynchrones	132
4.5.3	Call back	133
4.6	Ramasse miettes réparti	135

4.6.1	Principles.....	135
4.6.2	Main actions.....	136
4.6.3	Leasing: Time-To-Live stubs.....	137
4.6.4	Time-to-live stubs vs. counters.....	138
4.6.5	API with RMI DGC.....	139
	a) Property: java.rmi.dgc.leaseValue.....	139
	b) Listener on “no more remote reference”.....	140
	c) Stopping a Remote Object.....	140
CHAPITRE 5 Objets Réparties et Persistance.....		141
5.1	Objets activables.....	142
5.1.1	Class java.rmi.activation.Activable.....	143
5.1.2	Implémentation d'une classe Activable.....	145
	a) Importation.....	145
	b) Etendre Activable.....	145
	c) Constructeur spécial.....	145
	d) Remote interface.....	146
5.1.3	Classe de set-up.....	147
5.1.4	Implémentation d'une classe "setup".....	148
	a) Importation.....	148
	b) SecurityManager.....	148
	c) Créer un groupe d'activation (instance).....	148
	d) Créer un descripteur de la classe activable.....	149
	e) Enregistre la classe dans le rmid.....	150
	f) Enregistre le stub dans le rmiregistry.....	151
5.1.5	Démon d'activation: rmid.....	152
5.1.6	Schéma global.....	154
5.1.7	Compilation et exécution.....	155
5.1.8	Exemple.....	156
	a) classe Activable.....	156
	b) class Setup.....	157
5.1.9	java.rmi.activation.....	159
CHAPITRE 6 Composants et Architecture 3 1/3.....		161
6.1	Introduction.....	162
6.1.1	Des objets aux composants.....	162
6.1.2	Définitions.....	163
6.1.3	Exemples.....	164
6.1.4	Characteristics -- How ?.....	164
6.1.5	My Definition of Components.....	165
6.1.6	Component Orientedness.....	166
6.1.7	Architecture.....	167
6.1.8	Composants et Introspection.....	169
6.1.9	Classification des solutions industriels.....	169
6.2	Java Beans.....	172
6.2.1	Principes et concepts.....	172
6.2.2	Exemple.....	175

6.2.3	Propriétés	176
6.2.4	Événements.....	180
6.2.5	Persistence.....	182
6.2.6	La BeanBox	183
6.2.7	Caractéristiques explicites: BeanInfo	184
6.2.8	Évaluation des JavaBeans	185
6.2.9	Exemple de bean: Juggler.....	186
	a) Juggler.java.....	186
	b) JugglerBeanInfo.java	195
	c) Images et icônes	196
6.2.10	Conclusion on JavaBeans.....	197
6.3	Principes des architectures n 1/3.....	199
6.3.1	Introduction.....	199
6.3.2	Architectures typiques 3 1/3	200
6.3.3	Objectifs	202
6.3.4	Avantages	202
6.3.5	Inconvénients	203
6.3.6	Couches détaillés	205
6.3.7	Outils et techniques standards.....	206
6.4	Enterprise Java Beans: EJB	208
6.4.1	Introduction.....	208
6.4.2	Principes et concepts.....	208
6.4.3	Plate-forme EJB	211
6.4.4	Rôles et contrats	213
6.4.5	Architecture à l'exécution	214
6.4.6	Autres aspects	215
	a) Session et Entity Beans.....	215
6.4.7	Autres aspects	216
	a) Message Driven Beans	216
	b) Transaction distribuées.....	217
	c) Sécurité.....	217
	d) Concurrence	217
	e) Répartition	217
6.4.8	Un exemple simple	218
	a) EJB Remote Interface.....	218
	b) L'interface Home	219
	c) La classe qui implémente l'EJB	220
	d) Création d'un fichier ejb-jar	224
	e) Déploiement du bean	227
	f) Le code du client.....	227
	g) Compilation et exécution du client.....	230
6.5	Récapitulatif EJB	231
6.5.1	Classes et interfaces EJB	231
6.5.2	Conclusion on EJB	232
6.6	Web vs P2P	234

CHAPITRE 7	Corba.....	238
7.1	Principes, objectifs, IDL, Mapping C++, etc.	239
7.2	Conclusion sur CORBA.....	240
7.2.1	Caractéristiques générales.....	240
7.2.2	Choisir un mapping (IDL --> ?)	243
7.2.3	Versions successives	244
	a) CORBA 1	244
	b) CORBA 2	244
	c) CORBA 3	244
7.3	Détails sur CCM.....	247
7.3.1	Principes	247
7.3.2	IDL 3 spécifications:.....	250
	a) Attributes:	251
	b) Facets	252
	c) Receptacles:	253
	d) Events:	254
7.3.3	Framework de développement: CIF	257
CHAPITRE 8	Conception et paternes pour la programmation répartie: ..	259
8.1	Principes et objectifs	260
8.2	Exemples de paterns.....	261
8.2.1	Acceptor et Connector	261
8.2.2	Thread par session.....	261
8.2.3	Thread par requête	261
8.2.4	Objets actifs vs. Moniteur.....	261
8.2.5	Services asynchrones	261
8.2.6	Migration (mobilité)	261
CHAPITRE 9	Introspection et Reflexion Java.....	262
9.1	Principes et objectifs	263
9.1.1	263
9.1.2	263
9.1.3	263
9.1.4	263
9.1.5	263
CHAPITRE 10	Jini.....	265
10.1	Principes et objectifs	266
10.1.1	266
10.1.2	266
10.1.3	266
10.1.4	266
10.1.5	266

FIGURE 1	Cluster de SMPs	15
FIGURE 2	Principe de la distribution	18
FIGURE 3	Principe d'un RPC	19
FIGURE 4	Vue temporelle d'un RPC	21
FIGURE 5	Vue en couche d'un RPC	22
FIGURE 6	Un serveur séquentiel	26
FIGURE 7	Un serveur parallèle	27
FIGURE 8	Démarrage d'un serveur	28
FIGURE 9	Démarrage d'un client	29
FIGURE 10	Interactions Client / Serveur	30
FIGURE 11	Principes et fichiers rpcgen	44
FIGURE 12	Couches du système RMI	72
FIGURE 13	Entités du système RMI	73
FIGURE 14	classes et interfaces RMI	79
FIGURE 15	Remote Hello world	87
FIGURE 16	Téléchargement de stub en RMI	106
FIGURE 17	Téléchargement due au polymorphisme	107
FIGURE 18	Téléchargement due au polymorphisme	111
FIGURE 18	Exécution de l'applet Hello World	111
FIGURE 19	Architecture d'un RPC avec une applet RMI	112
FIGURE 20	Téléchargement due au polymorphisme	125
FIGURE 20	Client-Serveur générique	125
FIGURE 21	Téléchargement due au polymorphisme	127
FIGURE 21	Un client : compute Pi	127
FIGURE 22	classes et interfaces RMI: objets activables	144
FIGURE 23	Schéma de fonction. des objets activables	154
FIGURE 24	Architecture d'un système à composants	168
FIGURE 25	Modificatin de propriétés	179
FIGURE 26	Principes des événements Java, JavaBeans	180
FIGURE 27	La BeanBox des JavaBeans	183
FIGURE 28	Images du Jongleur	196
FIGURE 29	Une architecture 3-tiers	200
FIGURE 30	Une architecture 4-tiers avec serveur http	201
FIGURE 31	Cout 2 tiers vs. 3 tiers	204
FIGURE 32	Principes du modèle d'exécution	210
FIGURE 33	Plate-forme EJB	212
FIGURE 34	Rôles dans la production d'une application	213
FIGURE 35	Architecture à l'exécution	214
FIGURE 36	Classes et interfaces EJB	231
FIGURE 37	Composants Corba CCM,	248
FIGURE 38	Exemple de composition	249
FIGURE 39	Role des specifications CIDL	257
FIGURE 40	Génération des stub/proxy des composansts	258

CHAPITRE 1 Introduction aux RPC

1.1 Architecture actuelles: cluster de SMPs

Préambule:

