# Réalisation d'un serveur pour l'accès à ORACLE : conception et réalisation d'un serveur d'information bibliographique

Marie-Christine Pajon

CONSERVATOIRE NATIONAL
DES ARTS ET METIERS
CENTRE AGREE DE GRENOBLE (C.U.E.F.A.)

---

MEMOIRE
présenté en vue d'obtenir
LE DIPLOME D'INGENIEUR C.N.A.M.
en
INFORMATIQUE
par
Marie-Christine PAJON

---

# REALISATION D'UN SERVEUR POUR L'ACCES A ORACLE
# &
# CONCEPTION ET REALISATION D'UN SERVEUR D'INFORMATION BIBLIOGRAPHIQUE

Soutenu le 14 Décembre 1989

---

Les travaux relatifs au présent mémoire ont été effectués au Centre de Recherche BULL de Grenoble sous la direction de Monsieur Christian LENNE.

Je remercie Mr Claude Kaiser, Professeur au Conservatoire des Arts et Métiers, qui me fait l'honneur de présider le jury de ce mémoire.

Je remercie Mr Jacques Courtin, Professeur à l'Université des Sciences Sociales de Grenoble, qui m'a suivie et conseillée au cours de ma formation CNAM.

Je tiens à remercier Mr Mauricio Lopez, Ingénieur au Centre de Recherche Bull de Grenoble, qui m'a accueillie au sein de son équipe pendant la préparation de ce mémoire.

J'adresse mes sincères remerciements à Mr Christian Lenne, Ingénieur au Centre de Recherche Bull de Grenoble, qui m'a fait partager ses compétences et sa bonne humeur tout au long de ce stage. Je tiens à lui témoigner toute ma gratitude pour la disponibilité qu'il m'a accordée et l'intérêt qu'il a manifesté pour mon travail. Ses remarques m'ont permis d'améliorer largement le contenu de ce document.

Je remercie vivement Mr Samer Haj Houssain, Ingénieur au Centre de Recherche Bull de Grenoble, pour la confiance qu'il m'a accordée en me proposant ce stage. Je lui suis très reconnaissante de l'aide qu'il m'a apportée dans la réalisation de ce travail.

Je remercie également Mr Jean-Paul Cahen, Chef de Projet Syseca, qui a accepté de participer à ce jury.

Je tiens à exprimer ma reconnaissance à Mr Louis Bolliet, Professeur Emérite à l'Université des Sciences Sociales de Grenoble. J'ai été très sensible aux conseils et aux encouragements qu'il m'a apportés tout au long de ma formation.

Je tiens à remercier tous les membres du Centre de Recherche Bull de Grenoble et du Laboratoire de Génie Informatique, pour leur accueil chaleureux et leur sympathie.

# RESUME

Le développement d'applications réparties, qui utilisent pour la gestion des données le Système de Gestion de Bases de Données (SGBD) relationnel Oracle, a mis en évidence la nécessité d'une interface performante pour les accès à la base.

La solution est la définition d'une interface procédurale, construite sur le code HLI (High Level Interface) d'Oracle.

L'interface ainsi définie, baptisée OPIDUM, "Oracle Procedural Interface for Distributed Unix Machines", réunit les avantages du pré-compilateur Oracle (expression des requêtes SQL en clair) et de l'interface HLI (élimination de la phase de pré-compilation).

Grâce au serveur Opidum, une application résidante sur une machine peut accéder à un SGBD implanté sur toute autre machine, connectée sur le même réseau de communication : l'accès aux données est indépendant de la localisation de l'application.

Un serveur d'Information Bibliographique accessible à distance, a été construit sur un SGBD Oracle. Le serveur BIB offre des fonctionnalités qui constituent un support au développement d'applications de gestion bibliographiques réparties : administration de la base, mise à jour et consultation de la bibliographie. Il met à la disposition des usagers un langage qui permet de formuler des requêtes de sélection, combinant plusieurs rubriques. Toutes les rubriques sont interrogeables, y compris les rubriques textuelles telles que le résumé ou la table des matières, qui décrivent le contenu des documents.

Les deux serveurs utilisent un mécanisme d'appel de procédure à distance, fonctionnant sur un réseau local Ethernet.

**MOTS-CLES**

UNIX, bases de données relationnelles, Oracle, interface procédurale, communication inter-processus, appel de procédure à distance, système bibliographique, recherche textuelle

# TABLE DES MATIERES


**Introduction** 1

**Chapitre 1**

**La communication inter-processus** 4

**1. Introduction** 5

**2. Les moyens de communication entre processus** 5

2.1. La communication locale entre processus 5

2.2. La communication entre processus distants 7

2.2.1. Les "sockets" 7

2.2.2. L'appel de procédure à distance 10

2.3. Choix du moyen de communication dans les applications distribuées 10

**3. Etude du mécanisme de l'appel de procédure à distance** 11

3.1. L'appel de procédure locale 11

3.2. Concept général de l'appel de procédure à distance 11

3.3. Problèmes dans la réalisation d'un service d'appel de procédure à distance 12

3.3.1. La sémantique de l'appel de procédure à distance 13

3.3.2. Le traitement des exceptions 13

3.3.3. La présentation des paramètres 14

**4. Etude de cas : le logiciel RPC de Sun** 14

4.1. Mise en oeuvre de la transparence de la distribution 14

4.2. La représentation des données par RPC/Sun 16

4.3. Conditions pratiques d'utilisation 17

4.3.1. Contraintes de programmation 17

4.3.2. Le langage RPC/Sun 20

4.3.3. Construction d'une application en langage C 22

**5. Conclusion** 23

## Chapitre 2


**OPIDUM : un serveur pour l'accès à Oracle** 25

**1. Introduction** 26

**2. Une interface procédurale pour l'accès à Oracle** 26
- 2.1. L'exécution d'un programme d'application par le SGBD Oracle 26
- 2.2. Choix d'une interface procédurale 28
- 2.3. Principe d'utilisation d'OPIDUM 31

**3. Choix et implémentation** 33
- 3.1. Choix du modèle client/serveur 33
- 3.2. Mise en oeuvre du modèle : un serveur par client 36
  - 3.2.1. Solution générale 36
  - 3.2.2. Problèmes rencontrés 39
- 3.3. Architecture du serveur Opidum 42
  - 3.3.1. En local 42
  - 3.3.2. En réparti 45

**4. Conclusion** 51


## Chapitre 3


**BIB : un serveur d'Information Bibliographique** 53

**1. Introduction** 54

**2. Définition d'un système de gestion bibliographique** 54
- 2.1. La fonction de gestion bibliographique 54
- 2.2. Analyse d'outils existants 56
  - 2.2.1. Des outils sous Unix 56
  - 2.2.2. Le produit SuperDoc 58
- 2.3. Spécifications générales du système proposé 59
  - 2.3.1. Les données 59
  - 2.3.2. Les traitements 62

**3. Spécifications techniques du système** **64**
3.1. Architecture logicielle 64
3.1.1. Principe général 64
3.1.2. Choix du schéma de fonctionnement 65
3.2. Le stockage des données 69
3.2.1. Les types internes à l'application 69
3.2.2. Les méthodes des signatures 70
3.2.3. La représentation relationnelle 74
3.3. Les fonctions de consultation 79
3.3.1. Le langage d'interrogation 79
3.3.2. Implantation de la recherche textuelle 83
3.3.3. Implantation de la formule de consultation par rubrique 86

**4. Conclusion** **90**

**Conclusion** **91**

**Annexes**

Annexe 1 - Les protocoles TCP/IP 93
Annexe 2 - Exemple d'utilisation du logiciel RPC/Sun 96
Annexe 3 - Manuel d'utilisation du logiciel OPIDUM 102

**Bibliographie**

# Introduction

Le stage en vue de la préparation de ce mémoire a été effectué au sein de l'équipe Bases de Données, au Centre de Recherche Bull de Grenoble. Le travail a consisté à concevoir et réaliser deux outils destinés aux concepteurs d'applications, disposant d'un environnement de travail réparti; l'un a un caractère général et l'autre est spécifique au domaine de l'informatique documentaire :

- OPIDUM : un serveur pour l'accès à Oracle,
- BIB : un serveur d'Information Bibliographique.

Le développement du serveur OPIDUM est une retombée du projet Esprit , intitulé "Design and Operationnal Evaluation of Office Information Servers" (DOEOIS), qui a été conduit de 1985 à 1988 par le groupe Bull et ses partenaires. Le résultat global de ce projet est la conception et la réalisation d'un Serveur d'Information Bureautique (SIB). La gestion des données, y compris des documents est faite en utilisant le Système de Gestion de Bases de Données (SGBD) relationnel Oracle; cela a conduit à prévoir l'existence d'une interface performante pour accéder à une base de données Oracle.

Le serveur OPIDUM, implanté sur le site du SGBD, exporte un ensemble de fonctions appelables à distance par des programmes d'applications, localisées sur d'autres sites connectés sur le même réseau de communication.

La mise en oeuvre d'un serveur d'information bibliographique a été motivée par la nécessité d'un système de gestion bibliographique, évolué et adapté aux besoins hétérogènes des utilisateurs. Le serveur BIB offre des fonctionnalités, qui constituent un support au développement d'applications de gestion bibliographique réparties : il permet principalement d'archiver des informations bibliographiques dans le SGBD Oracle sous-jacent et de les extraire en fonction de critères de recherche divers. Il est étayé d'un programme interactif de démonstration.

Les deux outils ont été développés en langage de programmation C, sur le matériel Bull DPX 2000 sous le système d'exploitation SPIX 31.1, qui est la version Bull du système Unix. Leur réalisation s'appuie sur les outils standard du système Unix, et sur des logiciels spécifiques développés au Centre de Recherche Bull; ils utilisent un mécanisme d'appel de procédure à distance.

Le serveur Opidum et le serveur BIB sont destinés à être utilisés à la fois par les ingénieurs du Centre de Recherche Bull, implanté à Gières et les membres du Laboratoire de Génie Informatique (LGI), situé sur le domaine universitaire de Saint Martin-d'Hères.

Les différents sites qui regroupent les utilisateurs potentiels, sont équipés de réseaux locaux Ethernet reliés entre eux par une liaison spécialisée louée aux PTT et gérée par le protocole X25. Ainsi communiquent entre eux des matériels hétérogènes tels que : BULL (SPS9, SPS7), DEC (Vax et Micro-Vax), SUN, etc. Les échanges intra-réseaux et inter-réseaux utilisent les protocoles de communication standard (TCP/IP) au niveau de la couche transport représentée dans la quatrième couche du modèle ISO. Au niveau de la couche application, les services offerts aux utilisateurs, les plus généralement employés, sont le transfert de fichiers (FTP : File Transfert Protocol), la connexion sur des ordinateurs distants (TELNET : TErminal NETwork protocol), et le courrier électronique (SMTP : Simple Mail Transfert Protocol). Des extensions des commandes standard du système Unix appelées les r-commandes (exemples : rcp : remote copy, rlogin : remote login, rsh : remote shell) sont également disponibles.

Ce mémoire est organisé en trois chapitres. Le premier chapitre présente une étude d'ordre général, sur les moyens de communication entre les différents éléments de traitement dans une application répartie. Il décrit plus particulièrement le logiciel d'appel de procédure à distance, RPC (Remote Procedure Call) de Sun, qui a été choisi pour l'implémentation des serveurs Opidum et BIB.

Le second chapitre est consacré à la description détaillée du logiciel Opidum. Après une étude des interfaces proposées par Oracle, il a été choisi de développer une interface procédurale pour l'accès aux données. Nous expliquons les choix et l'implémentation de la version distribuée du logiciel.

Le troisième chapitre présente la définition et les spécifications techniques du serveur BIB. L'étude des systèmes de gestion bibliographique existants nous a inspirés dans le choix des fonctionnalités de l'outil.

Grâce à cet outil, un groupe d'utilisateurs peut personnaliser sa base bibliographique, en définissant ses propres descriptions bibliographiques des documents. Le serveur BIB met à la disposition des usagers, un langage qui permet de formuler des requêtes d'interrogation de la base, combinant n'importe quel élément bibliographique, y compris les informations relatives au contenu des documents, telles que le résumé, la table des matières, etc.

# Chapitre 1

# La communication inter-processus

# 1. Introduction

L'évolution des technologies dans les domaines du traitement et du transfert des informations permet la mise en oeuvre d'applications distribuées, mettant en jeu des ordinateurs différents interconnectés par des réseaux de communication. Ainsi, ces applications tirent profit de la répartition des données et des ressources de traitement. Au niveau des utilisateurs, la transparence à la distribution est souhaitable : les services proposés doivent être identiques à ceux habituellement disponibles en milieu non réparti. Cette contrainte conduit à utiliser , dans le développement d'applications distribuées, un ensemble de mécanismes et fonctions de communication offrant une abstraction du système réparti telle que celui-ci soit vu comme local.

Le développement d'applications distribuées recouvre les différents aspects suivants : la distribution des ressources, la distribution des données, et la communication entre les éléments de traitement répartis. Dans ce chapitre, nous nous intéressons plus particulièrement à l'aspect de communication entre les processus qui composent l'application.

# 2. Les moyens de communication entre processus

Dans les applications implémentées en terme de processus, certains processus qui s'exécutent sur la même machine ou sur des machines différentes, doivent communiquer entre eux. Nous décrivons le principe, les avantages et inconvénients des différents moyens permettant la communication entre processus dans l'environnement Unix, avec pour objectif de déterminer le moyen le plus intéressant à utiliser dans le cadre d'applications distribuées.

Les différents systèmes Unix commercialisés, appartiennent tous à l'un des deux standards : Berkeley (BSD) et System V. Il existe des différences entre ces deux souches, concernant les commandes et le système, nous signalons les particularités propres à chaque origine.

## 2.1. La communication locale entre processus

Les systèmes Unix, BSD et System V, offrent un ensemble de mécanismes pour la communication entre processus locaux (les processus devant communiquer résident sur la même machine).

### 1) Les "tubes"

Des données peuvent être transmises entre deux processus, ayant des liens de parenté, au travers d'un *tube* (**pipe**). Un tube est unidirectionnel et possède deux extrémités, une est réservée à la lecture, et l'autre à l'écriture. Un tube est généralement créé par un ancêtre

commun aux deux processus devant communiquer. La communication entre les processus s'effectue alors selon le mode "producteur-consommateur". Les tubes sont disponibles sur tous les systèmes Unix qu'ils soient Berkeley ou System V.

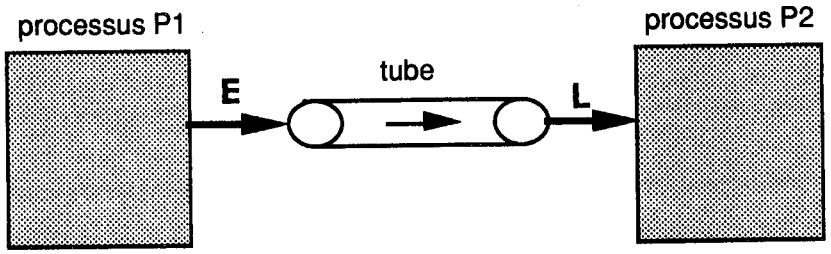



**Figure 1.1.** Communication inter-processus au travers d'un tube

### 2) La mémoire partagée

Les informations à communiquer entre les deux processus sont stockées dans une mémoire partagée, accessible en lecture/écriture par les deux processus. Les sémaphores servent alors à synchroniser les accès à la mémoire de sorte à préserver la cohérence des échanges. Ce mécanisme est disponible sur System V et sur certaines extensions de Berkeley.

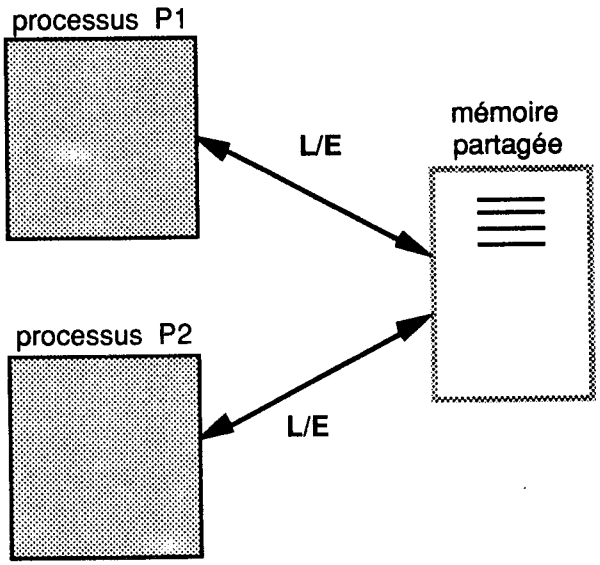



**Figure 1.2.** Communication inter-processus par mémoire partagée

### 3) Les messages

Deux processus peuvent communiquer à l'aide de messages, déposés dans une boîte aux lettres. Pour échanger des messages, les processus doivent bénéficier de la permission d'exploitation d'une **file d'attente de messages.** [SystemV 87]. Les messages peuvent être utilisés dans System V et dans certaines versions BSD.

L'inconvénient de ces méthodes est que les deux processus doivent s'exécuter sur une seule et même machine (par conséquent, il n'y a pas de répartition de l'application).

## 2.2. La communication entre processus distants

Les deux processus devant communiquer, s'exécutent sur des machines différentes interconnectées sur un réseau de communication.