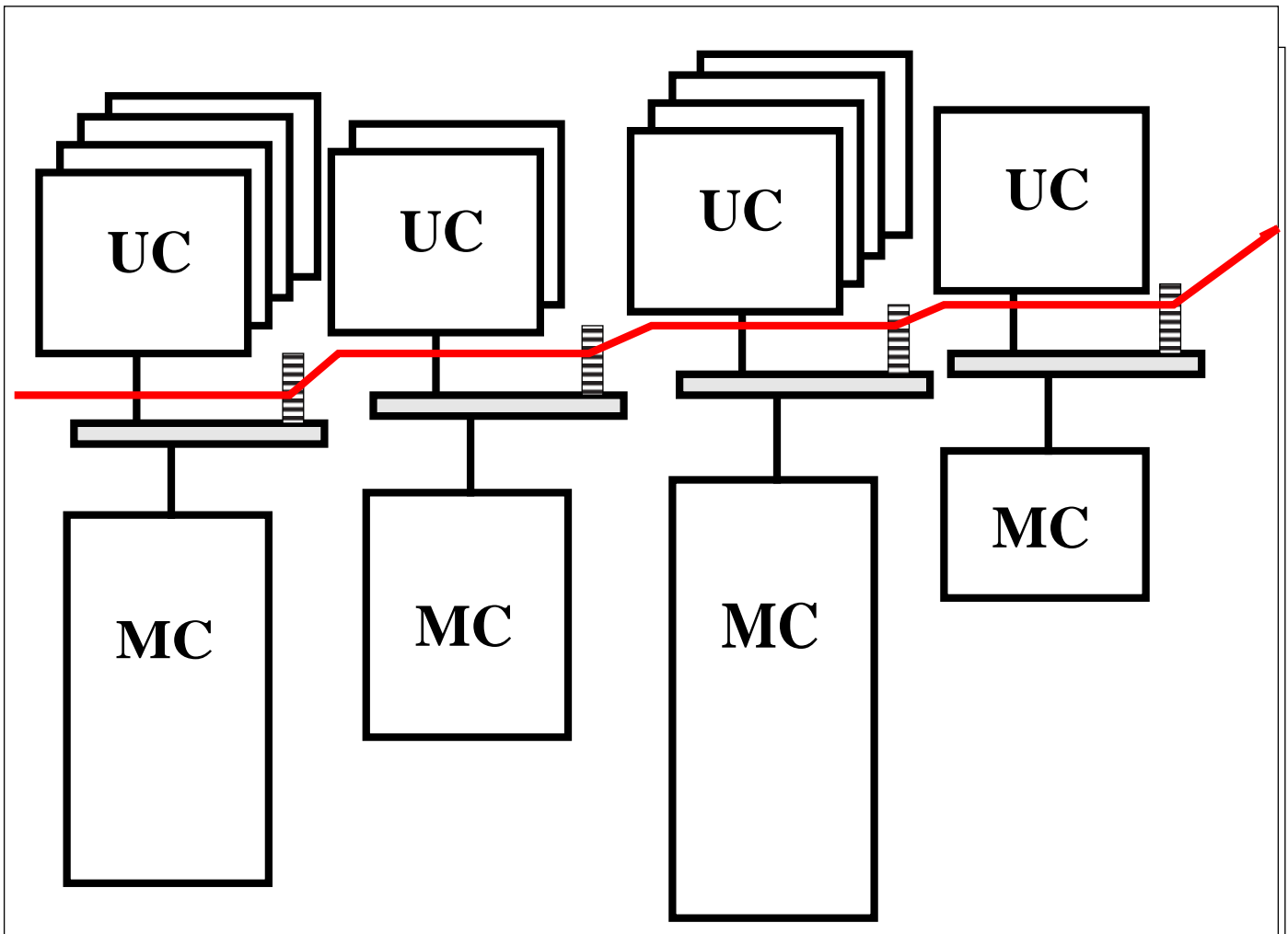


FIGURE 1 Cluster de SMPs

Des **OS et systèmes classiques**

configuration d'un LAN d'entreprise

au

Commerce Électronique

les gros serveurs de transactions ont de plus en plus cette architecture.

Difficile à programmer:

latence et bande passante varient selon la configuration et le placement des processus.

Architecture très hétérogène, avec

concurrence,

parallélisme,

répartition.

Utilisation importante des RPC.

Vers le “Grid” (Grille de calculs et de données)

1.2 Problème général et RPC

1.2.1 Propriétés souhaitables des langages/modèles

Une bonne abstraction des primitives de bases des architectures

- trop concret, bas niveau: obsolète rapidement
- trop haut niveau: performances difficiles à obtenir

Nécessité de compromis, en particulier par rapport à un certain nombre de propriétés.

Indépendant de l'architecture

Exécuter le programme sur des architectures différentes, *sans modifier le source*

Le modèle, théoriquement, doit s'abstraire de toutes les caractéristiques spécifiques à certaines architectures.

Abstrait

Cacher certains aspects du parallélisme
e.g. : synchronisations

Une des abstractions: **Remote Procedure Call**
à comparer aux: **sockets**.

1.2.2 Schéma de principe

L'UC de la machine M1 ne peut pas exécuter des opérations sur la MC de M2.

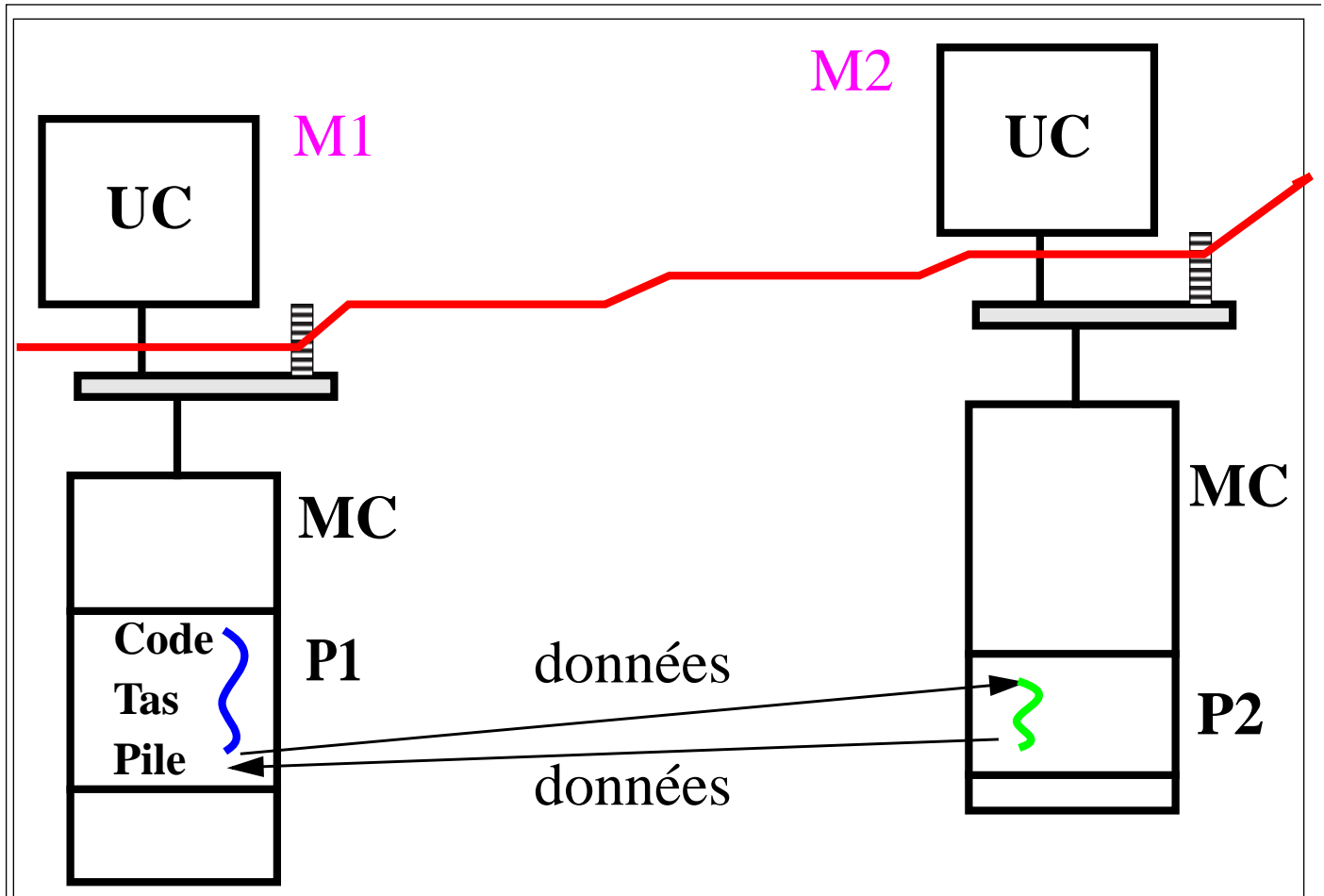


FIGURE 2 Principe de la distribution

Depuis le processus P1 de la machine M1, on a besoin d'exécuter un morceau de code sur la machine M2, evt. dans le cadre du processus P2. On a également besoin de transmettre des informations (données) de P1 vers P2. Eventuellement, P1 souhaite récupérer des données ou un acquittement à la fin.

Avant, ou programmation bas niveau: sockets

1.2.3 Communications + Exécution: RPC

Remote Procedure Call :

Depuis un processus (P1),
sur une machine (M1),
on va exécuter la procédure **foo**,
avec des paramètres **param**
sur une autre machine (M2),
(evt. dans un processus (P2))

on récupère éventuellement un résultat **res**

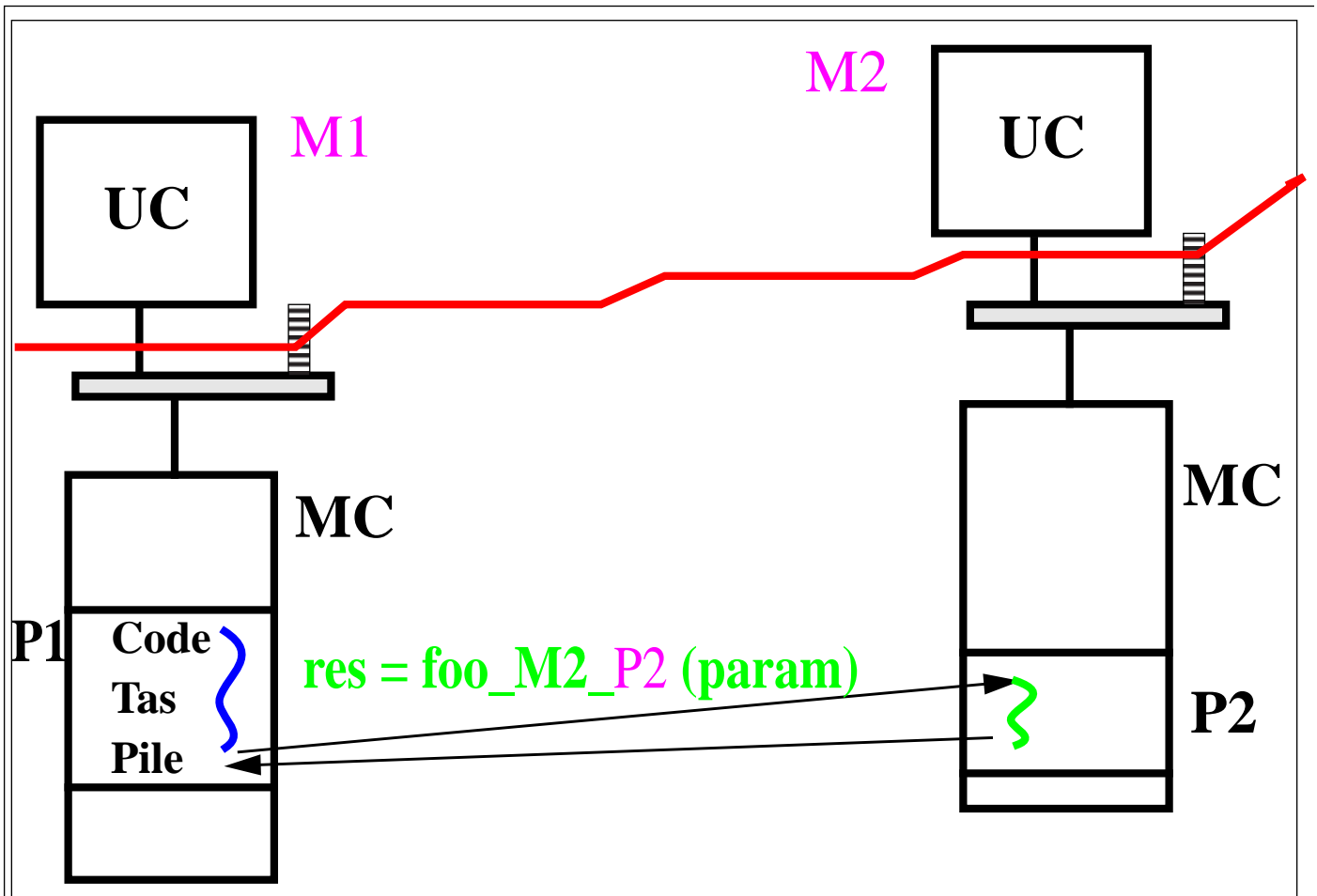


FIGURE 3 Principe d'un RPC

Modèle Client-Serveur

Plus structuré que simplement envoyer un message:

- On communique: paramètres
- On spécifie le code à exécuter: procédure
- On communique: résultats

Notes:

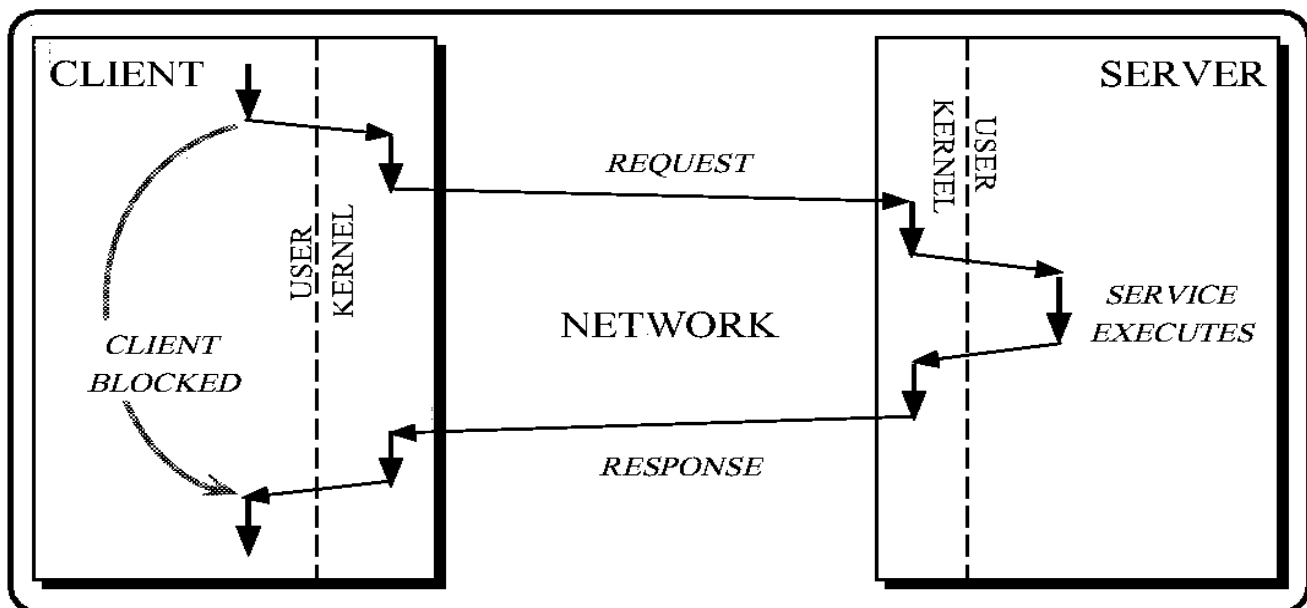
- NFS utilise pour son implémentation le RPC

- On utilise quelques fois le RPC même si il y a une mémoire unique:

communications sans partage afin de protéger les espaces d'adressage (OS)

1.2.4 Différentes vues d'un RPC

Vue temporelle:



- An RPC protocol contains two sides, the *sender* and the *receiver* (*i.e.*, *client* and *server*)
 - However, a server might also be a client of another server and so on...