### 2.2.1. Les "sockets"

Une couche logicielle, appelée **"sockets"**, construite sur les protocoles TCP/IP (ces protocoles sont décrits en annexe 1), permet d'établir la connexion entre deux processus, ceux-ci peuvent s'exécuter sur la même machine ou sur des machines différentes. La communication inter-processus est établie suivant le modèle client-serveur, le processus serveur et le processus client échangent des messages à travers un canal de communication bi-directionnel. Le processus serveur se met en attente de réception de messages sur un port (qui est un point d'accès au niveau TCP), à l'une des extrémités du canal, le processus client communique avec le serveur à travers un autre port, à l'autre extrémité du canal qui peut être localisée sur une autre machine.

Les primitives de création, nommage, envoi et réception de suite de caractères, appelées par chaque processus communiquant, servent à construire la **synchronisation du dialogue entre les processus.**

La fonction **socket** a pour effet d'ouvrir un port et d'affecter un numéro de port au processus qui exécute cette primitive [Unix 88]. La fonction **bind** donne un nom à un port et rend une valeur vrai ou faux en fonction de l'établissement de la connexion.

La communication entre processus peut se faire suivant deux modes, *avec connexion* ou *sans connexion.*

La communication en mode connexion fonctionne suivant le schéma un serveur par client : un "vrai" canal est établi entre le processus client et le processus serveur.

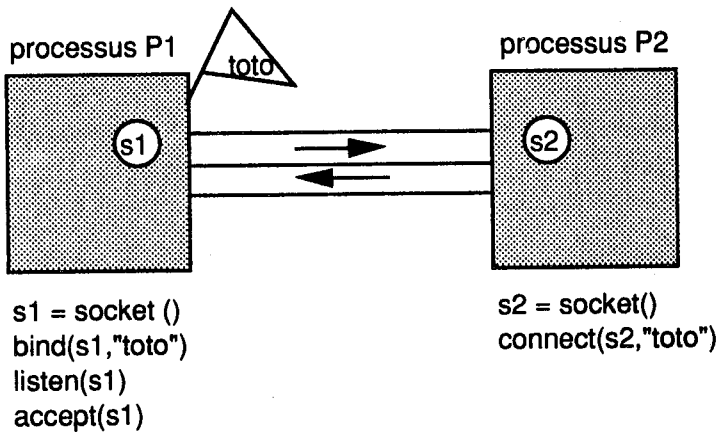



**Figure 1.3.** Communication inter-processus avec connexion

Dans l'exemple de la figure 1.3, le processus P1 joue le rôle du serveur, et le processus P2 le rôle du client. L'exécution des primitives **listen** et **accept** dans le processus P1 a pour effet de mettre celui-ci en attente de demandes d'ouverture de connexion, effectuées par des processus distants au moyen de la primitive **connect.**

Dans le mode sans connexion, le serveur satisfait les requêtes de manière individuelle, sans établissement de dialogue (une requête, une réponse) : ce sont des messages isolés qui circulent.

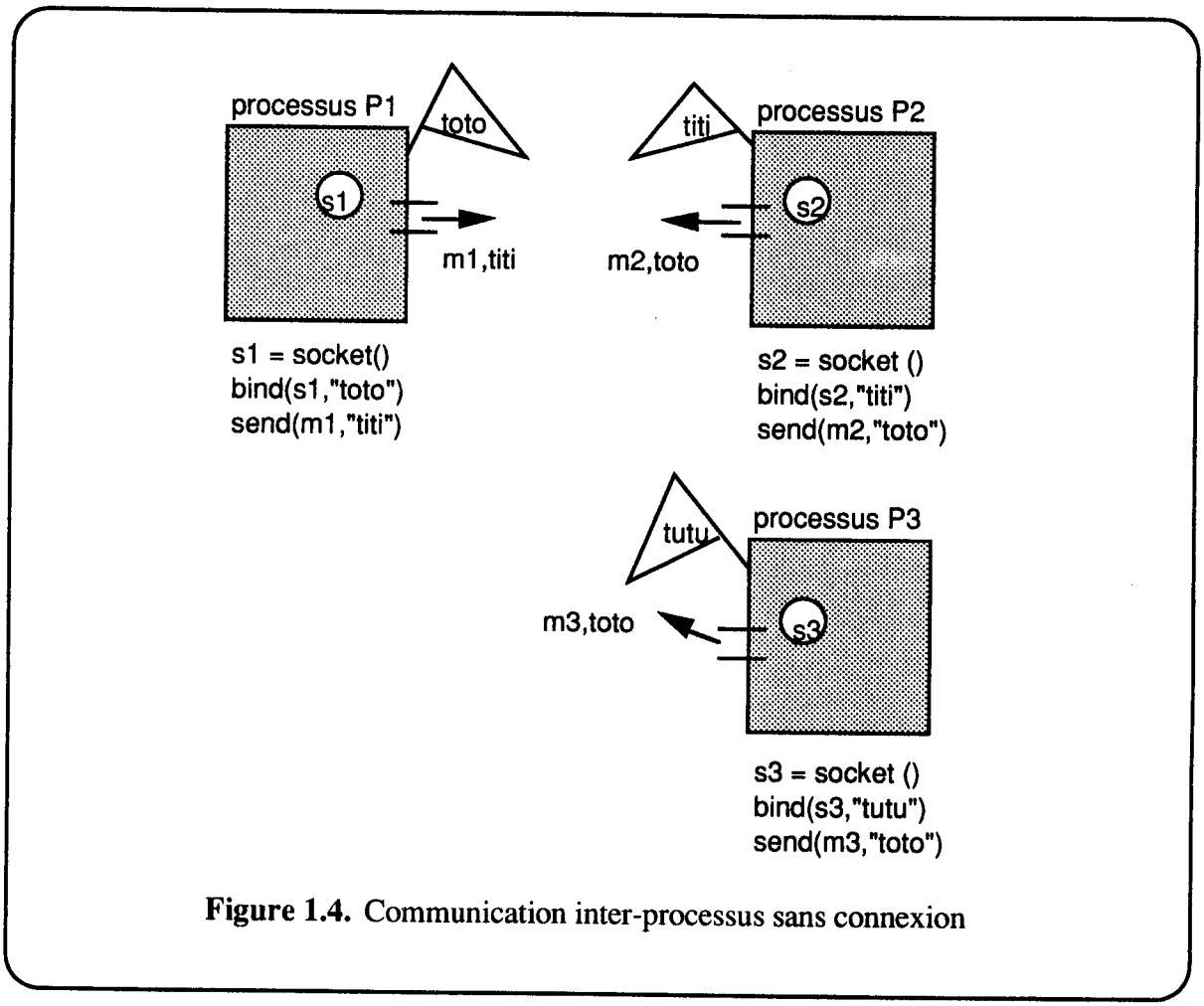



**Figure 1.4.** Communication inter-processus sans connexion

Les messages doivent spécifier l'adresse de la machine. Les ordres de lecture/écriture sont alors "Read from" et "Send to".

Ces deux modes de communication différent essentiellement par leur fiabilité et la nature de l'information transférée.

Grâce à l'utilisation des "sockets", une application peut être répartie sur plusieurs sites (des processus s'exécutent sur des sites différents).

La plupart des applications distribuées du système Unix utilisent l'interface des "sockets" (messageries, gestion d'imprimantes distribuées, ...).

### 2.2.2. L'appel de procédure à distance

L'appel de procédure à distance met en jeu deux processus, *appelant et appelé* pouvant s'exécuter sur une même machine ou sur des machines différentes interconnectées par un réseau de communication. Les relations entre processus distants sont définies par un mode **client/serveur.**

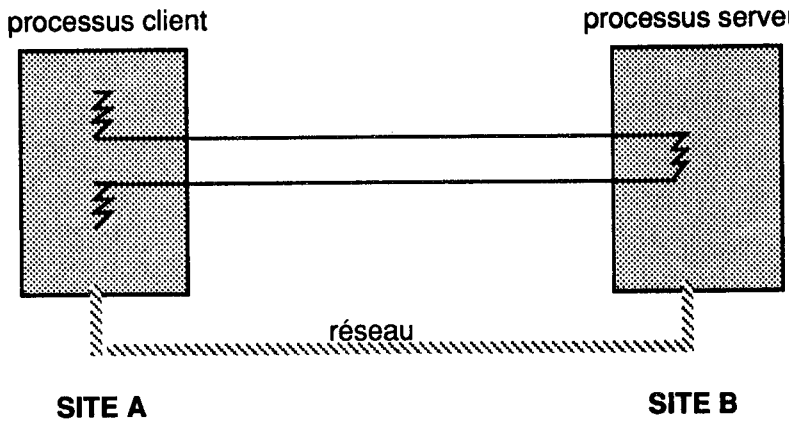



**Figure 1.5.** Communication inter-processus par appel de procédure à distance

L'avantage important de l'appel de procédure à distance par rapport à l'interface des "sockets", est que le canal de communication établi entre le processus client et le processus serveur, n'est plus visible par l'utilisateur : **une procédure définie sur un serveur est appelée comme si elle était locale.**

Par contre, l'appel de procédure à distance laisse moins de liberté dans la synchronisation du dialogue entre les processus : la relation client/serveur est figée pendant tout le temps de la communication.

## 2.3. Choix du moyen de communication dans les applications distribuées

Dans le cadre d'applications distribuées, l'appel de procédure à distance est le moyen le plus évolué et le mieux adapté pour faire communiquer des processus parce qu'il offre une meilleure transparence à la distribution. Nous présentons donc une étude approfondie de ce mécanisme fondée sur une expérience d'utilisation de ce type de service.

# 3. Etude du mécanisme de l'appel de procédure à distance

L'appel de procédure à distance est une extension de l'appel local de procédure, dans le sens où la procédure appelante et la procédure appelée s'exécutent sur des sites différents ou dans des espaces virtuels distincts.

## 3.1. L'appel de procédure locale

Un service d'appel de procédure locale permet à une application d'invoquer l'exécution d'une procédure, les détails de son exécution sont cachés à l'application appelante.

En local, l'appel d'une procédure est de type *synchrone*, c'est à dire que l'application appelante est suspendue durant l'exécution de la procédure appelée et réactivée à la fin de son exécution. La synchronisation est implicitement réalisée par l'intégration lors de la compilation et l'édition de liens, du code de la procédure appelée dans le corps du processus appelant [Mondain 87]. L'intégration des deux modules, appelant et appelé en un seul processus permet d'autre part le partage de mémoire entre ces deux modules, par le biais de déclarations de variables globales. Le passage des paramètres, lors de l'activation et la terminaison de l'appelé, est géré implicitement par le compilateur par l'intermédiaire de la pile d'exécution.

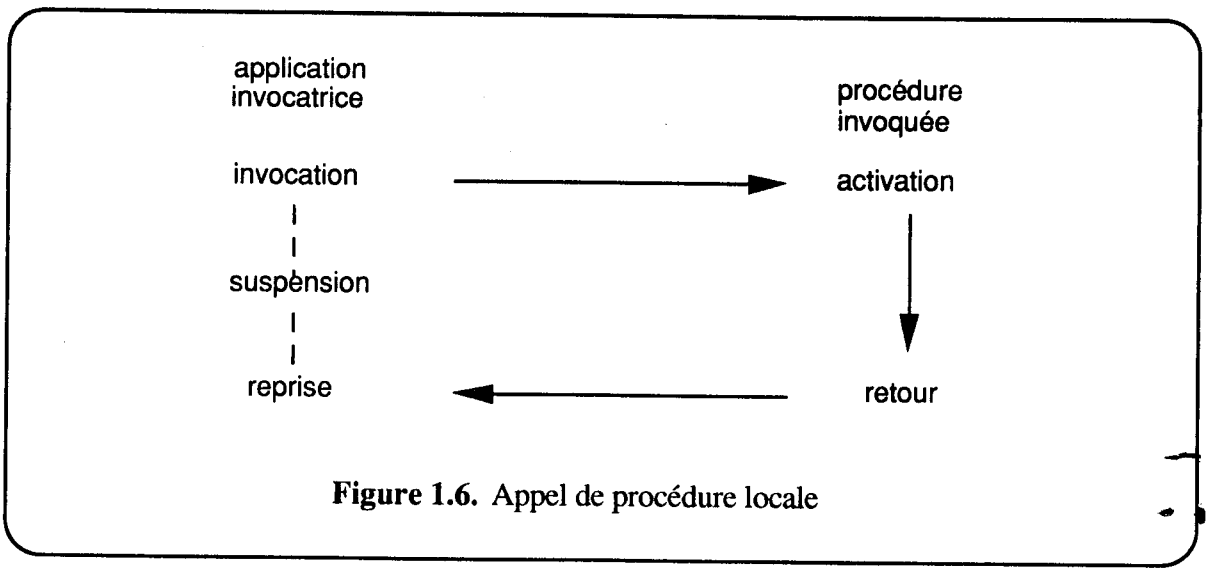



**Figure 1.6.** Appel de procédure locale

## 3.2. Concept général de l'appel de procédure à distance

Le service d'appel de procédure à distance doit permettre à une application d'invoquer l'exécution d'une procédure distante. La réalisation de ce service engendre les problèmes de communication entre deux processus appelant et appelé sur des sites différents, suivant le mode client/serveur. Par conséquent, un service d'appel de procédure à distance réalisant des

interactions inter-processus du type client/serveur, et devant posséder des propriétés identiques à un service d'appel de procédure locale, doit répondre aux spécifications suivantes :

- il doit assurer la gestion des contextes d'exécution distants de façon à préserver leur cohérence mutuelle lors de la défaillance de l'un ou l'autre des sites,
- il doit garantir les transferts physiques pour échanger les données (paramètres et résultats de la procédure),
- il doit gérer la synchronisation des processus distants pour respecter un mode d'interaction procédural (appel/réponse) comme dans le cas de l'appel local.

Un service d'appel de procédure à distance est donc **identique** à un service d'appel local, si ces caractéristiques sont mises en oeuvre de façon transparente à ses utilisateurs.

Les services d'appel de procédure à distance sont de type *synchrone*.

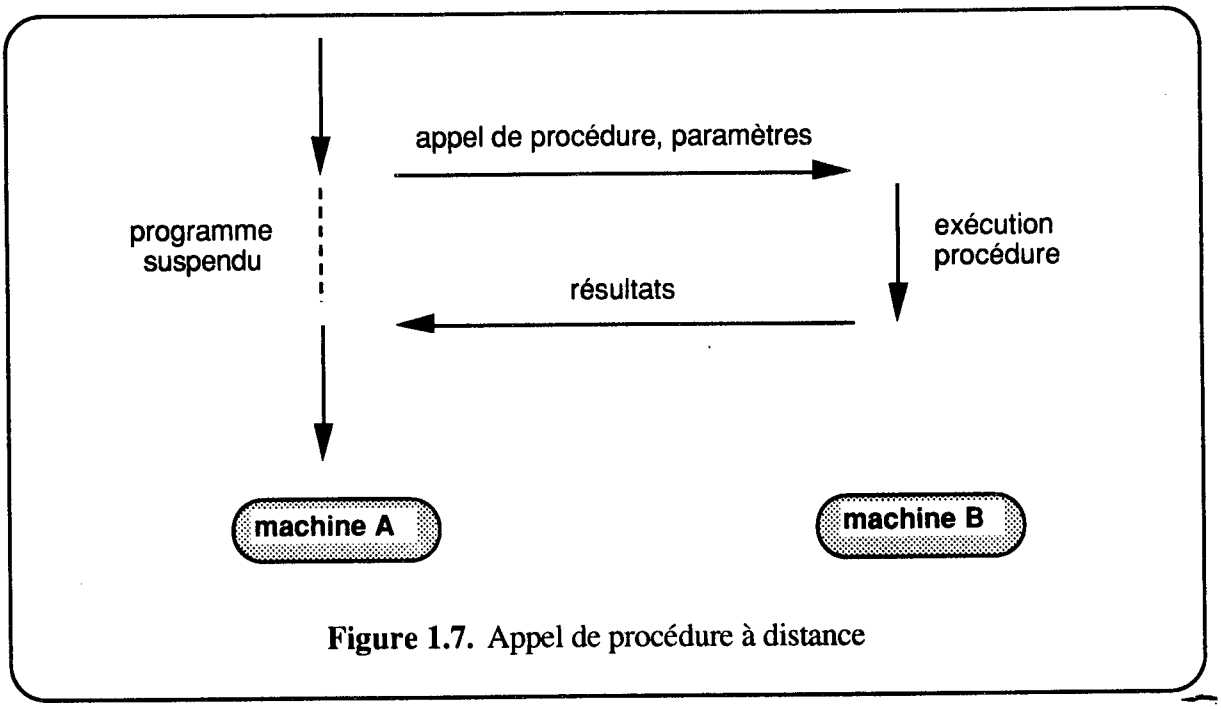



**Figure 1.7.** Appel de procédure à distance

### 3.3. Problèmes dans la réalisation d'un service d'appel de procédure à distance

Pour offrir un service comparable à l'appel de procédure locale, le mécanisme d'appel de procédure à distance doit apporter des solutions à un ensemble de problèmes tels que : la sémantique d'appel, le défaut de communication, la défaillance du client et/ou du serveur, le traitement d'exceptions et l'hétérogénéité des machines et des systèmes.

### 3.3.1. La sémantique de l'appel de procédure à distance

Différentes sémantiques sont envisageables pour prendre en compte des risques de terminaison anormale, due à une défaillance du système de communication de l'un des sites [Nelson 81] :

- **"au moins une fois"** : l'appelant est sûr d'obtenir les résultats de l'appel après l'exécution multiple d'une même procédure. Ceci nécessite un algorithme redemandant l'exécution distante en cas de retard (sur expiration d'un délai de garde), jusqu'à la réception du premier message qui retourne les résultats et éliminant les messages de retour suivants. Tout message d'appel doit donc faire référence à un identificateur unique de procédure.