FIGURE 4 Vue temporelle d'un RPC

Vue en couche

A Layered View of RPC

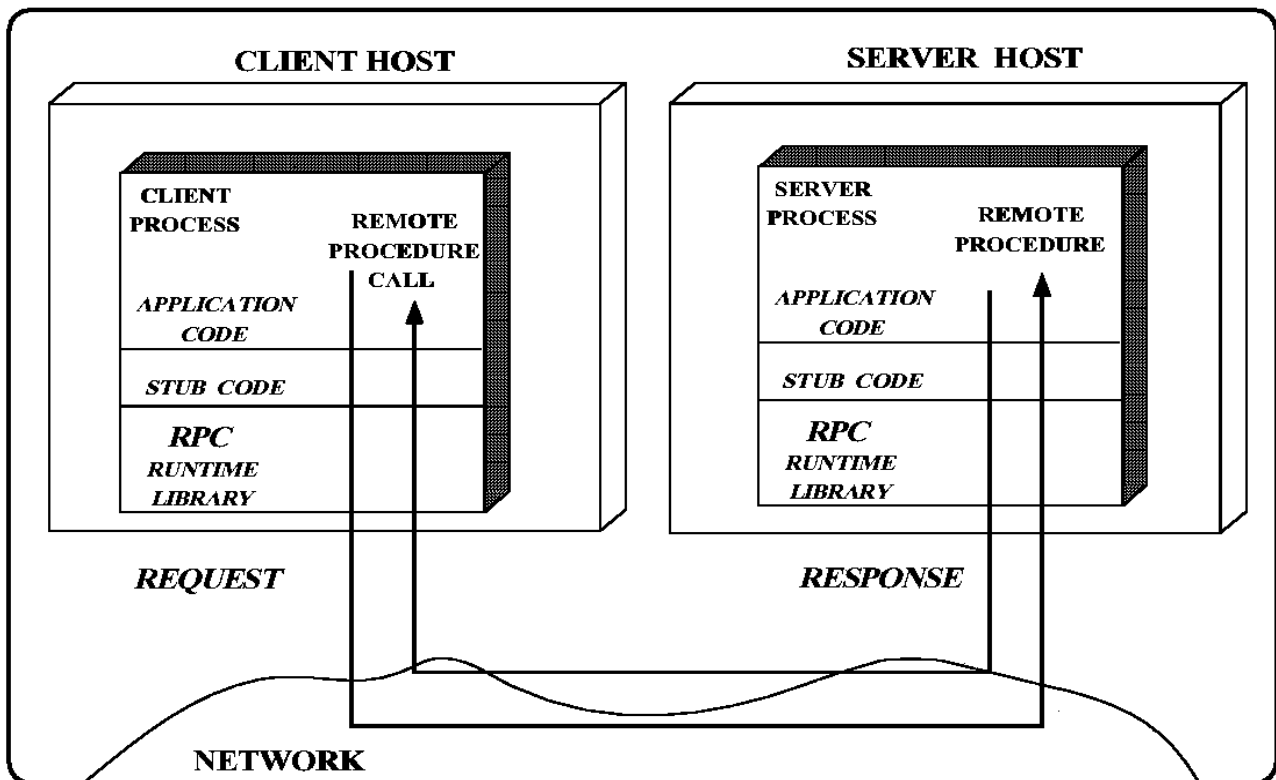


FIGURE 5 Vue en couche d'un RPC

Stub client = Talon client ou proxy

Stub server = Talon serveur ou Skeleton

1.2.5 Opérations successives d'un RPC

Appelant- Client:

- Appel de procédure RPC

Stub Client:

- Assemblage des arguments dans un message:
 - marshalling
 - gestion de l'hétérogénéité
- Génération d'un identificateur pour le RPC
- Un délai de garde est armé (G1)
- Détermination de l'adresse du serveur
- Transmission au protocole de transport pour émission sur le réseau vers le serveur

----- RÉSEAU -----

----> Dessin Synthétique au tableau

Stub Serveur:

- Le protocole de transport délivre le message au stub du serveur
- Le stub désassemble le message:
 - Unmarshalling
 - gestion de l'hétérogénéité
- Enregistrement de l'identificateur du RPC
- Appel de la procédure

Appelé - Serveur:

- Exécution de la procédure
- Retour au stub serveur avec param. résultats

Stub Serveur:

- Assemblage des résultats dans un message:
 - marshalling
 - gestion de l'hétérogénéité
- Un autre délai de garde est armé (G2)
- Transmission au protocole de transport pour émission sur le réseau vers le client

----- RÉSEAU -----

Stub Client:

- Le protocole de transport délivre le message au stub du client
- Dé-assemblage du message:
 - Unmarshalling
 - gestion de l'hétérogénéité
- Le délai de garde G1 est désarmé
- Un message d'acquiescement avec l'ID du RPC est envoyé au stub serveur (le délai de garde G2 est désarmé chez le serveur)
- Les résultats sont transmis à l'appelant

Appelant- Client:

- Retour de la procédure RPC et récupération du résultat

----- FIN du RPC -----

1.2.6 Traitement des RPC par le serveur

Un serveur séquentiel

```

loop {
    wait for RPC request;
    receive RPC request;
    decode arguments;
    execute desired function;
    send result to client
}
    
```

FIGURE 6 Un serveur séquentiel

Un seul RPC sera exécuté à la fois.

Problème si la fonction à exécuter est relativement longue,

si elle se bloque au milieu,

si elle fait un autre RPC .

---> que faire ?

Pour résoudre ce problème, on réalise des serveurs capables de traiter plusieurs RPC en parallèle (si la sémantique de l'application, et des procédures à exécuter le permettent).

----> Quelques fois, on ne veut pas :
on souhaite garder un traitement *séquentiel*
des requêtes, une seule à la fois, en *FIFO*

Un serveur parallèle

```
loop {  
    wait for RPC request;  
    receive RPC request;  
    decode arguments;  
    spawn a process or a thread{  
        execute desired function;  
        reply result to client  
    }  
}
```

FIGURE 7 Un serveur parallèle

On préfère largement le multi-threading pour un serveur parallèle car cela est plus efficace, et utilise moins de ressources qu'une implémentation à base de processus.

A voir en TD: la stratégie RMI pour gérer le multi-threading au niveau des Objets Remotes.