- **"la dernière fois"** : est équivalent à la sémantique précédente, sauf que l'algorithme doit sélectionner le dernier message de résultats comme message de retour et inclure des numéros d'ordre dans les messages d'appel et de retour.

- **"au plus une fois"** : c'est la sémantique du "tout ou rien". La procédure appelée est sensée s'exécuter une seule fois ou pas du tout; l'algorithme doit détecter les appels dupliqués et assurer l'atomicité de l'exécution.

- **"exactement une fois"** : l'appelant exige la terminaison correcte de l'appelé. Cela nécessite un algorithme de reprise après panne.

On peut remarquer que ces sémantiques impliquent des algorithmes de complexité croissante.

### 3.3.2. Le traitement des exceptions

Lorsqu'un programme a effectué un appel à une procédure qui existe sur un site distant, le déclenchement d'exceptions dans la procédure distante n'est pas détecté par le système local (propre à l'appelant), puisque l'exécution de la procédure s'effectue ailleurs.

Le logiciel d'appel de procédure à distance doit donc inclure un mécanisme pour transmettre l'exception dans l'environnement du programme appelant, afin que celle-ci soit prise en compte par le système local et arrête le programme appelant.

Les fautes contre lesquelles le logiciel d'appel de procédure à distance doit résister, sont de deux sortes :

- les cas exceptionnels dont la détection, la localisation et le traitement sont prévus dans la phase de conception du programme. La solution est de traiter l'exception comme un retour normal de procédure pour l'appelant, comme pour l'appelé. Il suffit de définir un message d'exception envoyé à la place du message de retour normal,

- les fautes non prévues, qui sont souvent des erreurs de conception. Le traitement consiste à les récupérer de façon systématique, de manière à remettre le système dans un état valide pour la poursuite ou la reprise de l'exécution. Aucune solution ne semble résoudre encore ce problème.

### 3.3.3. La présentation des paramètres

La représentation interne des données n'est pas normalisée, elle varie d'une machine à l'autre. Un mécanisme d'appel de procédure à distance doit donc masquer cette hétérogénéité des systèmes. Pour cela, il doit assurer la compréhension syntaxique entre le client et le serveur, en gérant les formats de données à échanger et en effectuant les transformations nécessaires sur les structures de données, pour les rendre compréhensibles entre matériels hétérogènes. Le problème est souvent résolu par la définition d'une nouvelle représentation des données indépendante des machines.

## 4. Etude de cas : le logiciel RPC de Sun

Le produit RPC (Remote Procedure Call), distribué par la société Sun, a été choisi pour le développement d'applications et d'outils, au Centre de Recherche Bull de Grenoble. La raison majeure de ce choix est la large diffusion de l'outil RPC/Sun et par conséquent sa maintenance assurée.

L'utilisation effective du logiciel RPC/Sun dans la réalisation d'applications distribuées, nous a permis de mettre en évidence les caractéristiques générales d'un logiciel d'appel de procédure à distance.

### 4.1. Mise en oeuvre de la transparence de la distribution

Quel que soit le logiciel d'appel de procédure à distance utilisé, le concepteur de l'application n'a pas à se soucier des aspects de communication et de synchronisation : l'appel d'une procédure distante par le client a la même syntaxe ou presque (nous expliquons pourquoi dans le paragraphe 4.3.) que l'appel de procédure locale. La transparence de la distribution entre le client et le serveur est assurée grâce à deux éléments du logiciel [Cooper 83] :

- le protocole d'appel de procédure à distance,
- le langage de description des interfaces et le compilateur pour la génération automatique des **"stubs"**, qui sont des interfaces de communication indispensables entre les processus appelant et appelé.

Le protocole d'appel de procédure à distance est réalisé au-dessus du protocole TCP (ce protocole est décrit en annexe 1). Il a pour fonction de gérer le dialogue entre le client qui appelle des procédures distantes et le serveur qui les exécute.

Un module nommé *stub client* gère la communication au niveau des "sockets" entre le programme client local et la procédure distante. Inversement, le retour de procédure distante s'effectue par l'intermédiaire d'un module appelé *stub serveur*. Ces deux modules sont générés automatiquement par le compilateur du logiciel, à partir de la description des procédures distantes, celle-ci codée dans le langage de spécification du logiciel est à la charge du programmeur. Le module stub client doit être lié au client pour constituer le processus client et le module stub serveur doit être lié au serveur pour constituer le processus serveur.

Pour chacune des procédures appelables à distance, il existe dans le stub client la définition d'une procédure possédant la même syntaxe : même nom et même paramètres. Le corps de ces procédures est constitué d'instructions de codage et décodage (entre la représentation interne de la machine du client et la représentation du logiciel), et d'un appel au protocole d'appel de procédure à distance, pour demander l'exécution des procédures distantes et la réception des paramètres de retour. **Le stub client représente le serveur sur le site client.**

Respectivement, le stub serveur effectue des appels au protocole d'appel de procédure à distance, permettant ainsi de recevoir les appels des clients, de les décoder, de réaliser ces appels dans l'environnement du serveur, d'intercepter les paramètres de retour pour les envoyer vers le client. Ceci est également transparent pour le serveur.

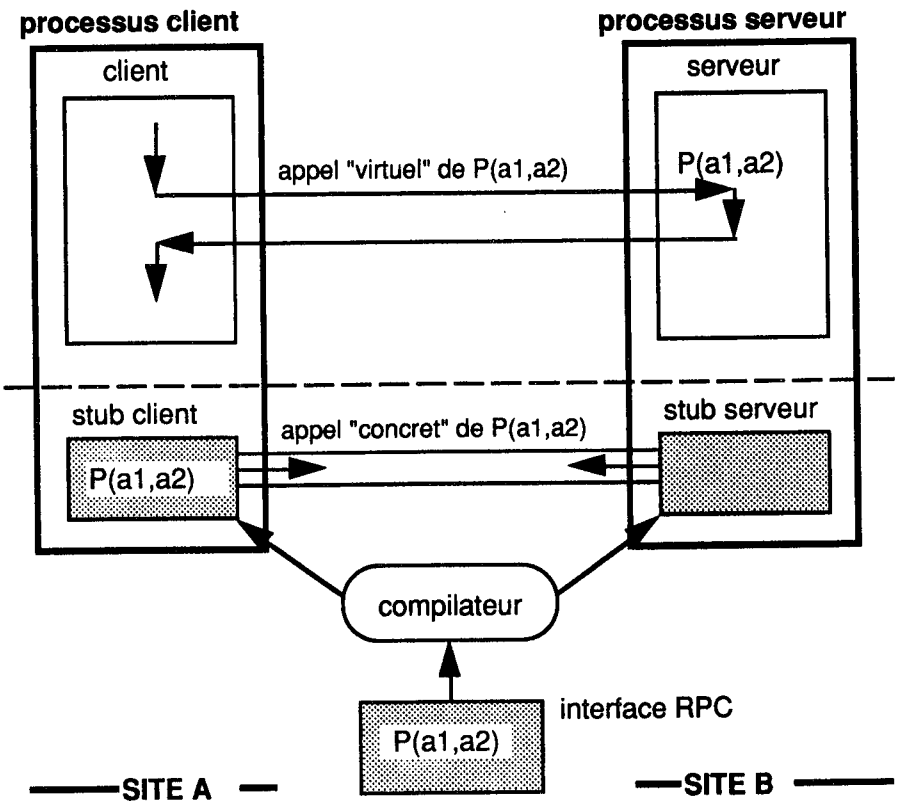



**Figure 1.8.** Fonctionnement des stubs

## 4.2. La représentation des données par RPC/Sun

Pour masquer l'hétérogénéité des systèmes distants, RPC/Sun est couplé avec **le protocole XDR** (eXternal Data Representation), qui transforme les données communiquées en un code de représentation unique, indépendant de tout système matériel (représentation standard).

Le module *stub client* convertit les paramètres d'appel, de la représentation interne de la machine du client à la représentation standard, avant de les transmettre sur la ligne. Après avoir décodé les paramètres en entrée (c'est à dire converti ceux-ci de la représentation standard à la représentation interne de la machine du serveur), le module *stub serveur* réalise l'appel effectif à la procédure désignée. Les paramètres de retour suivent les mêmes règles. Ces séquences de codage et décodage, incluses dans les *stubs*, sont générées automatiquement par le compilateur,

à partir de la description des paramètres d'appel et de retour, faite par le programmeur dans le langage propre au logiciel, qui est associé au protocole XDR. Chaque type élémentaire de ce langage, proche du langage C, tels que les chaînes de caractères, les entiers, les booléens etc, possède dans une librairie propre à chaque machine, à la fois une fonction XDR permettant de convertir la représentation interne de sa valeur en une représentation standard et la fonction inverse. Les structures complexes, pouvant être définies par le programmeur, sont représentées à partir des types de base.

## 4.3. Conditions pratiques d'utilisation

De la pratique de l'outil RPC/Sun, nous avons retiré ses principales caractéristiques techniques (non explicitées dans la documentation fournie par Sun) et son mode d'utilisation.

### 4.3.1. Contraintes de programmation

Nous avons constater que l'outil RPC/Sun impose certaines contraintes au niveau de la programmation des modules client et serveur.

a) Dans le contexte RPC/Sun, **un serveur qui exporte un ensemble donné de services répond aux appels de tous les clients.**

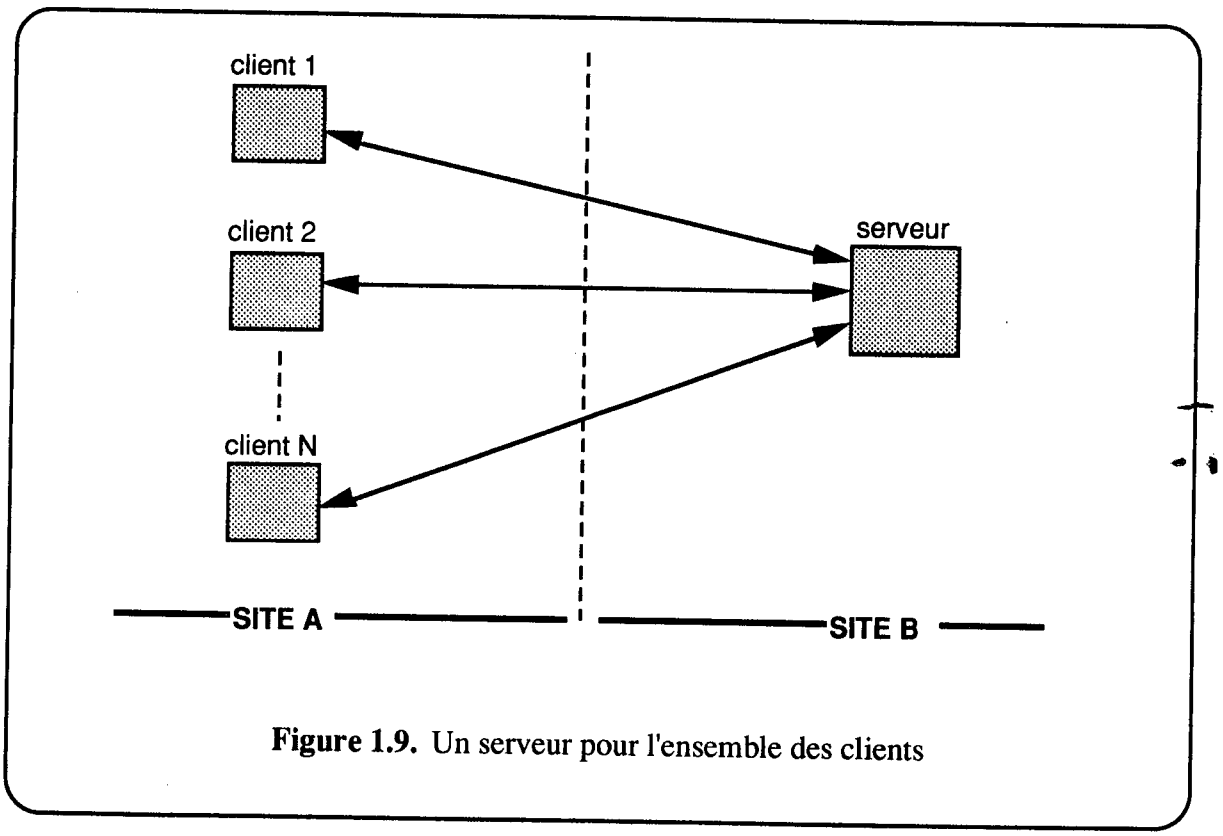



**Figure 1.9.** Un serveur pour l'ensemble des clients

Le programme client demande l'établissement de la communication avec le processus serveur par un appel à la primitive :

**clnt_create (hôte, n° programme, n° version, protocole)**

qui retourne une structure qui identifie le lien établi [RPC 87].
Le premier paramètre **hôte** est le nom de la machine recueillant le module distant. Le couple **(n° programme, n° version)** représente de manière unique un serveur ou service. Si le processus serveur ainsi désigné est actif, la communication est établie. Deux serveurs identifiés par le même couple (n° programme, n° version) ne peuvent pas être actifs en même temps sur la même machine. Ce principe d'identification des serveurs facilite la gestion des versions et des services.
Le dernier paramètre, **protocole** indique le type de protocole de transport utilisé, en l'occurrence *tcp* (ce protocole est décrit en annexe 1).
L'appel à une procédure distante se fait par référence au lien établi. Le nom de chaque procédure est suffixé par "_n° version".
La sémantique d'appel est du type "au plus une fois". La communication client/serveur est détruite lorsque le programme client se termine.

b) Bien que le passage des paramètres par valeur est obligatoire dans le contexte de l'appel de procédure à distance (une adresse n'a de sens que par rapport à un processus), **le logiciel RPC/Sun gère quand même le passage de pointeurs, pour les chaînes de caractères et les structures récursives telles que les listes chaînées et les arbres.**

Nous illustrons cette situation à l'aide d'exemples.

Exemples :

La procédure *imprimer (chaîne)* est exportée par le serveur et est définie comme suit en langage C :

```
imprimer (chaîne)
   char *chaîne ;
   {
      printf ("%s", chaîne) ;
   }
```

Elle imprime la valeur de la variable *chaîne* sur le site serveur.

L'appel par le programme client suivant :

```
...
char *ma_chaîne = "bonjour" ;
imprimer (ma_chaîne) ;
...
```

provoque l'impression de "bonjour" sur la machine serveur. Ce cas de figure ne pose aucun problème.

La procédure *récupérer (chaîne)* permet au client de récupérer la valeur de la variable *chaîne* et est définie comme suit en langage C (passage par adresse) :

```
récupérer (chaîne)
   char **chaîne ;
   {
      *chaîne = "bonjour" ;
   }
```

Si l'appel client est :

```
...
char *ta_chaîne ;
récupérer (@ta_chaîne) ;
...
```

RPC/Sun ne sachant pas qu'il faut restituer la chaîne "bonjour", envoie l'adresse de *ta_chaîne* au serveur. Définie de la manière ci-dessus, la procédure récupérer ne répond pas aux spécifications.

c) En effet avec RPC/Sun, **les paramètres de retour ne sont pas utilisables comme des paramètres d'appel, mais doivent être affectés au type de la procédure.**
La solution à l'exemple précédent se programme de la façon suivante :

```
char *récupérer ()
   {
   char *chaîne;
      chaîne = "bonjour" ;
      return (chaîne) ;
   }
```

L'appel client est alors : `ta_chaîne = récupérer () ;`

Dans le cas de plusieurs paramètres de retour, ceux-ci doivent être regroupés sous la forme d'une structure.

d) **RCP/Sun autorise un seul paramètre en entrée d'une procédure.**

Par conséquent les paramètres d'appel doivent être définis à l'intérieur d'un structure (de la même manière que les paramètres de retour). Cette contrainte implique une certaine gymnastique concernant les déclarations des structures de communication, de plus ces déclarations doivent être faites dans le langage propre au logiciel.

e) Dans les programmes client et serveur, **la manipulation respective des résultats et des paramètres se fait par l'intermédiaire de pointeurs.**

Toutes ces contraintes imposées par RPC/Sun montrent que la transparence à la distribution au niveau du client n'est pas complète.

### 4.3.2. Le langage RPC/Sun

Le concepteur de l'application décrit avec le langage du logiciel RPC/Sun, l'interface exportée par le serveur, c'est à dire les procédures qui peuvent être appelées ainsi que leurs paramètres. Nous présentons les principaux éléments de syntaxe qui caractérisent le langage RPC/Sun [RPC 87].