1.2.7 Interactions Client-Serveur

Démarrage d'un serveur

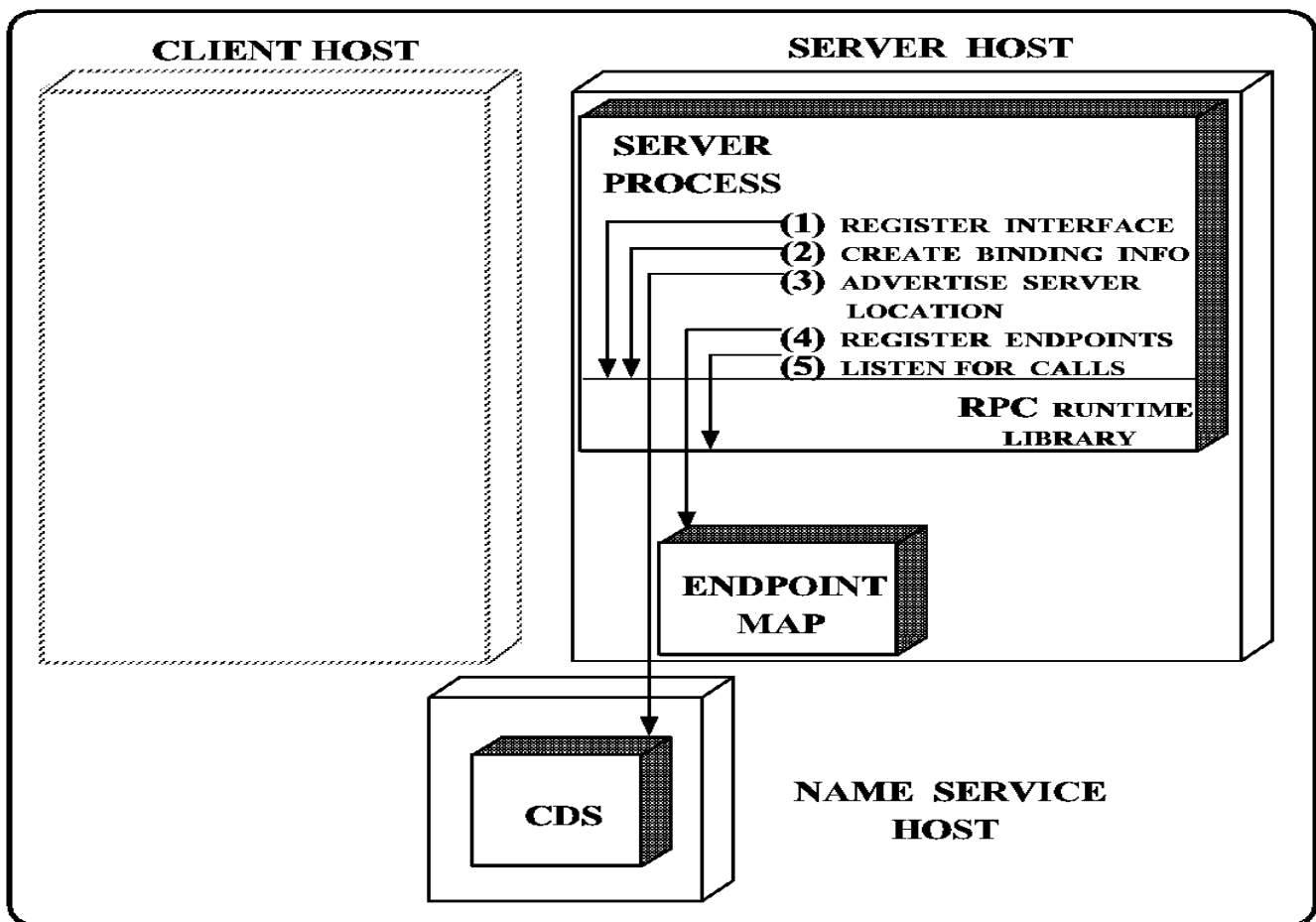


FIGURE 8 Démarrage d'un serveur

EndPoint Map = e.g. les ports d'une machine,
Port Mapper

CDS= Cell Directory Service, LDAP, JNDI,

...

Directory service = Un service de répertoire qui
permet à un client de trouver un serveur (binding),
rpcbnd, portmap, RMIregistry

à partir d'un NOM

Notes:

- CDS Name Service Host, Trouver la machine
- Endpoint map: at the level of the OS

Démarrage d'un client

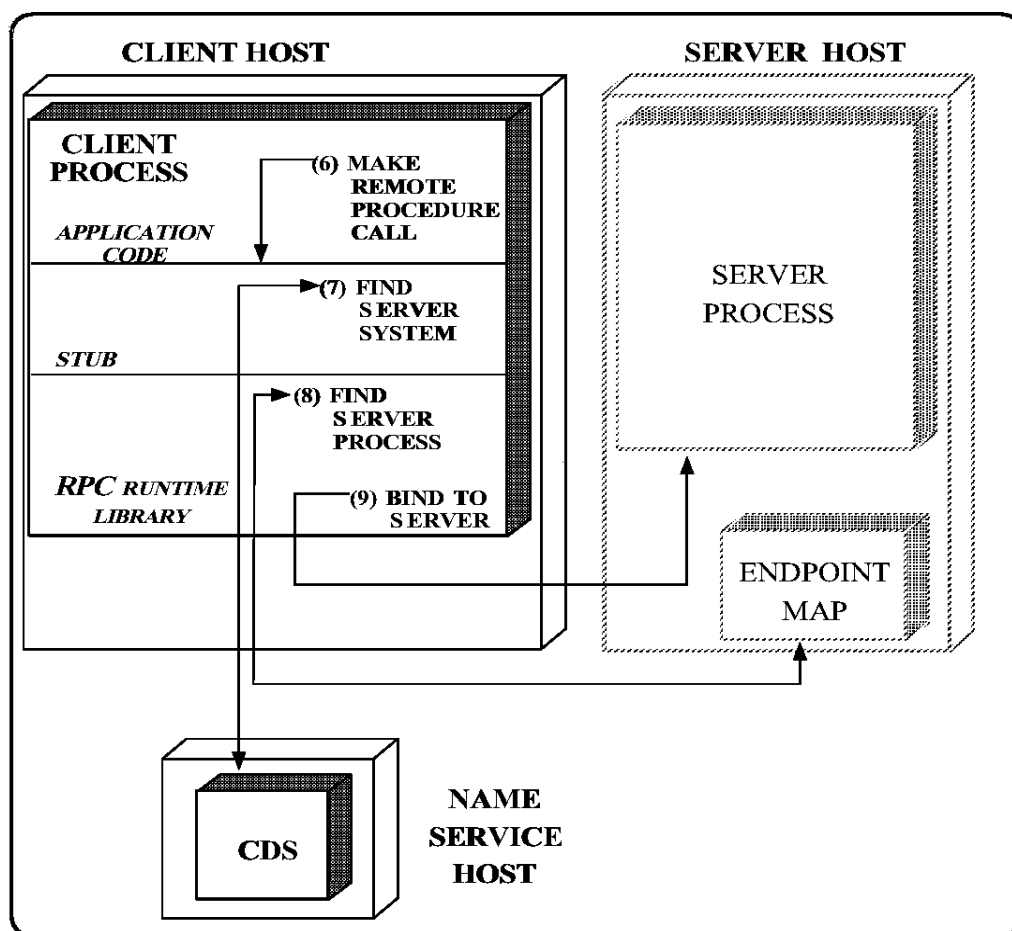


FIGURE 9 Démarrage d'un client

EndPoint Map= No Port TCP ou UDP, etc.