Les types de données sont séparés en deux grandes classes :

- **les types élémentaires**, qui correspondent aux types généralement admis dans les langages de programmation : integer, unsigned integer, float, double, boolean;

- **les types construits :**

  - constant (correspondance entre un libellé et une valeur entière non signée)

```
const DOZEN = 12 ;
```

  - enumeration (liste de constantes)

```
enum colortype {
      RED = 0,
      GREEN = 1;
      BLUE = 2;
} ;
```

  - array (tableau de valeurs de taille fixe ou variable)

```
colortype palette [8] ;        /* exactement 8 éléments */
int heights <12> ;             /* au plus 12 éléments */
int widths <> ;                /* nombre d'éléments variable */
```

  - string (chaîne de caractères de taille fixe ou variable)

```
string name <32> ;
```

  - opaque (chaîne de bits de taille fixe ou variable)

```
opaque diskblock [512] ;
opaque filedata <1024> ;
```

- pointer (utilisé pour déclarer des arbres et des listes chaînées)

```
listitem *next ;
```

- typedef (définition de types)

```
typedef string fname_type <255> ;
```

- structure (liste de membres nommés)

```
struct coord {
      int x ;
      int y ;
} ;
```

- union (choix entre plusieurs membres)

```
union read_result switch (int errno) {
case 0 :
      opaque data [1024] ;
default :
      void ;
} ;
```

- program (utilisé pour la déclaration des procédures distantes)

Pour déclarer les procédures distantes regroupées sous le même identifiant (n°programme, n°version), on utilise le type *program* dont la syntaxe est la suivante :

```
program-definition :
      "program" program-ident "{"
            version-list
      "}" "=" value
version-list :
      version ";"
      version ";" version-list
version :
      "version" version-ident "{"
            procedure-list
      "}" "=" value
procedure-list :
      procedure ";"
      procedure ";" procedure-list
procedure :
      type-ident procedure-ident "(" type-ident ")" "=" value
```

On suppose les deux procédures exportées, timeget () et timeset (x), qui respectivement donne l'heure en secondes, et règle l'heure avec plus ou moins x secondes. La déclaration de ces procédures s'écrit de la façon suivante :

```
program TIMEPROG {
      version TIMEVERS {
            unsigned int TIMEGET (void) = 1;
            void TIMESET (unsigned int) = 2;
      } = 1;
} = 44 ;
```

Ces instructions génèrent la définition des constantes suivantes :

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Le programme client demande alors l'établissement de la communication avec le processus serveur par l'appel : `clnt_create (hôte, TIMEPROG, TIMEVERS, "tcp")`

### 4.3.3. Construction d'une application en langage C

Le programmeur de l'application doit dans un premier temps, définir les procédures appelables à distance avec leurs paramètres, puis écrire dans le langage de RPC/Sun, l'interface (fichier.**x**), décrivant en même temps les structures passées en appel et en retour, et les procédures distantes (regroupées sous le même identifiant (n° programme, n°version) qui représente le serveur). Enfin, il doit écrire les programmes client (client.c) et serveur (serveur.c).

A partir de l'interface RPC/Sun, le compilateur RPCGEN génère les éléments suivants (commande : **rpcgen** fichier.x) :

- Un fichier (fichier.**h**), regroupant les déclarations des structures de communication, des fonctions XDR de conversion associées et des procédures exportées (le nom de chaque procédure est suffixé par "_n° version"). Ce fichier est inclus dans les autres modules générés et dans les programmes client et serveur.
- Les fonctions XDR de conversion pour les paramètres (fichier_**xdr.c**).
- Le module *stub client* (fichier_**clnt.c**) et le module *stub serveur* (fichier_**svc.c**).

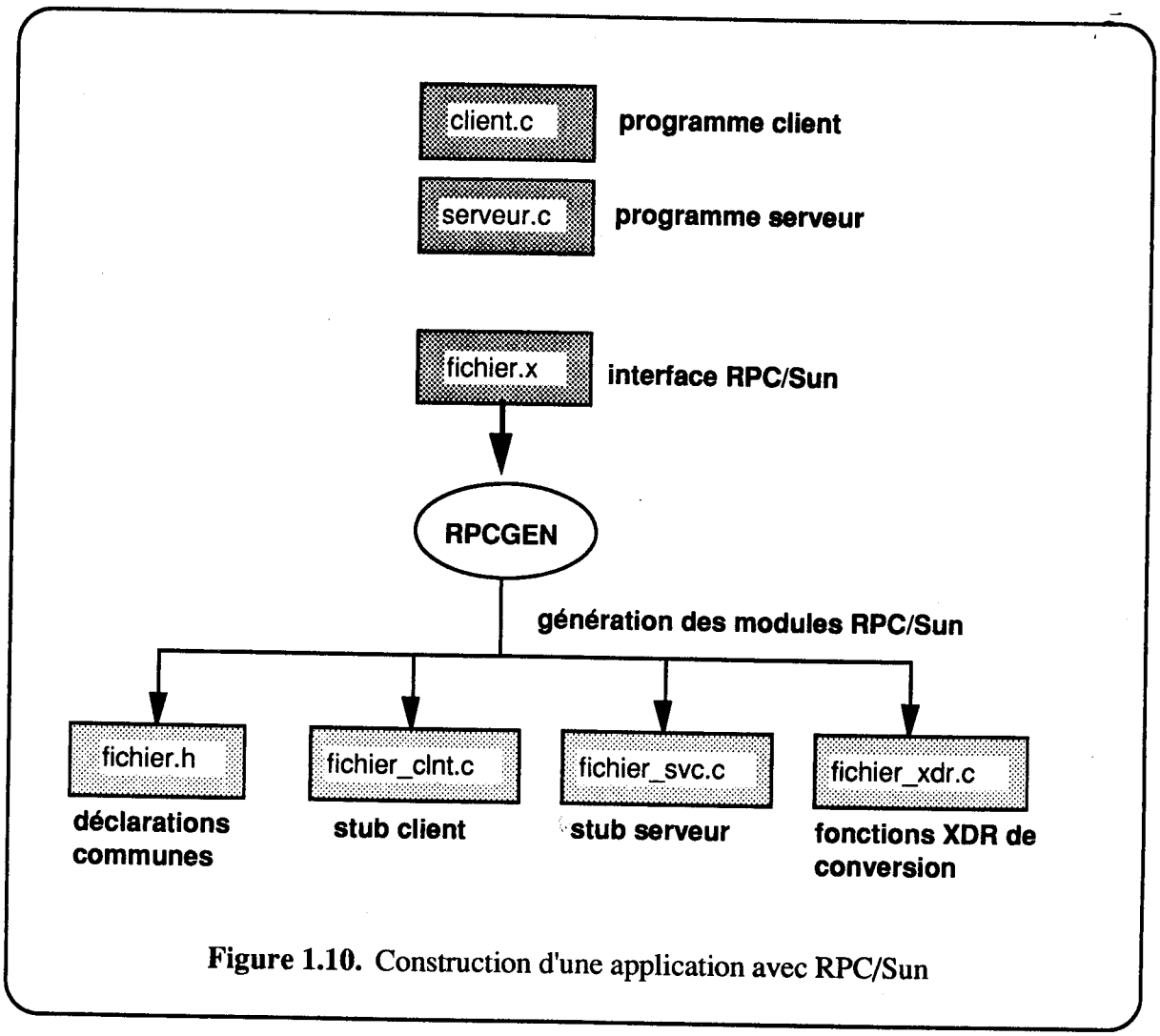



**Figure 1.10.** Construction d'une application avec RPC/Sun

Ensuite les différents modules doivent être assemblés, pour constituer les processus client et serveur :

cc client.c fichier_clnt.c fichier_xdr.c -o **client**
cc serveur.c fichier_svc.c fichier_xdr.c -o **serveur**

Le serveur doit être lancé en "arrière plan" (commande : **serveur &**), de manière à ce qu'il soit actif lors d'un appel d'un client.

Un exemple d'utilisation du logiciel RPC/Sun est donné en annexe 2.

## 5. Conclusion

En pratique, nous constatons qu'un logiciel d'appel de procédure à distance offre un niveau d'abstraction moyen, au programmeur qui l'utilise, contenu des transformations imposées dans la programmation des modules client et serveur. D'autre part, la transparence à la distribution au niveau des utilisateurs des services n'est pas totale : l'appel de procédure distante par le client

n'a pas exactement la même syntaxe que l'appel de procédure locale. Il semble que la transparence à la distribution puisse être améliorée grâce à l'introduction d'interfaces entre les modules client et serveur. Nous détaillons le rôle de ces interfaces, dans le chapitre suivant, dans le cadre de la réalisation d'un serveur pour la communication avec Oracle.

# Chapitre 2

# OPIDUM : un serveur pour l'accès à Oracle

# 1. Introduction

Ce chapitre présente un outil destiné au concepteur d'applications distribuées, utilisant un Système de Gestion de Bases de Données (SGBD) Oracle, pour le stockage des données. Grâce à cet outil, une application résidante sur une machine peut accéder à un SGBD Oracle implanté sur toute autre machine, connectée sur le même réseau de communication : l'accès au données est donc indépendant de la localisation de l'application.

Dans une première partie, nous justifions le choix d'une interface procédurale pour accéder à Oracle, et nous présentons ses caractéristiques. La seconde partie décrit l'implémentation et les choix retenus liés à l'aspect communication.

# 2. Une interface procédurale pour l'accès à Oracle

Afin de mettre en évidence les avantages d'une interface procédurale, nous présentons une brève étude sur les différents outils existants qui permettent d'accéder à un SGBD Oracle, depuis un programme d'application. Mais préalablement, nous décrivons brièvement le fonctionnement du SGBD.

## 2.1. L'exécution d'un programme d'application par le SGBD Oracle

Lorsqu'un programmeur écrit un programme d'application, il le fait uniquement à partir de la connaissance qu'il a de la base de données, c'est à dire au travers d'un schéma **externe** (vues) [Delobel 82]. Le SGBD interprète les instructions exprimées en termes du schéma externe, pour les convertir en termes du schéma **conceptuel** (relations), puis en ordres sur la base de données **physique**. L'architecture d'un SGBD a pour but d'assurer la circulation du flot des ordres et du flot des données. La figure 2.1 illustre ce phénomène, à noter que le flot des ordres va du niveau externe vers le niveau physique, alors que le flot des données va en sens inverse.

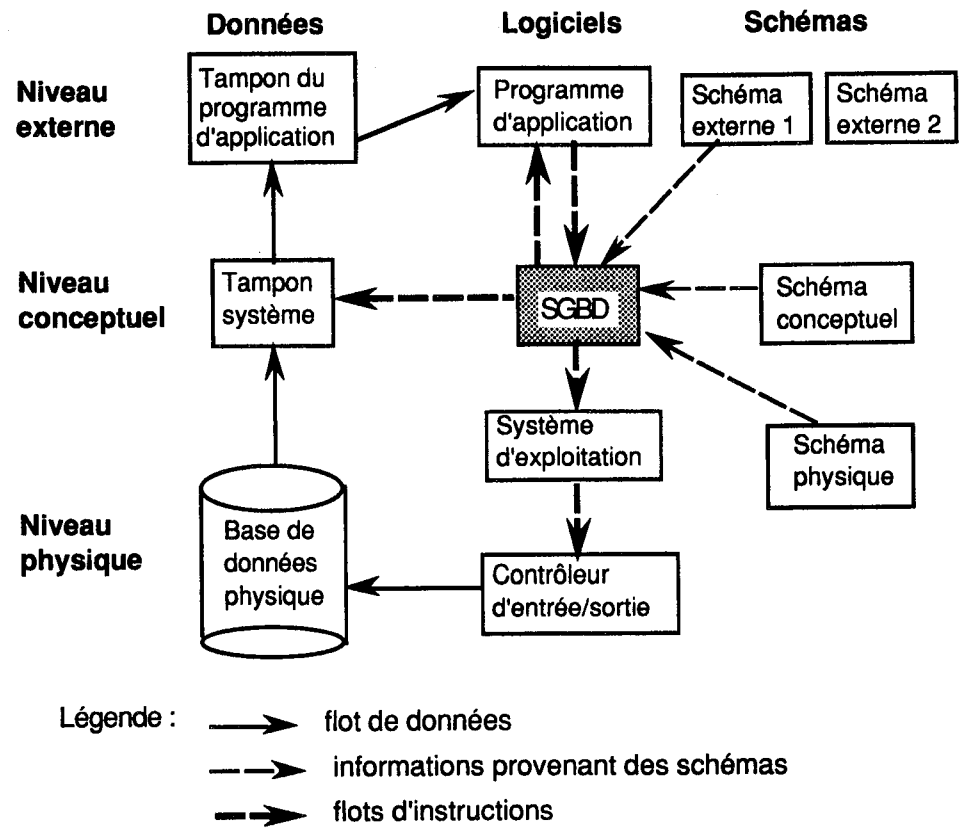



**Figure 2.1.** L'architecture d'un SGBD

L'architecture d'un SGBD fait apparaître les principaux éléments suivants :

- le noyau du SGBD, c'est à dire le logiciel qui assure le bon fonctionnement de la base de données,
- le système d'exploitation de l'ordinateur sur lequel le logiciel du SGBD est construit,
- les différents schémas : externe, conceptuel et physique qui sont stockés sous forme de catalogues internes obtenus à partie des descriptions fournies au système à l'aide du langage de description de données (LDD).

Dans le contexte Oracle, lorsqu'un programme d'application demande la connexion au SGBD, le logiciel Oracle lui alloue une zone mémoire *unique* (correspondant au tampon système), appelé **PGA** (Program Global Area) pour la communication des données [Oracle 87]. Un processus Oracle est alors activé, qui sélectionne parmi l'ensemble des données reçues celles qui sont nécessaires au programme, et transmet ces données dans une zone mémoire appelée **LDA** (Logon Data Area) réservée par ce dernier. **Oracle alloue une et une seule zone PGA**

**par processus** (programme). Par conséquent, Oracle ne sait pas gérer plusieurs connexions dans le même programme.

Pour rappel, SQL (Simple Query Language) est le langage standard des SGBD relationnels. A chaque ordre SQL exécuté par le programme, est associé une zone de données, appelée **curseur** qui contient les informations relatives au statut de la requête en cours. Dans un même programme, plusieurs curseurs peuvent être ouverts simultanément.

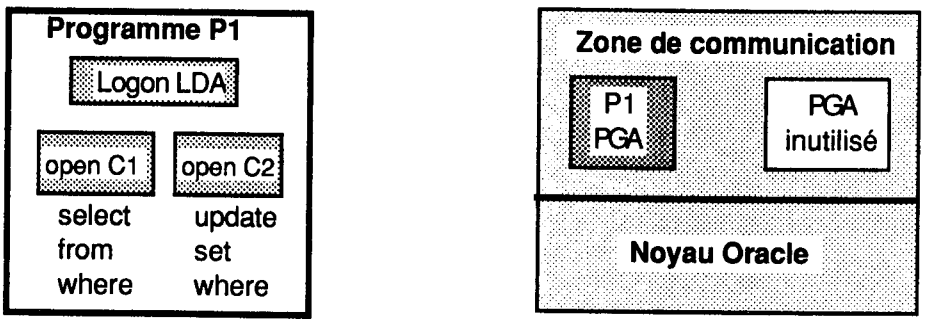



**Figure 2.2.** Communication avec le SGBD Oracle

Dans le cas où plusieurs programmes se déroulent en parallèle, il appartient au SGBD de gérer les accès concurrents sur les données.

## 2.2. Choix d'une interface procédurale

Dans le cas de développements de logiciels complexes, avec de nombreux accès à Oracle, la première possibilité consiste à utiliser l'interface **HLI** (High Level Interface) d'Oracle [Soupe 88]. Une liste de fonctions est mise à la disposition du programmeur, celles-ci lui permettent d'accéder à un SGBD Oracle depuis un programme d'application. Mais ces fonctions HLI sont proches du code généré par Oracle, par conséquent, l'écriture des programmes devient plus complexe à cause des multiples appels de fonctions nécessaires et des nombreuses structures de données à connaître et à manipuler (LDA, curseur, ...).

Un ordre de lecture sur la base de données, se programme sous la forme d'une succession d'appels aux primitives HLI comme le montre la figure 2.3 :

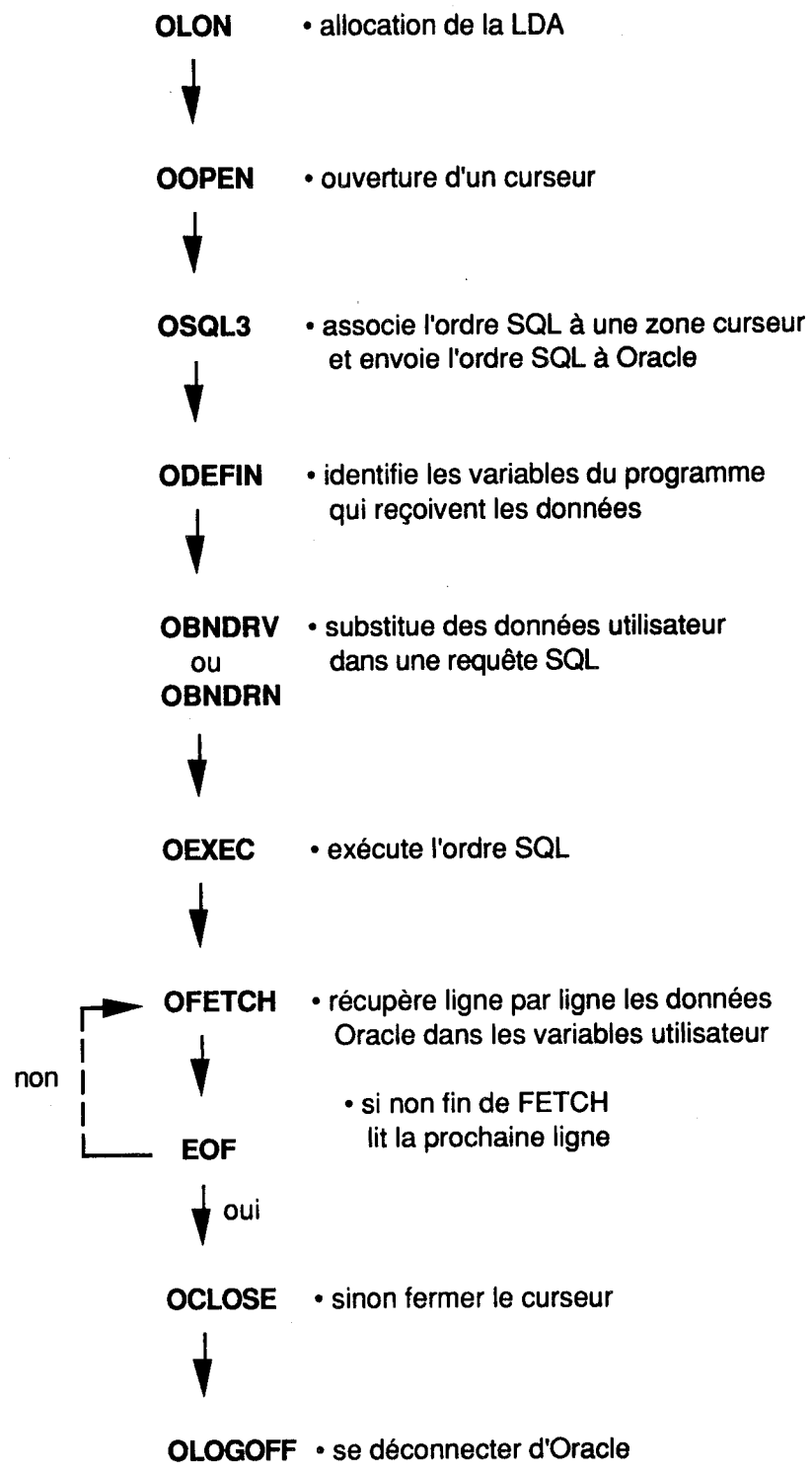



**Figure 2.3.** Programmation HLI d'un ordre de lecture sur la base

La seconde possibilité est l'utilisation du **pré-compilateur** d'Oracle. Des ordres SQL peuvent être incorporés à un programme écrit en langage de programmation tel que C, Pascal, Cobol ou Fortran. Par exemple, la connexion à Oracle se programme de la façon suivante :

```
EXEC SQL CONNECT : uid IDENTIFIED BY pwd
```

ou *uid* et *pwd* sont des variables contenant respectivement le nom de l'utilisateur et son mot de passe.

Dans ce cas, le programme subit une phase de pré-compilation pour transformer les ordres SQL en appels au SGBD. Le programme résultant est ensuite compilé. La figure 2.4 illustre cette situation.

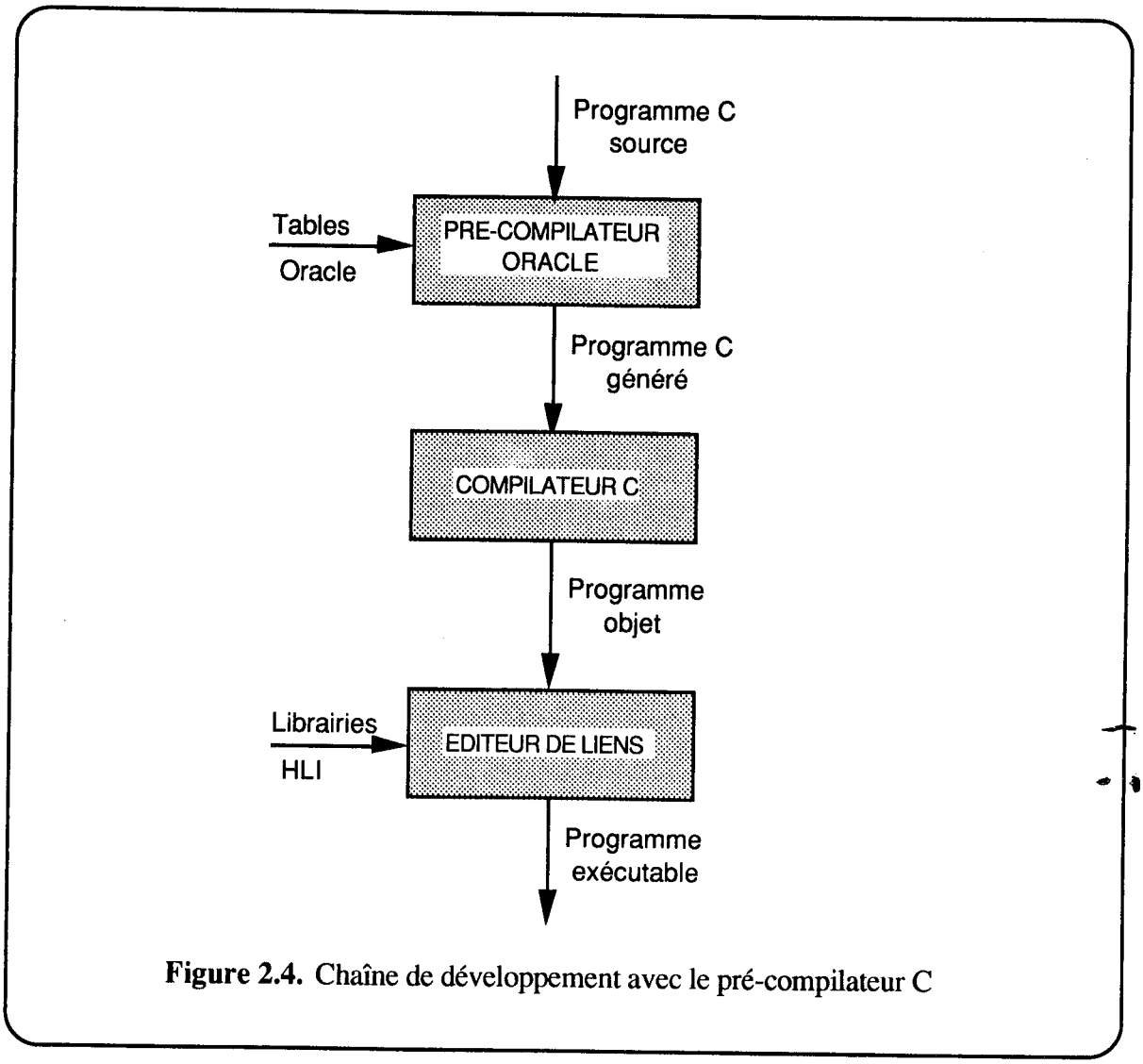



**Figure 2.4.** Chaîne de développement avec le pré-compilateur C

L'utilisation du pré-compilateur complique l'organisation des fichiers, multiplie leur nombre et augmente significativement les temps de compilation (pré-compilation plus compilation proprement dite).

Les objectifs d'une interface procédurale sont donc, d'une part de supprimer la phase de pré-compilation, et d'autre part de fournir au programmeur des fonctions simples à utiliser et qui suffisent à la plupart des applications de bases de données. La solution est la définition d'une couche logicielle construite sur l'interface HLI d'Oracle, qui propose une syntaxe proche des fonctions **printf** et **scanf** de la librairie standard du langage C.

L'interface ainsi définie, est baptisée OPIDUM qui signifie Oracle Procedural Interface for Distributed Unix Machines, et réunit donc les avantages du pré-compilateur Oracle (expression des requêtes SQL en clair) et de l'interface HLI (élimination de la phase de pré-compilation). Le cycle de développement des applications se trouvent ainsi écourté. La figure 2.5 illustre cette situation :

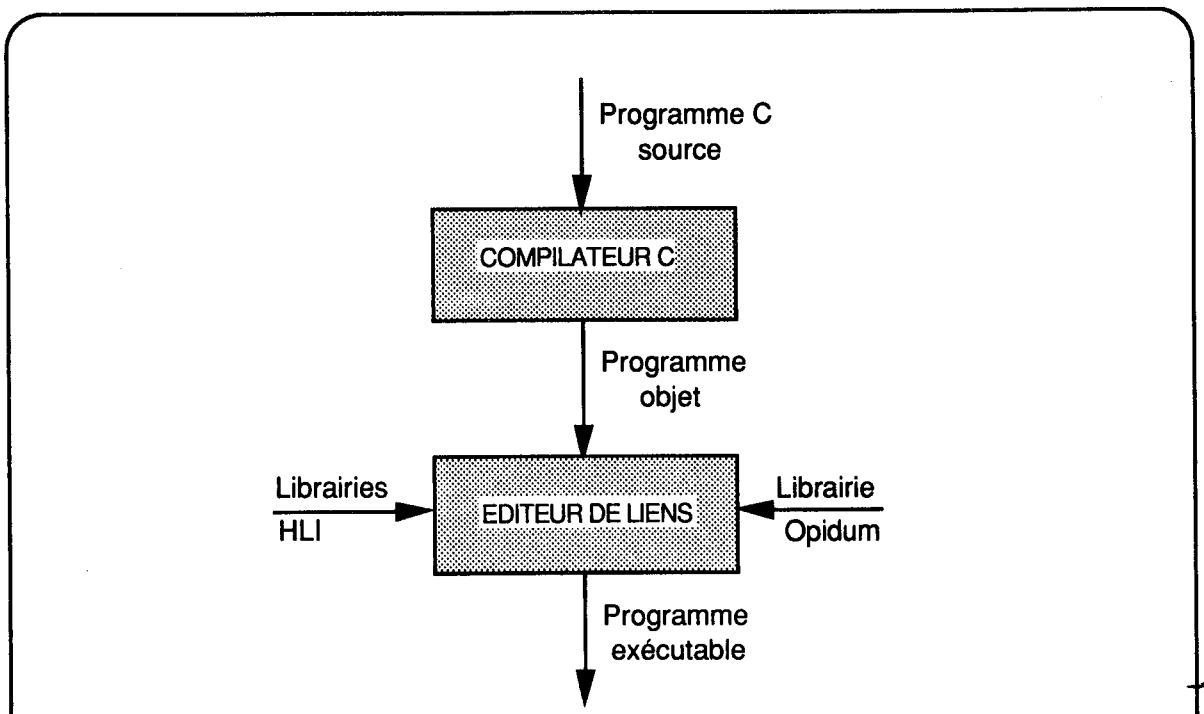



**Figure 2.5.** Chaîne de développement avec Opidum

## 2.3. Principe d'utilisation d'OPIDUM

Le logiciel Opidum propose au programmeur une dizaine de fonctions, qui satisfont à la totalité de ses besoins. La requête SQL en clair, est passée en paramètre à la fonction Opidum, qui enchaîne les différents appels aux fonctions HLI, nécessaires à l'exécution de la requête. Par un

mécanisme de substitution de variables à l'appel, les requêtes peuvent être paramétrées. La lisibilité des programmes est ainsi accrue, et la maintenance facilitée. L'utilisation de ces fonctions est simple et nécessite peu de temps d'apprentissage. L'exemple de programme présenté comme-suit, montre différentes utilisations des fonctions **connection** et **sql** :

```
char nom[21], prenom[21];
...
traiter_tuple();
{
 printf("nom : %s prenom : %s (%d)\n", nom, prenom, age);
 return (1);
}
...
main()
{
 int age;
 ...
 if (!connection (1, "test", "test")) /* connexion au SGBD Oracle */
 {
       printf ("Connexion impossible"\n);
       exit (2);
 }
 /* récupération automatique de résultats à l'écran */
 age = 30;

 sql ("select * from demopidum where age < %i", age);

 /* mise à jour de l'âge de la personne "DUPONT" */

 sql ("update demopidum set age = %i where nom = 'DUPONT'", 40);

 /* récupération de valeurs dans des variables et affichage des
    résultats par fonction */

 sql ("select nom &20, prenom &20s, age &i from demopidum {f}",
                  nom,        prenom,  &age,          traiter_tuple);
 ...
 connection (0); /* déconnexion d'Oracle */
 ...
}
...
```

La description des fonctions Opidum et leur utilisation est donnée en annexe 3 dans la documentation pour le programmeur.

Une première version de cette interface procédurale, appelée SOUPE (Simple Oracle User Programming Extension), avait été réalisé au centre de recherche Bull, en 1985. Mais contenu de son nouvel aspect distribué, le code a été entièrement repris.

## 3. Choix et implémentation

Un SGBD Oracle implanté sur une machine du réseau de communication, doit être accessible par toute application résidante sur n'importe quelle autre machine de ce même réseau : cela est possible grâce à un serveur présent sur la même machine que le SGBD, et exportant l'interface procédurale, décrite précédemment. Le serveur doit mettre à la disposition du programmeur, les mêmes fonctions que l'interface procédurale en local. De plus, il doit permettre l'accès à Oracle, à plusieurs programmes clients en parallèle.

Nous présentons dans cette partie, les choix faits et les techniques utilisées dans la mise en oeuvre du serveur OPIDUM.

### 3.1. Choix du modèle client/serveur

Deux types de schéma client/serveur sont envisageables :

- un serveur pour l'ensemble des clients ou **serveur à états**
- un serveur par client ou **serveur sans état**

Dans le cas du serveur à états, les réponses du serveur aux clients peuvent être inter-dépendantes : une application de gestion de ressources dans laquelle, la réponse du serveur au client courant dépend des réponses aux clients précédents, serait implémentée suivant ce mode. Dans le cas du serveur pour Oracle, cette caractéristique du serveur à états n'est pas utile.

De plus, ce schéma présente un inconvénient majeur : lorsqu'un programme client fait un appel à distance au serveur, demandant le connexion à Oracle, un processus Oracle est créé liant la zone LDA du programme serveur, et la zone PGA associée à celui-ci. Si parallèlement, un autre programme client désire se connecter à Oracle, sa demande est refusée parce que la zone PGA du programme serveur est déjà allouée. (Oracle alloue une et une seule zone PGA par programme). Le serveur ne peut donc satisfaire qu'un seul client à la fois, cette contrainte va à l'encontre de ses fonctionnalités.

C'est donc le schéma **un serveur par client** qu'il faut prévoir dans notre cas. Or, le logiciel RPC de Sun qui doit être utilisé dans l'implémentation, fonctionne suivant le premier schéma (un serveur est identifié par un couple (n° programme, n° version) affecté une fois pour toutes),

par conséquent il faut définir un moyen pour réaliser le schéma, un client par serveur avec l'outil RPC. La solution proposée est la suivante :

- un serveur général (daemon), unique pour une machine hôte, lancé en arrière plan, centralise les appels des clients,
- les clients demandent d'abord la connexion au serveur général,
- à chaque appel d'un client, le serveur général active un nouveau serveur Opidum identifié par un couple (n° programme, n° version),
- dés que le serveur général a la certitude que le serveur Opidum est actif, il retourne au client le couple (n° programme, n° version), identifiant le serveur qui lui est rattaché,
- le client demande alors, la connexion à son serveur rattaché,
- le client, lorsqu'il sait ne plus avoir besoin du serveur Opidum, envoie au serveur général, une demande de déconnexion
- le serveur général détruit alors, le serveur Opidum affecté à ce client.

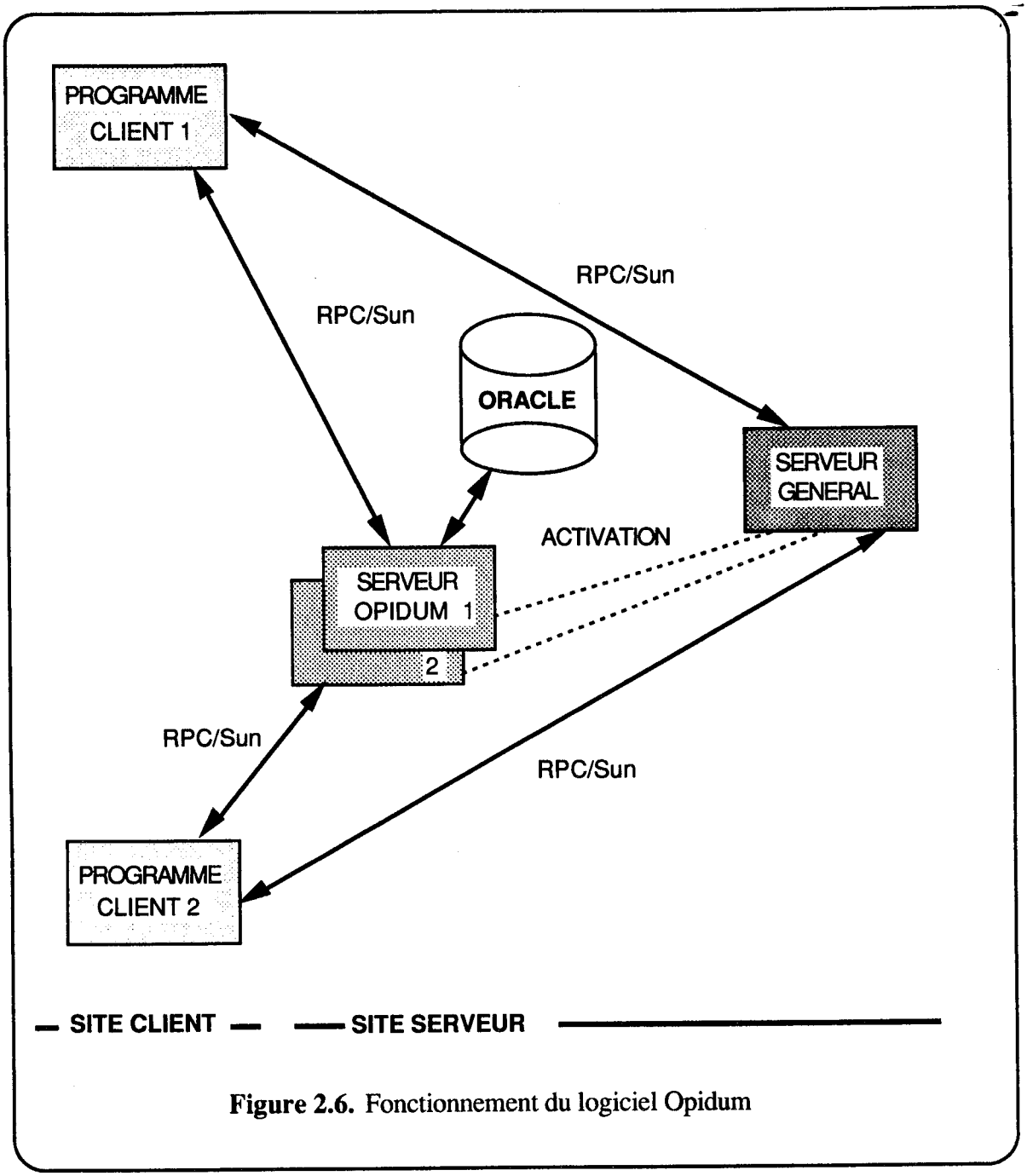



**Figure 2.6.** Fonctionnement du logiciel Opidum

Ce type de fonctionnement, liant un serveur à un client ne pénalise que le client rattaché au serveur en cas de problème de ce dernier.