Interactions Client / Serveur

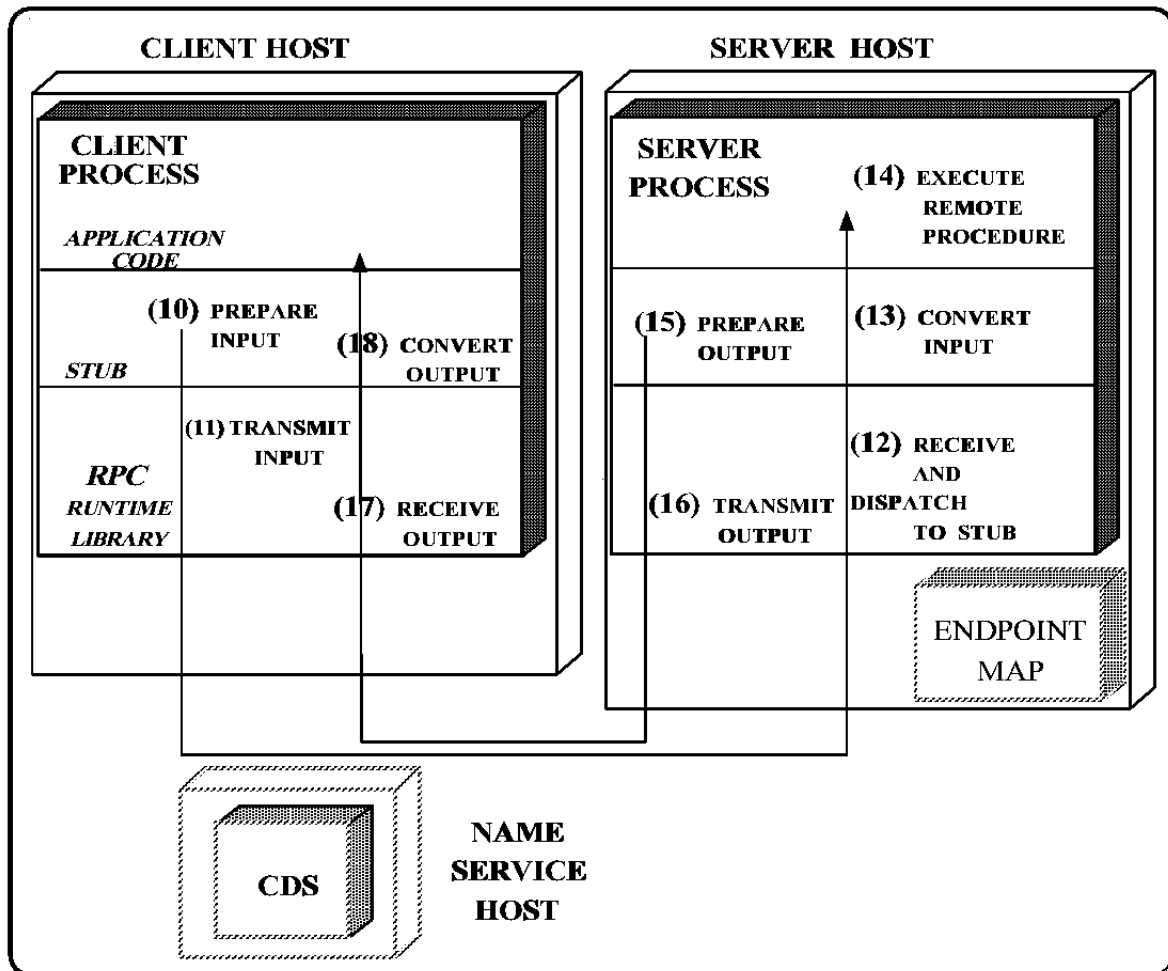


FIGURE 10 Interactions Client / Serveur

(12) = trouver la routine à exécuter

1.3 Composantes et variations d'un RPC

1.3.1 Forme, syntaxe

Forme et syntaxe du RPC varient dans le code:

```
res = foo_M2 (param);  
foo_M2_P2 (param, res);  
res = P2.foo (param);  
foo_ (Param, M2P2)  
foo_ (Param, M2, P2)
```

1.3.2 Passage des paramètres

Sémantique par copie, par référence, etc.

1.3.3 Représentation des données

Intéropérabilité entre des machines incompatibles

Utilisation de XDR: External Data Representation

1.3.4 Sémantique de l'appel

Synchrone

Asynchrone

Futur

1.3.5 Gestion des erreurs

Panne ou congestion du réseau

Panne du client, du serveur

Avant ou pendant le traitement du RPC

Sémantique “at most one“:

Si **OK**, alors exécution **1 fois**

Si **erreur**, alors **0 ou 1 fois** (pas 2 ou plus)

Remontées des erreurs vers le client par des exceptions

1.3.6 Code sur machine distante

Comment est assurée la disponibilité du code sur la machine distante ?

--> • **Code Mobile**

--> • **Java**

--> • **Serveurs d'applications**

1.3.7 Découverte et nommage

Découverte et nommage :

des machines

des processus

des services (interfaces)

des objets accessibles

1.3.8 Terminaison et GC distribué

Comment savoir si un serveur peut être arrêté ?

Comment savoir si un objet accessible à distance n'est plus référencé par personne ?

1.3.9 Persistance des “objets“, Tolérance aux fautes

Le temps de la vie en mémoire:

résistance aux pannes

ou

Autre solution:

se trouvent sur un support de type disque, avec
“**activation**“ (mise en mémoire) à la demande

Economise également des ressources.

1.3.10 Sécurité

Peut-on contrôler:

qu'un client donné a le droit d'appeler un serveur donné

qu'il a le droit d'accéder à un service donné

que personne n'intercepte les communications

On peut avoir:

du contrôle d'accès,

des protocoles d'authentification,

des protocoles de cryptage des communications : A, I, C,

--> • **A : Authentification**

--> • **I : Intégrité**

--> • **C : Confidentialité**

Une propriété souvent recherchée (pas simple à obtenir): NR

--> • **Non Répudiation**

Note: ces protocoles ne sont pas encore intégrés correctement avec des modèles généraux de RPC.

Recherches en cours (en particulier avec traitement de la mobilité du code et des calculs).

1.3.11 Autres aspects et concepts

Transactions

Migration du client sur la machine du serveur

Communications de groupe, et typés

Middleware à Messages (MOM, e.g. Java JMS):

Messages, queues de messages,

Messages vus comme des ressources transactionnelles

Asynchrone, mais Non typé

Communication événementielle:

événements, et réactions (traitement associé à l'occurrence d'un événement)

Anonymat: indépendance entre l'émetteur et les consommateurs d'un événement

...

1.4 Conclusion et Evolutions en cours

La programmation distribuée est en train de devenir de plus en plus une composante incontournable des logiciels.

Commerce Electronique, Portails de Calculs, Cartes à Puces et Terminaux, Applications collaboratives, Jeux distribués, etc.

Grille, Web Services (SOAP, WSDL), P2P

C'est également en passe de devenir bien plus sophistiqué qu'une simple interaction Client-Serveur.

De moins en moins de Clients, et de Serveurs purs, mais des processus, voire des objets distribués, qui communiquent, inter-agissent en s'échangeant des références, font des callbacks, etc.

Certains aspects vont prendre de l'importance:

- *Intégration plus fine au modèle N1/3*
- *Interaction entre la programmation distribuée et les Composants*
- *Sécurité*

Rares sont les programmeurs
qui ne font pas de la Programmation Répartie !!!!

CHAPITRE 2 RPC en C

Utilisation du RPC de sun: rpcgen

2.1 Principes

2.1.1 Généralités

Génère du code C pour réaliser du RPC en C

Prend en entrée un langage (similaire au C, .h) qui permet de spécifier l'interface du serveur.

Génère les fichiers stubs pour clients et serveurs

Gère l'interopérabilité par XDR.

Echange de données indépendantes de l'architecture interne des machines

Avec RPC de Sun:

On peut avoir plusieurs paramètres

Un seul résultat

Adressage d'une procédure distante:

nom du programme,
numéro de version (du programme),
nom de la procédure.

Enregistrement du serveur auprès d'un service dédié :

portmapper

2.1.2 Serveur de noms RPC : portmapper

Enregistrement d'un serveur RPC auprès d'un service dédié: **le portmapper**

Il joue le rôle d'un **serveur de nom** pour RPC.

Il convertit :

<Numéro Program. (Nom) + Version + Protocole>

--->

< numéro de port >

Caractéristiques:

Nom du service : **portmapper**, ou **rpcbind**

Processus:

linux : portmap ou encore rpc.portmap

Solaris : rpcbind

Windows : portmap.exe

Numéro de port: 111

Fonctionnement:

Lorsqu'on lance un nouveau serveur RPC:

- - il crée une socket (appel noyau) qui lui assigne dynamiquement un port libre.
- puis il s'enregistre auprès du portmapper en lui donnant : No Prog., No Version, Port + Protocol

Lorsqu'un client veut se connecter sur un server RPC d'une machine donnée :

- - il contacte le portmapper et lui demande le numéro de port correspondant à :
< No Prog., No Version, Nom protocole >
- puis il contacte le serveur directement par le protocole et le numéro de port.

Normalement c'est le seul service RPC à avoir besoin d'un numéro de port attribué **STATIQUEMENT**, les autres services RPC auront un numéro attribué **dynamiquement** (grâce au système et au portmapper qui mémorise cette association justement).

Services RPC connues ("well known RPC") :

Certains numéro de services (programmes) RPC ont un nom et des alias afin de faciliter leur utilisation. Exemple:

/etc/rpc - miscellaneous RPC-based services

```
#
rpcbind      100000 portmap sunrpc rpcbind
rstatd       100001 rstat rup perfmeter
rusersd      100002 rusers
nfs          100003 nfsprog
ypserv       100004 ypprog
mountd       100005 mount showmount
ypbind       100007
walld        100008 rwall shutdown
```

yppasswdd 100009 yppasswd

Ce mapping est décrit dans : [/etc/rpc](#)

D'autre part, il existe certains services RPC qui ont des ports récurrents:

```
program vers proto port
100003 2 udp 2049 nfs
100003 3 udp 2049 nfs
```

Pour connaître les services actuellement enregistrés dans le portmapper:

rpcinfo -p (Solaris)

/usr/sbin/rpcinfo (Linux)

Exemple:

```
clio > rpcinfo -p
program vers proto port service
100000 4 tcp 111 rpcbind
100000 3 tcp 111 rpcbind
100004 2 udp 752 ypserv
100004 1 udp 752 ypserv
100003 2 udp 2049 nfs
100003 3 udp 2049 nfs
100026 1 udp 32956 bootparam
100026 1 tcp 32804 bootparam
300598 1 udp 32958
805306368 1 udp 32958
805306368 1 tcp 32805
```

...