Ainsi, la communication entre le programme client et le SGBD Oracle s'opère de la même façon qu'en local : le processus serveur Opidum rattaché au client est lié à Oracle à travers une zone PGA, et assure la transparence au niveau des échanges de données et des accès concurrentiels à la base.

## 3.2. Mise en oeuvre du modèle : un serveur par client

La solution proposée peut s'appliquer à n'importe quelle application distribuée, nécessitant un serveur sans état : il ne s'agit donc pas d'une solution "ad hoc", mais d'une solution générale.

### 3.2.1. Solution générale

Le fonctionnement du serveur sans état est implémentée de la manière suivante :

- le processus serveur général, toujours en attente d'un appel d'un client, gère une table de N serveurs Opidum. On ne connaît pas à l'avance le nombre exact de clients, susceptibles de demander l'accès à Oracle simultanément, ni par conséquent le nombre maximum de serveurs Opidum à prévoir. La dimension N de la table des serveurs doit être un paramètre de l'application, modifiable à la demande,
- c'est le client qui demande l'établissement de la communication avec le serveur général,
- le serveur général parcourt alors la table, pour repérer le premier serveur Opidum non actif (p, v), qui sera affecté au client courant,
- pour activer le serveur Opidum (p, v), le serveur général exécute un appel à la fonction **fork**, pour créer un nouveau processus, appelé processus "fils". Ce dernier est la réplique exacte (même code et même contexte) de son processus créateur, en l'occurrence du processus serveur général, mais il a un identificateur de processus propre. Le processus "fils" demande alors sa transformation en processus serveur Opidum (p, v), par un appel à la fonction **execl.** Ce nouveau processus est créé à partir du fichier exécutable du serveur Opidum, auquel le couple (p, v) est passé en argument,
- le serveur général verrouille l'élément (p, v) dans la table des serveurs,
- il s'assure que le serveur Opidum (p, v) est bien actif (se référer au paragraphe 3.2.2), avant de retourner au client les informations (n° programme = p, n° version = v) relatives au serveur qui lui est rattaché,
- le programme client demande alors l'établissement de la communication avec le serveur (p, v),
- lorsque le client se déconnecte d'Oracle, il signale au serveur général qu'il peut tuer le serveur Opidum (p, v),

- le serveur général exécute alors un appel à la fonction **kill**, pour envoyer le signal SIGKILL au processus serveur Opidum qu'il désire détruire. La réception de ce signal, qui ne peut être ignoré, provoque la mort du processus désigné (se reporter au paragraphe 3.2.2).

- enfin, il déverrouille le serveur (p, v) dans la table, qui peut alors être affecté à un nouveau client.

La première possibilité, pour implémenter la communication inter-processus entre le client et le serveur général, est l'utilisation de l'interface des "sockets". Mais pour des raisons de commodités de programmation, il est plus intéressant d'utiliser l'appel de procédure à distance.

Les deux procédures **n°serveur()** et **kill_n°serveur()** sont exportées par le serveur général, et sont appelables par les programmes clients :

- la procédure *n°serveur()* effectue le lancement du serveur Opidum pour le programme client appelant,

- la procédure *kill_n°serveur()* détruit le serveur Opidum rattaché au programme client appelant.

Mais dans un premier temps, l'interface RPC/Sun (**daemon.x**) qui décrit les procédures distantes est à définir :

```
program DAEMON_PROG {
 version DAEMON_VERS {
       int N°SERVEUR (void) = 1;
       int KILL_N°SERVEUR (int) = 2;
 } = V;
} = P;
```

Les procédures distantes définies dans le module serveur **daemon.c**, sont décrites comme suit par leur algorithme simplifié :

```
Inclusion du fichier des déclarations communes : daemon.h
Déclaration de la table des serveurs S(N) en variable globale
...
procédure n°serveur_V()
début
| Repérage du premier serveur Opidum disponible identifié par (P,V+i)
| avec i =1,N
| Appel à la fonction fork pour créer un nouveau processus
| si (Branche du processus fils)
| alors
|   | Appel à la fonction execl pour transformer le processus fils en
|   | serveur Opidum (P,V+i)
| finsi
| si (Branche du processus père)
| alors
|   | Verrouillage du serveur (P,V+i) dans la table S des serveurs
|   | Stockage de l'identificateur de processus associé
|   | Envoi de l'identité i du serveur Opidum au client
| finsi
fin

procédure kill_n°serveur_V (i)
début
| Récupération  de l'identificateur de processus du serveur Opidum à
| tuer dans la table S des serveurs
| Appel à la fonction kill pour tuer le processus désigné
| si (la destruction est OK)
| alors
|   | Déverrouillage du serveur Opidum identifié par (P,V+i)
|   | Envoi du message "OK" au client
| sinon
|   | Envoi du message "NOT OK" au client
| finsi
fin
```

Le programme client est défini comme suit par son algorithme simplifié :

```
Inclusion du fichier des déclarations communes : daemon.h
...
début
| Récupération du nom de la machine hôte
| Demande de connexion avec le serveur général :
| Lien_daemon = clnt_create (hôte, DAEMON_PROG, DAEMON_VERS, "tcp")
| si (Connexion établie avec le serveur général)
| alors
|   | Appel à la procédure distante n°serveur_V :
|   | i = n°serveur_V (NULL, lien_daemon)
|   | si (Serveur Opidum (P,V+i) activé)
|   | alors
|   |   | Demande de connexion avec le serveur Opidum (P,V+i) :
|   |   | Lien_opidum = clnt_create (hôte, OPIPROG, OPIVERS + i,"tcp")
|   |   | si (Connexion établie avec le serveur Opidum)
|   |   | alors
|   |   |   | Appels aux procédures exportées par le serveur Opidum
|   |   |   | ...
|   |   |   | Appel à la procédure distante kill_n°serveur_V :
|   |   |   | Message = kill_serveur_V (i, lien_opidum)
|   |   | finsi
|   | finsi
| finsi
fin
```

### 3.2.2. Problèmes rencontrés

#### a) La synchronisation de processus

Le serveur général doit s'assurer que le processus serveur Opidum est effectivement actif, avant de retourner son identité au client : cela pose le problème de la synchronisation de deux processus, que l'on a résolu dans notre cas, par l'utilisation de la fonction **pipe**.

La fonction pipe crée un mécanisme d'entrée-sortie, appelé *tube* [Bourne 85]. Après la création du tube, deux (ou plus) processus (créés par un appel de la fonction fork) vont se transmettre des données au travers du tube.

Dans notre cas, la solution est la suivante :

- le processus serveur général exécute un appel à la fonction **pipe (fichdesc)**,
- il lance le serveur Opidum, qui lorsqu'il est actif, range le message "OK" en entrée du tube, par l'ordre **write (fichdesc[1], "OK", 2)**,

- parallèlement, le serveur général se met en attente de lecture du message écrit dans le tube, par l'ordre **read (fichdesc[0], buf, 2)**,

- dés que le message "OK" est transmis en sortie du tube, le serveur général obtient le message, puis reprend son exécution pour retourner l'identité du serveur Opidum au client.

### b) Les processus zombies

Un processus se termine toujours par un appel à la fonction **exit** (l'appel peut être explicite ou implicite), puis il se transforme en processus zombie.

Un processus zombie est un processus n'occupant qu'une entrée dans la table des processus, aucun autre espace ne lui est affecté dans l'espace utilisateur ou noyau [SystemV 87]. Le processus, qui se termine, envoie un signal de "mort d'un fils", à son processus père. Si ce dernier est en train d'exécuter un appel système **wait**, il est informé de la fin du processus fils, et le noyau du système libère alors l'entrée du processus fils dans la table des processus. Ce mécanisme a été prévu, de manière à ne pas encombrer inutilement la table des processus, par des processus "morts-vivants".

Or l'appel de la fonction *wait* interrompt le processus père jusqu'à la réception du signal envoyé par son fils qui se termine. L'exemple suivant montre que l'appel système *wait* , pour supprimer les processus zombies n'est pas toujours opportun :

```
programme daemon
début
   Appel à la fonction signal : signal (SIGCLD,SIG_IGN)
   tant que (Appel d'un client)
   faire
      Appel à la fonction fork  pour créer un nouveau processus
      si (Branche du processus fils)
      alors
         ...
         Appel à la fonction exit
      finsi
   finfaire
fin
```

A chaque appel d'un client, le daemon doit aussitôt créer un processus fils. Si le programme daemon exécute un appel à la fonction *wait*, il reste bloqué jusqu'à la réception des signaux des processus fils précédents qui se terminent, par conséquent il ne peut pas activer un nouveau processus. Dans ce cas, le processus père ne doit donc pas exécuter d'appel *wait* mais il doit faire un appel à la fonction **signal** pour ignorer les signaux "mort d'un fils" envoyés par les

processus fils. Ainsi, le noyau libère les entrées dans la table des processus et le processus père n'est pas bloqué.

Nous nous trouvons en présence du même cas de figure dans le fonctionnement en mode un serveur par client, mis à part que les processus fils sont tués par le processus père (appel implicite à la fonction exit). De la même façon, le serveur général doit donc faire un appel à la fonction *signal*.

### c) Les dépendances système

Deux souches du système Unix sont actuellement commercialisées : la version System V distribuée par la société AT&T, et la version Berkeley de l'université du même nom. Il existe des différences, concernant les commandes et le système, entre les deux origines. Particulièrement, le problème lié aux zombies ne se résout pas de la même manière, dans les deux systèmes.

La solution exposée précédemment est celle proposée par System V, installé sur les ordinateurs Bull. Dans la version Berkeley, implantée sur les matériels Sun et Digital, cette solution n'est pas valide. Elle peut être remplacée par un appel à la fonction **wait3** dans le processus père [Berkeley 86]. En effet, la fonction wait3 ne provoque pas de blocage du processus appelant, en attendant le signal de fin d'un processus fils.

Ce problème de dépendances système nous amène à adapter la procédure d'installation du logiciel, de sorte à assurer sa portabilité. La commande de compilation du programme serveur général est de la forme :

**SYSTEM=***nom du système Unix hôte*
**cc -D$(SYSTEM) -c daemon.c**

**$(***var* **)** a pour effet de rendre la valeur de la variable *var*, et l'option **-D***nom* réalise pour le compilateur la définition de *nom* (de la même manière qu'un *#define* inclus dans le fichier source), et la valeur de *nom* par défaut est à vrai . Le programme source du serveur général contient les directives de compilation conditionnelle suivantes :

```
#ifdef BSD
      printf ("zombies à la Berkeley\n");
      wait3;
#endif
#ifdef S5
      printf ("zombies à la System V\n");
      signal (SIGCLD, SIG_IGN);
#endif
```

de manière à provoquer la compilation conditionnelle du programme serveur général, en fonction de la version du système Unix.

## 3.3. Architecture du serveur Opidum

Nous présentons dans ce paragraphe l'architecture du serveur Opidum avec les extensions apportées pour le mode distribué.

### 3.3.1. En local

Le serveur Opidum met à la disposition du programmeur, des fonctions qui permettent d'établir l'interaction entre le programme d'application et la base de données : ouverture et fermeture du dialogue, contrôle des transactions, accès aux données.

Les fonctions dont dispose le programmeur sont les suivantes :

- une fonction de connexion et déconnexion à Oracle :
  **connection (on/off, user, pwd)**,
- deux fonctions pour le contrôle de la validation des transactions :
  **commit (on/off)** et **autocommit (on/off)**,
- et un ensemble de fonctions, permettant d'exécuter n'importe quel ordre SQL.

Chacune de ces fonctions est définie comme une succession d'appels HLI, qui s'exécutent partiellement ou complètement sur le site d'accueil du SGBD. Par exemple, la fonction *connection* est définie de la manière suivante ( les ordres HLI sont écrits en gras) :

```
procédure connection (on/off, user, pwd)
début
| Déclarations : lda, cursors
| si (on)
| alors
|   | Etablissement de la communication avec Oracle à travers une zone LDA
|   | olon(lda, user, pwd, ...)
|   | Annulation de la validation immédiate des transactions : ocof(lda)
|   | Allocation de zones curseurs : oopen(cursors)
| sinon
|   | Libération des ressources Oracle allouées pour l'ouverture des zones
|   | curseurs : oclose(cursors)
|   | Validation de toutes les transactions : ocom(lda)
|   | Déconnexion d'Oracle et libération des ressources allouées pour le
|   | programme : ologof(lda)
| finsi
fin
```

Les requêtes de mise à jour et de consultation de la base mettent en jeu deux phases :

- la compilation de la requête ( ordre HLI : **osql3**) après la substitution éventuelle des paramètres en entrée dans la requête (c'est la fonction **sqls** qui effectue la phase de substitution des paramètres),
- et l'exécution de la requête (ordre HLI : **oexec**),

qui sont réalisées par la fonction **sqlb.**

L'interrogation de la base nécessite une étape supplémentaire de récupération des tuples répondant à la sélection, c'est le rôle de la fonction **sql_fetch.** La restitution des résultats lors d'un ordre SELECT peut être réalisé de deux manières :

- à la demande, le programmeur gère lui-même la récupération des tuples par des appels à la fonction sql_fetch (un appel à la fonction sql_fetch est toujours précédé d'un appel aux fonctions sqlb ou sqls),

- ou automatiquement, à l'aide de la fonction *sql* : les résultats s'affichent directement à l'écran, ou bien sont stockés dans des variables du programme et traités par une fonction définie par le programmeur. Cette fonction est appelée à chaque tuple. L'adresse de la fonction est passée en paramètre d'appel de la primitive sql.

La fonction **sqli** est une variante de la fonction sql avec un nombre fixe de paramètres, elle permet notamment l'interfaçage avec le langage Pascal, où un nombre variable de paramètres n'est pas possible. La fonction **sqlo** est aussi une variante de la fonction sql pour les requêtes ne contenant pas de paramètres d'entrée, ni de retour.

Une structure générale à l'ensemble des fonctions, appelée **sqlstatus**, rend des informations sur le déroulement de l'appel. Elle contient des informations de contrôle telles que :

- le type de requête SQL exécutée (select, update, delete, ...),
- le nombre de lignes insérées, modifiées ou supprimées dans une table,
- le nombre d'attributs sélectionnés dans une table,
- le type et la longueur de chaque attribut sélectionné,
- les codes des erreurs détectées par l'interface Opidum et par Oracle.

En local, l'interface Opidum est structurée en deux modules :

- le module **opidumagcl_h.c**, qui décrit les fonctions de "plus haut niveau" telles que sqls, sql, sqli et sqlo qui sont composées de la phase de substitution dans la requête des paramètres d'entrée, d'un appel à la fonction sqlb, et d'appels à la fonction sql_fetch, chaque appel est suivi de l'affectation des résultats dans des variables du programme ou de l'affichage à l'écran,
- le module **opidum.c**, qui décrit les fonctions "de base" telles que connection, commit, autocommit, sqlb et sql_fetch.

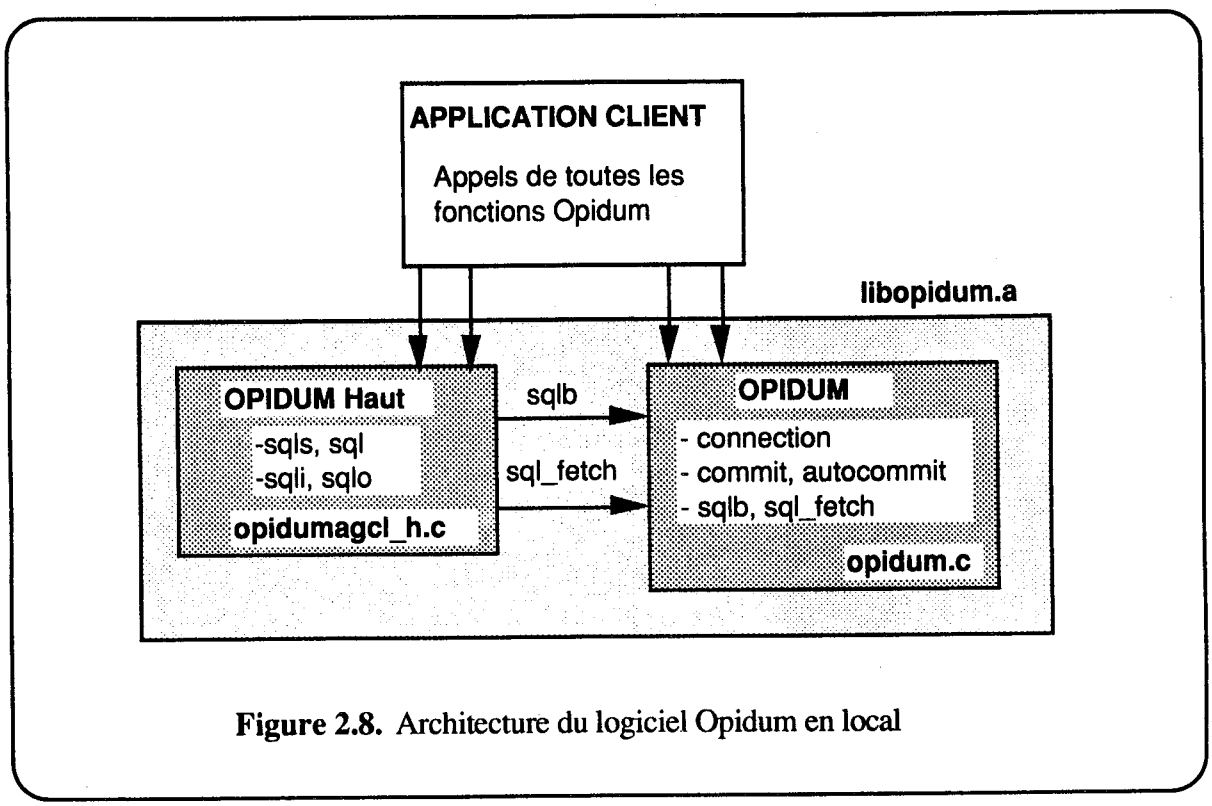



**Figure 2.8.** Architecture du logiciel Opidum en local

Le code objet du programme d'application doit être lié à la librairie objet *libopidum.a*, pour constituer le code exécutable :

**cc -o** application application.o **libopidum.a**

### 3.3.2. En réparti

Pour des raisons implicites de programmation, nous avons implanté le module opidumagcl_h.c sur le site client. En effet, dans le cas où la récupération des tuples est automatique, l'affichage des résultats à l'écran ou leur traitement par une fonction définie dans le programme appelant sont des opérations qui doivent être impérativement exécutées localement. Les fonctions appelables à distance sont donc celles définies dans le module opidum.c. L'appel à distance aux

fonctions Opidum doit avoir la même syntaxe que l'appel en local, pour cela nous avons introduit des interfaces appelées **agent client** et **agent serveur**, qui se chargent des transformations imposées par le logiciel RPC/Sun.

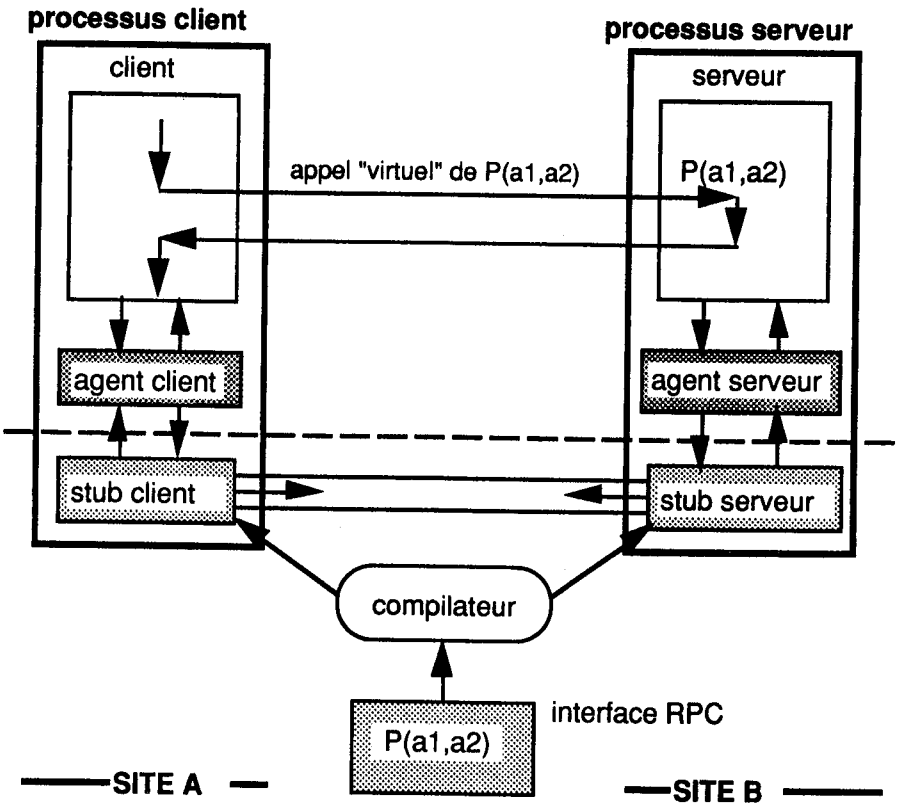



**Figure 2.9.** Introduction des agents client et serveur

L'agent client est composé de la description de toutes les procédures susceptibles d'être appelées à distance avec la même syntaxe que l'appel en local, c'est à dire, même identifiant et mêmes paramètres. Chaque procédure de l'agent client appelle une procédure définie dans l'agent serveur, mais préalablement elle intègre les paramètres d'appel dans la structure de communication passée en appel. Inversement, au retour de l'appel, elle affecte les champs de la structure de communication en retour dans les paramètres résultats de la procédure.

L'agent serveur définit autant de procédures que l'agent client. Chaque procédure de l'agent serveur reconstitue l'appel de la fonction tel qu'il a été formulé dans le programme d'application.

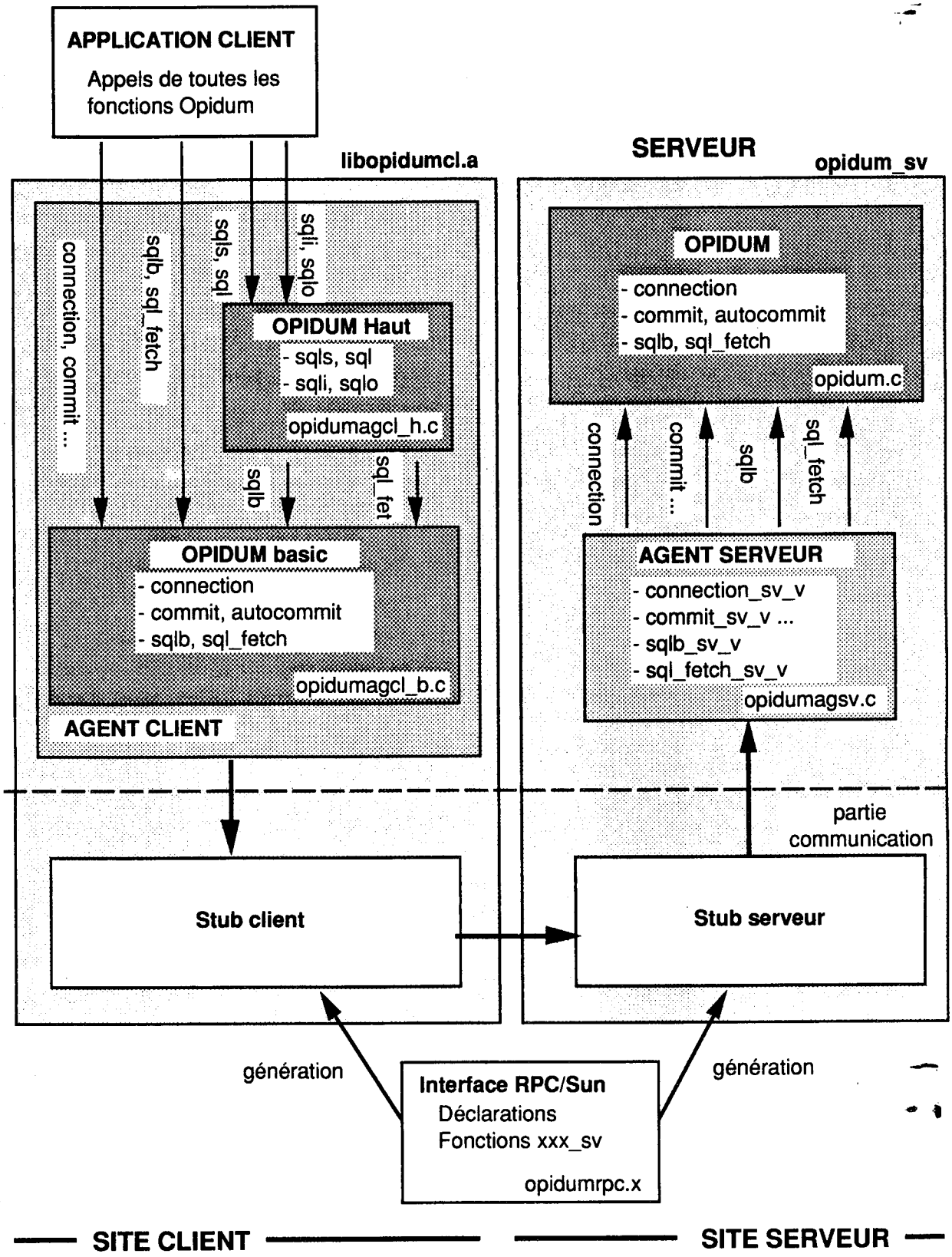



**Figure 2.10.** Architecture du serveur Opidum en réparti

Supposons qu'un programme d'application veuille se connecter à un SGBD Oracle localisé sur la machine "toutatis", sous le nom d'utilisateur "test" avec le mot de passe "test". L'appel depuis le programme client est le suivant :

```
if (connection (1, "test@toutatis", "test") == 0)
 {
        printf ("connexion impossible\n");
        /* impression des codes erreurs retournés */
        exit (1);
}
```

L'interface RPC/Sun, qui décrit à la fois les structures de communication et les procédures appelables à distances est à définir . Pour l'exemple de la fonction *connection* , l'interface **opidumrpc.x** est de la forme :

```
…
struct s_in_connect {
        int on;
        string user<30>;
        string pwd<30>;
}
struct s_out_connect {
        int booleen;
        unsigned short errnum;
        short int_err;
};
…
program OPIPROG {
 version OPIVERS {
        s_out_connect CONNECTION_SV (s_in_connect) = 1;
        …
 } = v;
} = p;
```

Les agents vont se servir du module des déclarations communes, appelé opidumrpc.h, créé par le compilateur RPCGEN.

L'agent client réalise les tâches suivantes dans le cas de la fonction *connection* :

- dans le cas d'une connexion à Oracle, l'établissement de la communication entre le programme d'application et un serveur Opidum, selon le mode un serveur par client (se reporter au paragraphe 3.2),
- dans le cas d'une déconnexion d'Oracle, la libération du serveur Opidum rattaché,
- l'affectation des paramètres d'appel dans les champs de la structure de communication, appelée **s_in_connect**,
- l'appel à la fonction **connection_sv_v**, définie chez l'agent serveur,
- l'affectation des paramètres de retour, stockés dans la structure de communication **s_out_connect**, dans les champs de la structure globale sqlstatus.

L'agent client **opidumagcl_b.c** est donc de la forme :

```
Inclusion des déclarations communes : opidumrcp.h
...
procédure connection (on/off, user, pwd)
début
  si (on)
  alors
    Récupération du nom de la machine hôte
    Demande de connexion avec le serveur général :
    Lien_daemon = clnt_create (hôte, DAEMON_PROG, DAEMON_VERS, "tcp")
    si (Connexion établie avec le serveur général)
    alors
      Appel à la procédure distante n°serveur_V :
      i = n°serveur_V (NULL, lien_daemon)
      si (Serveur Opidum (P,V+i) activé)
      alors
        Demande de connexion avec le serveur Opidum (P,V+i) :
        Lien_opidum = clnt_create (hôte, OPIPROG, OPIVERS + i,"tcp")
        si (Connexion établie avec le serveur Opidum)
        alors
          Affectation des paramètres d'appel dans la strucure de
          communication s_in_connect :
             s_in_connect.on <- on
             s_in_connect.user <- user
             s_in_connect.pwd <- pwd
        finsi
      finsi
    finsi
  sinon
    Affectation des paramètres d'appel dans la structure de
    communication s_in_connect :
       s_in_connect.on <- off
       s_in_connect.user <- user
       s_in_connect.pwd <- pwd
  finsi

  Appel à la fonction connection_sv_v :
     s_out_connect = connection_sv_v(s_in_connect,lien_opidum)
  Affectation des paramètres de retour de s_out_connect dans la
  structure globale sqlstatus :
     sqlstatus.errOrac <- s_out_connect.errOrac
     sqlstatus.errOpid <- s_out_connect.errOpid
  si (off)
  alors
    Appel à la procédure distante kill_n°serveur_V :
       Message = kill_serveur_V (i, lien_opidum)
  finsi

fin
```

L'agent serveur (**opidumagsv.c**), dans lequel sont définies toutes les fonctions suffixées par "_v" (v étant le numéro de version), effectue l'appel à la fonction connection (avec la même syntaxe que dans le programme client), et l'affectation du résultat de l'appel et de la structure globale sqlstatus dans les champs de la structure de communication appelée **s_out_connect.**

```
procédure connection_sv_v (s_in_connect)
début
  si (s_in_connect.on)
  alors
    Appel à la fonction connection pour se connecter à Oracle
    connection (on, s_in_connect.user, s_in_connect.pwd)
  sinon
    Appel à la fonction connection pour se déconnecter d'Oracle
    connection (off)
  finsi
  Affectation de sqlstatus dans la structure s_out_connect
     s_out_connect.errOrac <- sqlstatus.errOrac
     s_out_connect.errOpid <- sqlstatus.int_Opid
  Return (s_out_connect)
fin
```

Cette démarche, nécessaire pour rendre invisible au programme d'application les étapes de communication entre les deux sites distants, est à appliquer pour toutes les fonctions définies dans le module opidum.c.

Sur le site client, le code exécutable de l'application est construit de la même manière qu'en local par la commande :

**cc -o** application application.o **libopidumcl.o**

## 4. Conclusion

Un des avantages importants du logiciel Opidum concerne la transparence donnée à l'application au niveau de la localisation de la base de données. Le programme d'application accède à la base de données par des appels aux primitives Opidum, de la même manière qu'en local (la syntaxe est la même); il doit néanmoins connaître le nom de la machine hôte du SGBD.

L'écriture du logiciel Opidum a été simplifiée grâce à l'emploi du service d'appel de procédure à distante, RPC de Sun. Toutefois, l'introduction d'agents client et serveur est indispensable pour offrir l'abstraction totale de la répartition, quels que soient les outils ou les applications distribués réalisés. Le développement de l'interface Opidum a permis la mise au point d'un mécanisme général de construction d'applications et d'outils distribués avec le logiciel,

RPC/Sun. Ce mécanisme a d'ailleurs été repris lors la réalisation du serveur d'information bibliographique, présenté dans le chapitre suivant.

# 1. Introduction

Nous abordons dans ce chapitre la méthodologie de conception et de réalisation d'un serveur d'Information Bibliographique. Son développement a été motivé par la volonté de remplacer les outils existants par un système mieux adapté aux besoins hétérogènes des utilisateurs, et plus performant. Les techniques utilisées pour sa mise en oeuvre sont adaptées à l'environnement : la gestion des données est faite en utilisant le Système de Gestion de Bases de Données relationnel Oracle, le serveur utilise l'interface procédurale Opidum pour accéder au SGBD, la distribution du serveur est réalisée via le protocole d'appel de procédure à distance, RPC/Sun. Le serveur BIB offre des fonctionnalités qui constituent un support au développement d'applications de gestion bibliographique : il permet d'archiver des notices bibliographiques dans le SGBD sous-jacent et de les extraire en fonction de critères de recherche divers (mot-clé, auteur, contenu, ...).

En première partie, nous présentons l'étude générale sur les systèmes bibliographiques, qui nous a influencés dans la définition des fonctionnalités de l'outil. La seconde partie est consacrée à la présentation des spécifications techniques du système proposé.

# 2. Définition d'un système de gestion bibliographique

Le développement d'un système d'information débute par la phase d'analyse de l'information, dont l'objectif est de déterminer les informations qui doivent figurer dans le futur système, le sens exact reconnu par les différents utilisateurs de chacune de ces informations, et la forme des échanges mis en jeu entre l'utilisateur et le système. Le concepteur doit, dans un premier temps, procéder à l'étude de l'existant, puis recueillir les souhaits des utilisateurs potentiels du système et leur proposer des orientations opérationnelles.

## 2.1. La fonction de gestion bibliographique

Le rôle essentiel d'un système de gestion bibliographique est d'informer au fil des jours, le spécialiste de ce qui est publié dans son domaine et de lui permettre de retrouver tous les documents utiles à la résolution d'un problème particulier. La base de données inhérente au système est telle que, à la suite d'une question exprimée à l'aide de mots-clés, nom d'auteur, date d'édition ..., le système répond en indiquant les références (titre de l'article, revue dans laquelle il a été publié, date, pages et éventuellement un résumé, ...) du (des) document(s) dans le(s)quel(s) la question posée est traitée.

La fonction de gestion bibliographique s'attache à l'aspect extérieur des documents, leur localisation et permet éventuellement l'édition de certains produits : catalogue des ouvrages, liste des périodiques, dernières acquisitions. Les bases de données bibliographiques s'opposent ainsi aux bases de données factuelles qui donnent directement le renseignement recherché, telles que par exemple, certaines bases de données juridiques qui donnent accès au texte intégral. Par définition, les bases de références bibliographiques sont textuelles. Les bases de données factuelles peuvent comprendre du texte et des données numériques.

La norme **Afnor** de la documentation [Afnor 87] a servi de référence pour l'étude : nous donnons les principales définitions relatives à la fonction.

Une bibliographie est une liste de documents **secondaires**, appelés **notices bibliographiques** classées selon certains critères, où sont recensés et décrits des documents **primaires** dans lesquels se trouve le texte intégral, pour en permettre l'identification.