Les deux derniers : Programmes Utilisateurs

2.2 Mise en oeuvre

2.2.1 Fichiers de base

A partir d'un fichier, 4 fichiers sont générés:

Interface.x ---- rpcgen ---->

--> • **Interface.h**

Header file

--> • **Interface_clnt.c**

Stubs client

--> • **Interface_svc.c**

Stubs serveur

--> • **Interface_xdr.c**

Routines de conversions XDR pour les structures définies par l'utilisateur

SYNOPSIS

```
rpcgen Interface.x
```

Principes et fichiers de rpcgen

IDL

exemple de mise en oeuvre

RPCGEN : langage C / Unix

Interface.x

rpcgen

Talon_client.c

Interface.h

Talon_serveur.c

Code_client.c

Code_serveur.c

CC

RPC_lib.a

CC

Programme_client

Programme_serveur

FIGURE 11 Principes et fichiers rpcgen

IDL: Interface Definition Language

2.2.2 Exemple d'utilisation

On peut également générer:

--> • **Interface_client.c**

Un exemple de client (sample client)

--> • **Interface_server.c**

Un exemple de serveur (sample server)

--> • **makefile.Interface**

Exemple de makefile

SYNOPSIS

rpcgen -C

ANSI C compilers, and C++

rpcgen -N

Permet d'avoir des procédures à arguments multiples. Attention, ce n'est pas le défaut (pb. de compatibilité)

rpcgen -Sc Interface.x

Génère l'exemple de client

rpcgen -Ss Interface.x

Génère l'exemple de serveur

rpcgen -Sm Interface.x

Génère l'exemple de makefile

rpcgen -C -N -a Interface.x

Génère tout (4 fichiers + exemples client et serveur + makefile)

2.2.3 Démarrage et configuration

Le serveur peut être démarré par un démon (e.g. *inetd*) ou directement.

Option **-I** pour faire une génération qui permet un démarrage avec *inetd*

On peut choisir les transports pour lesquels on démarre le service (variable *NETPATH*).

Voir aussi l'option **-n netid** pour contrôler la génération de stub côté serveur.

rpcgen -n tcp ...

Option **-M** pour générer des stubs “thread-safe”

Option **-K seconds** pour contrôler le temps d'inactivité après lequel le serveur se termine (si démarrage avec *inetd*). Permet aussi de mettre 0 si démarrage d'un nouveau processus à chaque requête.

rpcgen

usage: rpcgen infile

rpcgen [-abkCLNTM][-Dname[=value]] [-i size] [-l [-K seconds]] [-Y path] infile

rpcgen [-c | -h | -l | -m | -t | -Sc | -Ss | -Sm] [-o outfile] [infile]

rpcgen [-s nettype]* [-o outfile] [infile]

rpcgen [-n netid]* [-o outfile] [infile]

options:

- a generate all files, including samples
- b backward compatibility mode (generates code for SunOS 4.1)
- c generate XDR routines
- C ANSI C mode
- Dname[=value] define a symbol (same as #define)
- h generate header file
- i size size at which to start generating inline code
- l generate code for inetd support in server (for SunOS 4.1)
- K seconds server exits after K seconds of inactivity
- l generate client side stubs
- L server errors will be printed to syslog
- m generate server side stubs
- M generate MT-safe code
- n netid generate server code that supports named netid
- N supports multiple arguments and call-by-value
- o outfile name of the output file

- s nettype** generate server code that supports named nettype
- Sc** generate sample client code that uses remote procedures
- Ss** generate sample server code that defines remote procedures
- Sm** generate makefile template
- t** generate RPC dispatch table
- T** generate code to support RPC dispatch tables
- Y path** directory name to find C preprocessor (cpp)

2.3 Exemple

- D'après Michel Syska
- D'après Comer, D. E. and D. L. Stevens [1996], "Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications," Prentice-Hall

Programme de dictionnaire accessible à distance

RDICT

2.3.1 Principes

Contenu de *rdict.tar* :

Makefile

rdict.x

Spécification de l'interface du serveur

rdict.c

Programme client

```
handle = clnt_create(argv[1], RDICTPROG, RDICTVERS,  
                                                             "tcp");  
...  
insertion(Mot);
```

rdict_cif.c

Fichier intermédiaire du côté client qui permet de se passer du nom de version et du pointeur distant vers le serveur (handle):

```
int insertion(char* Mot) {  
    return *insertion_1(Mot, handle);  
}
```

svc = sans doute à “Server Code”

rdict_server.c

Procédures appelées par le runtime RPC du côté serveur.

```
int* insertion_1_svc(char *argp, struct svc_req *rqstp){  
    static int resultat;  
    /*  
    * insert server code here  
    */  
    resultat = insertion(argp);  
    return(&resultat);  
}
```

rdict_srp.c

Implémentation effective des procédures du serveur.

Code purement serveur

```
int
insertion(Mot)
char*Mot;
{
    strcpy(Dictionnaire[CompteurMots], Mot);
    CompteurMots++;
    return CompteurMots;
}
```

Fichiers Générés:

rdict.h Fichier .h avec toutes les déclarations

rdict_xdr.c Conversions de format

rdict_clnt.c Stub/Proxy Client

rdict_svc.c Skeleton (stubs) Serveur

2.3.2 Interface .x

```

/* rdict.x */

/* Declarations en RPCL du programme dictionnaire pour l'outil rpcgen */

const TAILLEMOT = 50;      /* Taille maximale d'un mot du dictionnaire */
const TAILLEDICTIONNAIRE = 100; /* nombre d'entrées du dictionnaire */
/* Ajout pour montrer XDR */
struct example {          /* unused structure declared here to */
    int  exfield1; /* illustrate how rpcgen builds XDR */
    char exfield2; /* routines to convert structures. */
};
/*-----
 * RDICTPROG - programme distant qui propose les services de gestion
 * du dictionnaire : initialise, insertion, suppression, recherche
 * Les numéros sont ceux utilisés par portmap
 * voir rpcinfo -p localhost (les numéros sont en décimal)
 *-----
 */
program RDICTPROG { /* nom du programme */
    version RDICTVERS { /* numéro de version */
        int INITIALISE(void) = 1; /* première procédure du programme */
        int INSERTION(string) = 2; /* seconde procédure */
        int SUPPRESSION(string) = 3; /* troisième procédure */
        int CHERCHE(string) = 4; /* quatrième procédure */
    } = 1; /* numéro de la version du programme */
} = 0x30090949; /* numéro de programme */

```

2.3.3 Exemple de client

rdict.c

```

#include <rpc/rpc.h>

#include <stdio.h>
#include <ctype.h>

#include "rdict.h" /* Généré par RPCGEN */

#define TAILLEMOT 50      /* Taille maximale d'un mot du dictionnaire */

```

Exemple

```
#define RSERVEUR "localhost" /* nom du "remote host" serveur */
                          /* à passer en paramètre argv[1] */
CLIENT *handle;          /* handle de la procédure serveur */

/*-----
 * main - insere, supprime, ou cherche un mot dans le dictionnaire
 *-----
*/
int
main(argc, argv)
int  argc;
char *argv[];
{
    char  Mot[TAILLEMOT+1]; /* allocation de la mémoire pour un mot */
    char  Commande;        /* commande saisie */
    int   LongueurMot;     /* longueur réelle du mot saisi */

    /* connexion avec le serveur pour les appels RPC */
    handle = clnt_create(argv[1], RDICTPROG, RDICTVERS, "tcp");
    if (handle == 0) {
        printf("La connexion avec le programme distant ne peut etre
établie...\n");
        exit(1);
    }
    /* appels RPC vers le serveur */
    while (1) {
        initialise_1(0, handle); /* Convention: nom de la procedure_version */
        insertion_1(Mot, handle);
        suppression_1(Mot, handle);
        recherche_1(Mot, handle);
    }
}
```

On peut l'écrire comme ci-dessus, mais en pratique:

```
insertion(Mot);
```

2.3.4 Fichier intermédiaire client: cif.c

```

/* rdict_cif.c - initialise, insertion, suppression, recherche */
#include <rpc/rpc.h>
#include <stdio.h>
#include "rdict.h"
/* Talon (ou souche | "stub") coté client à écrire "à la main" */
externCLIENT*handle; /* handle de la procédure serveur */
/*-----
 * initialise - procedure interface client qui appelle initialise_1
 *-----
 */
int initialise()
{
    return *initialise_1(handle);
}
/*-----
 * insertion - procedure interface client qui appelle insertion_1
 *-----
 */
int insertion(char* Mot)
{
    return *insertion_1(Mot, handle);
}

/*-----
 * suppression - procedure interface client qui appelle suppression_1
 *-----
 */
int suppression(char* Mot)
{
    return *suppression_1(Mot, handle);
}
/*-----
 * recherche - procedure interface client qui appelle recherche_1
 *-----
 */
int recherche(char* Mot)
{
    return *recherche_1(Mot, handle);
}

```

2.3.5 Exemple de serveur

rdict_server.c

Procédures appelées par le runtime RPC du côté serveur.

```
#include "rdict.h"
```

```
int* initialise_1_svc(struct svc_req *rqstp)
```

```
{  
    static int resultat;  
  
    /*  
     * insert server code here  
     */  
    resultat = initialise();  
  
    return (&resultat);  
}
```

```
int* insertion_1_svc(char *argp, struct svc_req *rqstp)
```

```
{  
    static int resultat;  
  
    /*  
     * insert server code here  
     */  
    resultat = insertion(argp);  
    return(&resultat);  
}
```

```
int * suppression_1_svc(char * argp, struct svc_req *rqstp)
```

```
{  
    static int resultat;  
  
    /*  
     * insert server code here  
     */  
    resultat = suppression(argp);  
  
    return(&resultat);  
}
```

Exemple

```
}  
  
int * cherche_1_svc(char *argp, struct svc_req *rqstp)  
{  
    static int resultat;  
  
    /*  
     * insert server code here  
     */  
    resultat = cherche(argp);  
    return(&resultat);  
}
```

L'implémentation réelle des fonctions est dans un autre fichier :

rdict_srp.c

Implémentation effective des procédures du serveur.

```
/* rdict_srp.c - initialise, insertion, suppression, recherche */  
#include <rpc/rpc.h>  
#include <stdio.h>  
#include "rdict.h"  
/* procédures distantes coté serveur avec leurs données */  
char Dictionnaire[TAILLEDICTIONNAIRE][TAILLEMOT+1]; /* dictionnaire de mots */  
int CompteurMots = 0; /* nombre de mots dans le dictionnaire */  
  
/*-----  
 * initialise - initialise le dictionnaire avec 0 mots  
 *-----  
 */  
int  
initialise()  
{  
    CompteurMots = 0;  
    printf("Initialisation du dictionnaire ...\n");  
    return 1;  
}
```


Exemple

```
/*-----  
* insertion - insère un mot dans le dictionnaire  
*-----  
*/  
int  
insertion(Mot)  
char* Mot;  
{  
    strcpy(Dictionnaire[CompteurMots], Mot);  
    CompteurMots++;  
    return CompteurMots;  
}  
  
/*-----  
* suppression - supprime un mot du dictionnaire  
*-----  
*/  
int  
suppression(Mot)  
char* Mot;  
{  
    int    i;  
  
    for (i=0 ; i<CompteurMots ; i++)  
        if (strcmp(Mot, Dictionnaire[i]) == 0) {  
            CompteurMots--;  
            strcpy(Dictionnaire[i], Dictionnaire[CompteurMots]);  
            return 1;  
        }  
    return 0;  
}  
  
/*-----  
* recherche - cherche un mot dans le dictionnaire  
*-----  
*/  
int  
cherche(Mot)  
char* Mot;  
{  
    int    i;  
  
    for (i=0 ; i<CompteurMots; i++)
```

```
        if (strcmp(Mot, Dictionnaire[i]) == 0)
            return 1;
    return 0;
}
```

2.3.6 Conversions XDR

rdict_xdr.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "rdict.h"

bool_t
xdr_example(register XDR *xdrs, example *objp)
    /* example == NOM DE LA STRUCTURE */
/* procedure generee */{
    register long *buf;

    if (!xdr_int(xdrs, &objp->exfield1))
        return (FALSE);
    if (!xdr_char(xdrs, &objp->exfield2))
        return (FALSE);
    return (TRUE);
}
```

2.3.7 Header file: rdict.h

```

/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _RDICT_H_RPCGEN
#define _RDICT_H_RPCGEN

#include <rpc/rpc.h>
#define TAILLEMOT 50
#define TAILLEDICTIONNAIRE 100

struct example {
    int exfield1;
    char exfield2;
};
typedef struct example example;

#define RDICTPROG ((unsigned long)(0x30090949)) /* Numero du programme */
#define RDICTVERS ((unsigned long)(1))
#define INITIALISE ((unsigned long)(1))
extern int * initialise_1(); /* kind of stub client */
#define INSERTION ((unsigned long)(2))
extern int * insertion_1();
#define SUPPRESSION ((unsigned long)(3))
extern int * suppression_1();
#define CHERCHE ((unsigned long)(4))
extern int * cherche_1();
extern int rdictprog_1_freeresult();

/* the xdr functions */
extern bool_t xdr_example();

#endif /* !_RDICT_H_RPCGEN */

```

2.3.8 Stubs client

rdict_clnt.c

```

#include "rdict.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int * initialise_1(argp, clnt) /* clnt == Handle */
    void *argp;
    CLIENT *clnt;
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    /* REIFICATION of the call */
    if (clnt_call(clnt, INITIALISE, /* target (Handle), Method Name */
        (xdrproc_t) xdr_void, (caddr_t) argp, /* Params */
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res, /* Where to put result */
        TIMEOUT) != RPC_SUCCESS) { /* == G1: Guard Delai */
        return (NULL);
    }
    return (&clnt_res);
}

int * insertion_1(argp, clnt)
    char **argp;
    CLIENT *clnt;
{
    static int clnt_res;

    memset((char *)&clnt_res, 0, sizeof (clnt_res));
    if (clnt_call(clnt, INSERTION,
        (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
        (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
etc.
suppression_1 ... recherche_1 ...

```

2.3.9 Stubs serveur

rdict_svc.c (skeleton)

```

/* Please do not edit this file. It was generated using rpcgen. */
#include "rdict.h"
...
static void
rdictprog_1(struct svc_req *rqstp, register SVCXPRT *transp) {
    union {
        char *insertion_1_arg;
        char *suppression_1_arg;
        char *cherche_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);
    _rpcsvccount++;
    switch (rqstp->rq_proc) {          /* Procedure to call */
    case NULLPROC:
        (void) svc_sendreply(transp,
            (xdrproc_t) xdr_void, (char *)NULL);
        _rpcsvccount--;
        _rpcsvcstate = _SERVED;
        return;
    case INITIALISE:
        _xdr_argument = (xdrproc_t) xdr_void;    /* Decoding arguments */
        _xdr_result = (xdrproc_t) xdr_int;
        local = (char *(*)(char *, struct svc_req *)) initialise_1_svc;
                                                    /* Setting the procedure to call */
        break;
    case INSERTION:
        _xdr_argument = (xdrproc_t) xdr_wrapstring;
        _xdr_result = (xdrproc_t) xdr_int;
        ...
    result = (*local)((char *)&argument, rqstp); /* Execution of the (reified) call */
    ... }
    main() {
        pid_t pid;
        int i;
        ...
    if (!svc_create(rdictprog_1, RDICTPROG, RDICTVERS, "netpath")) {
    ... }

```