Note 1 : une bibliographie peut se présenter soit sous la forme d'un document autonome ("répertoire bibliographique"), soit sous la forme d'une annexe à un document ou à une partie de document primaire (elle est alors dite "bibliographie cachée").

Note 2 : une bibliographie peut parfois indiquer la localisation.

Une bibliographie **signalétique** est une bibliographie dont chaque notice est suivi d'un résumé.

La notice bibliographique est l'ensemble des éléments présentant la **description bibliographique** et la **vedette** d'un document pour le classement dans un catalogue ou une bibliographie.

La description bibliographique est l'ensemble des données bibliographiques relatives à un document généralement prises dans celui-ci et servant à son identification : titre, mention de responsabilité, mention d'édition, adresse, **collation**, **collection**, notes, numéro international normalisé, reliure, prix et **dépouillement**.

Note : Dans certains cas, cette description peut être simplifiée et ne pas contenir tous ces éléments.

La collation est l'ensemble des éléments matériels décrivant un document.

Une collection est une publication en série comprenant un ensemble de volumes ayant chacun son titre propre, réunis sous un titre commun et paraissant pendant une durée non limitée à l'avance.

Dans le cas d'une publication en plusieurs volumes, le dépouillement sont les éléments bibliographiques propres à chacun des volumes.

La vedette d'un document est un mot ou un groupe de mots, symboles, généralement en tête de la notice bibliographique, et servant au classement et à la recherche dans un catalogue, une bibliographie ou un index.

## 2.2. Analyse d'outils existants

Le système Unix propose un ensemble d'outils qui permettent d'alléger la manipulation des documents. Parmi ces outils, l'utilitaire **nroff/troff**, utilisé conjointement avec un éditeur de texte, réalise la mise en page de documents. Les autres outils, que l'on peut combiner avec le formateur de texte, permettent la création de tableaux, de figures ou la recherche d'informations dans une bibliographie. Des outils de gestion bibliographique, indépendants du système Unix, ont aussi été étudiés.

### 2.2.1. Des outils sous Unix

#### 1) REFER

REFER est un préprocesseur du formateur de texte nroff/troff, qui permet de rechercher et de mettre en forme des notices bibliographiques [Tuthill 88]. REFER nécessite deux entrées : un document à formater et une base de notices bibliographiques.

Les utilisateurs doivent construire leur propre fichier de notices : une notice est constituée d'un ensemble de lignes, chacune contenant un élément bibliographique. Chaque ligne qui contient un élément, commence par un "%" suivi d'une lettre-clé symbolisant le nom de l'élément (exemple : %A pour auteur).

La commande **refer** [ *options* ] [ *fichier* ] a pour effet : dans le document, les marques .[ *mots-clés* .] sont remplacées par les notices bibliographiques, extraites de la base.

D'autres commandes sont disponibles pour faciliter l'utilisation de REFER :

- **pubindex** [ *fichier* ] ... : création d'un index dans les fichiers de notices bibliographiques, qui peut être utilisé par les commandes lookbib et refer.
- **lookbib** [ *fichier* ] ... : recherche de notices bibliographiques par mot-clé et affichage sur la sortie standard.

### 2) BIB

BIB est un programme qui réalise la collecte et le mise en forme de listes de notices dans des documents [Budd 88]. C'est une variante du programme REFER. BIB nécessite aussi deux entrées : un document à formater et une base de notices bibliographiques.

La commande **bib** **-t***style* [*fichier* ] a deux effets :

- dans le texte du document, les citations imprécises [. *mots-clés* .] sont remplacées par les citations plus conventionnelles, faisant référence aux notices sélectionnées, dans le *style* défini,
- la liste entière des notices bibliographiques est insérée, suivant l'appel à la macro .[] .

Des commandes permettent la mise à jour et l'interrogation de la base de notices bibliographiques :

- **addbib** crée une base incrément.
- **catbib** concatène base.incrément avec base.
- **majbib** permet la mise à jour de la base sous éditeur.
- **invbib** provoque l'indexation (création d'index inversé).
- **lookbib** permet la recherche de notices bibliographiques par mot-clé ou auteur.

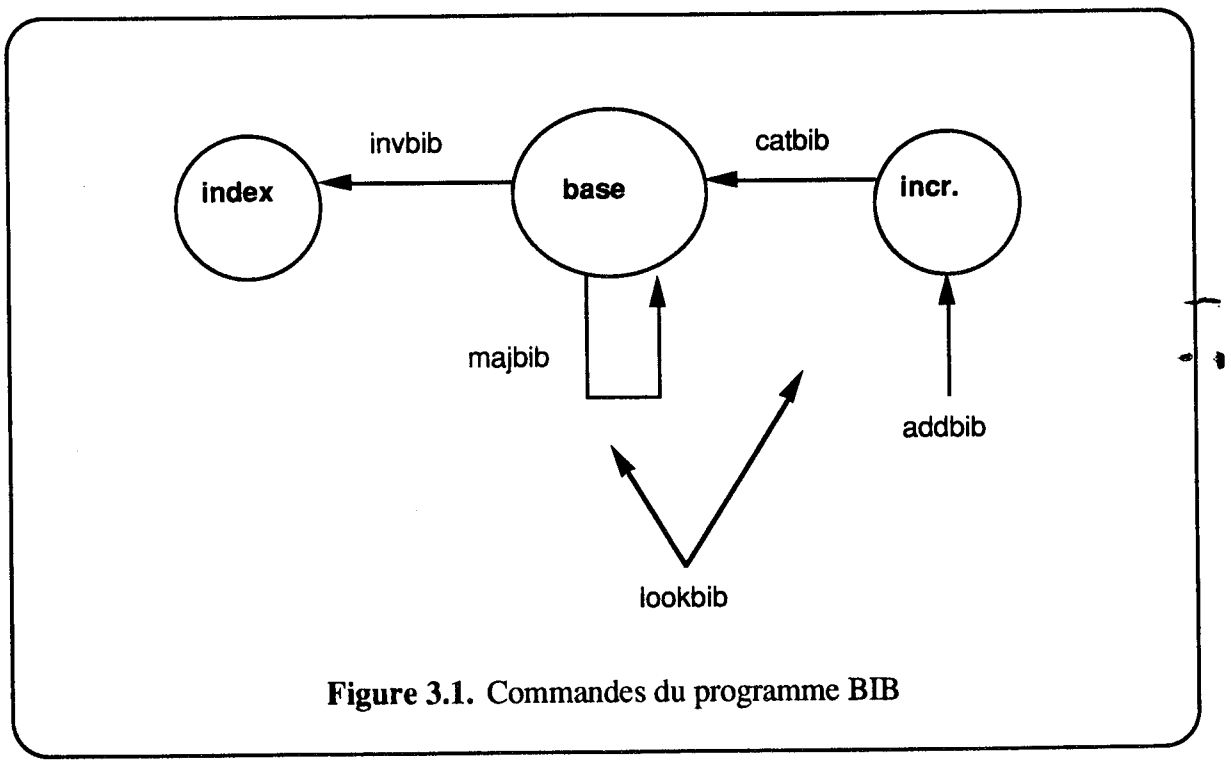



**Figure 3.1.** Commandes du programme BIB

L'utilisation de BIB a été adaptée aux normes LATEX, concernant les types de documents et leur description bibliographique associée.

**3) BIBTEX**

BIBTEX fait partie du système LATEX, système complet pour la préparation de documents. BIBTEX a des fonctionnalités similaires à BIB.

Ces outils sont peu utilisés par les membres du centre de recherche parce qu'ils sont peu conviviaux et mal adaptés aux besoins : ils offrent une interactivité très limitée, et les types de documents proposés et les notices associées ne correspondent pas ou mal aux nécessités.

De plus, tous ces systèmes utilisent les index inversés dont nous décrivons le mécanisme :

L'opération de construction de l'index est subdivisée en deux phases [Lesk 88]:

- création des mots-clés : toutes chaînes alpha-numériques du fichier d'entrée qui ne sont pas parmi les mots les plus fréquents et qui sont non numériques (excepté les chaînes de 1900 à 1999). Les clés sont chargées en minuscules et tronquées sur six caractères. Toute ponctuation est supprimée.

- hachage et tri : exécution du programme de hachage et écriture des fichiers inverses.

Cette opération est supposée peu fréquente.

L'opération de recherche, qui restitue une notice bibliographique, en réponse à la requête fournie par l'utilisateur, est composée de deux phases :

- recherche par rapport aux mots-clés fournis : parcours des fichiers inverses et détermination des adresses correspondantes.

- restitution : par rapport aux adresses, recherche des notices originales.

Les systèmes à index inversé présente des inconvénients importants. Toute mise à jour, suppression ou ajout d'une notice bibliographique, nécessite la modification des fichiers inverses, qui ne peut se faire sans nouvelle création de l'index. Ces systèmes sont donc plutôt adaptés à des données stables. D'autre part, ces outils recherchent par rapport à l'ensemble des mots-clés par conséquent, il est impossible de rechercher par rapport à des expressions arithmétiques ou logiques (exemple : "date >= 1970").

### 2.2.2. Le produit SuperDoc

SuperDoc, produit par la société Aidel, est un progiciel de gestion de bases de données textuelles pour micro-ordinateur fonctionnant sous le système MS-DOS [Aidel 88].

SuperDoc est construit sur un système de fichiers de type indexés directs.

L'utilisateur regroupe l'ensemble des informations qu'il veut gérer, dans une base. Chaque information à mémoriser est intégrée dans une unité structurée d'enregistrement, appelée **notice**. Une notice est constituée de **rubriques** qui contiennent des informations variées et qui décrivent la notice. Les principales fonctions, mises à la disposition de l'utilisateur sont :

- la création d'une base : l'utilisateur définit le format de la notice, avec la possibilité de désigner un nombre illimité de rubriques index,
- la mise à jour du contenu d'une base : ajout, modification et suppression de notices,
- l'interrogation d'une base par mots-clés, contenus dans le lexique pré-défini par l'utilisateur, sur une ou plusieurs rubriques index,
- l'interrogation à partir d'un thesaurus, par rapport à l'ensemble des rubriques index,
- la recherche de chaîne de caractères dans une seule rubrique à la fois (la recherche "plein texte" entraîne un parcours séquentiel des fichiers),
- la modification de la structure d'une base (ajout de rubriques),
- l'édition de notices sélectionnées.

L'utilisateur est guidé par un enchaînement de menus.

L'utilité de SuperDoc, en tant que système bibliographique, est limitée parce qu'il empêche de gérer des notices bibliographiques par type de document : le format de la notice est unique pour une base.

## 2.3. Spécifications générales du système proposé

La définition du système amène à faire des choix relatifs aux informations à prendre en compte dans le SGBD sous-jacent, et aux fonctions que le système doit assurer.

### 2.3.1. Les données

Bien que l'accord ne soit pas parfait, nous sommes parvenus à déterminer les types de documents à inclure dans la base et le format des notices associé. Une **notice** est constituée d'un ensemble de **rubriques**, chacune désignant un élément bibliographique, tels que auteurs, titre, etc. Nous détaillons la description bibliographique de chaque type de document.

Remarque : [Rubrique] signifie que la rubrique est facultative.

**actes** (proceedings)

- Titre du congrès
- Date du congrès
- [Nom(s) de(s) éditeur(s) scientifique(s) (editors)]
- [Nom de l'éditeur commercial (publisher)]
- [Lieu du congrès]

**article** (article)

- Nom(s) et prénom(s) (ou initiales) de(s) auteur(s)
- [Leur(s) adresse(s) professionnelle(s)]
- Titre de l'article
- [Description bibliographique du document d'appartenance]
- [Numérotations : séries, année, volume (tome), numéro de fascicule ou mois, première et dernière page]

**compte-rendu** (minutes)

- Nom(s) et prénom(s) (ou initiales) de(s) auteur(s)
- Titre
- Date
- [Nom du diffuseur du document]

**journal** (newspaper)

- Titre
- Date de parution
- [Numérotations : numéro ou mois ...]

**livre** (book)

- Nom(s) et prénom(s) (ou initiales) de(s) auteur(s) ou collectivité-auteur, ou éditeur scientifique
- Titre
- [Nom de l'éditeur commercial (publisher) ou nom de l'imprimeur]
- [Lieu de publication ou lieu d'impression]
- [Année de publication]
- [Collection (titre de la collection dans laquelle le livre est publié)]
- [Numérotations : séries, volume ...]

**manuel** (textbook)

- Nom(s) et prénom(s) (ou initiales) de(s) auteur(s) ou collectivité auteur
- Titre
- [Nom de l'éditeur scientifique]
- [Adresse de l'éditeur scientifique]
- [Nom de l'éditeur commercial ou nom de l'imprimeur]
- Date de publication ou d'impression

**mémoire** (memoir)

- Nom(s) et prénom(s) de(s) auteur(s)
- Titre
- [Lieu d'édition]
- [Nom de l'éditeur commercial ou de l'imprimeur]
- Année de soutenance
- Nature du mémoire
- [Nom de l'université d'origine]

**rapport** (report)

- Nom(s) et prénom(s) de(s) auteur(s) (ou initiales)
- Titre
- [Nature du rapport : rapport technique ...]
- [Adresse de l'organisme-éditeur]
- [Nom de l'organisme-éditeur]
- Date
- [Numérotation du rapport]
- [Centre de diffusion du document]

**revue** (periodical)

- Titre
- Date de publication
- [Nom de l'éditeur commercial]
- [Numérotations : numéro ou mois ...]

Les rubriques suivantes :

- [Etat : non publié, à paraître, ébauche (draft) ...]
- [Notes]
- [Mots-clés]
- [Keywords]

- Adresse physique : rangement
- [Nom et prénom de l'emprunteur]
- [Date d'emprunt]
- [Résumé : contenu, tables des matières]
- [Abstract]

sont communes à tous les types de documents.

Cette présentation des informations bibliographiques des documents ne fait pas l'unanimité, l'outil ne doit donc pas figer les types de document et leur description : un groupe d'utilisateur doit pouvoir personnaliser sa base bibliographique, en définissant ses propres descriptions bibliographiques des documents.

### 2.3.2. Les traitements

Trois grandes classes de traitement sont envisagées :

#### a) L'administration de la base

Chaque bibliographie relative à un thème donné, ou liée à un projet ou une équipe donné, est mémorisée dans une base, identifiée par le titre du thème, le nom du projet ou de l'équipe. Préalablement à la création de la base, les types de document à stocker avec leur description bibliographique sont définis par les utilisateurs concernés. Tout au long de la vie de l'application, l'administrateur de la base désigné par les utilisateurs, est chargé de maintenir la structure de la base : ajout de types de document, mise à jour du format des notices (ajout, suppression et modification de rubriques).

Toutefois, des descriptions bibliographiques standards (celles décrites dans le paragraphe 2.3.1) sont proposées aux utilisateurs, elles sont mémorisées dans une base sous le nom d'usager "Super Administrateur", qui sert de modèle.

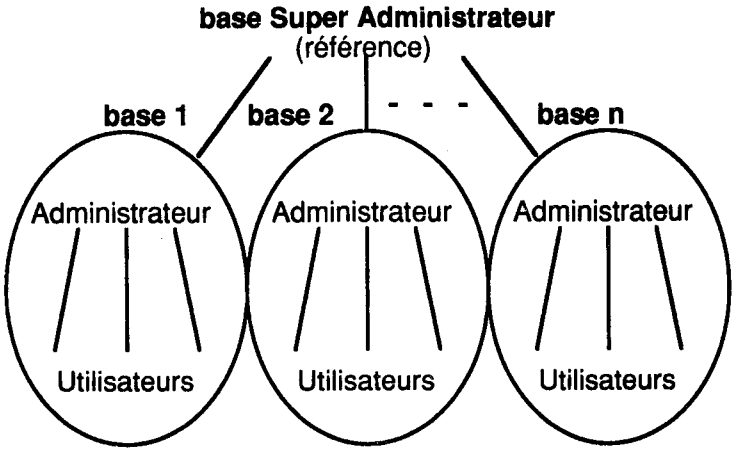



**Figure 3.2.** Organisation de l'application

### b) La mise à jour de la bibliographie

Pour l'insertion d'une notice bibliographique, l'utilisateur choisit le type de document, puis le menu approprié (c'est à dire les rubriques à renseigner) s'affiche à l'écran. Ainsi l'utilisateur est guidé pour la saisie des informations bibliographiques du document.

L'image d'une notice bibliographique existante est appelable pour l'insertion d'une notice presque similaire.

La mise à jour et la suppression de notices sont des traitements indispensables.

### c) La consultation de la bibliographie

Le système permet d'extraire des notices bibliographiques, en fonction de plusieurs critères de recherche. Il met à la disposition de l'usager, un langage qui permet de formuler des requêtes composées d'expressions arithmétiques et logiques, combinant plusieurs rubriques. Toutes les rubriques sont interrogeables, y compris les rubriques textuelles qui décrivent le contenu des documents.

D'autres traitements plus spécifiques sont prévus :

- un traitement de reprise des données existantes est nécessaire pour effectuer le chargement initial de la base. Notamment, un processus de conversion du format utilisé par l'outil BIB aux formats personnalisés est à envisager.

- le couplage avec un formateur de texte (nroff/troff, LATEX,...) pour permettre de référencer et éditer des notices dans un document.

# 3. Spécifications techniques du système

L'étude technique du système succède à la phase de définition. L'objectif de cette étape est de définir à la fois l'architecture logicielle du système, l'organisation interne des données à stocker, et les spécifications détaillées des traitements.

## 3.1. Architecture logicielle

Dans un premier temps, nous présentons le principe général d'organisation du système proposé, qui offre les deux aspects de distribution suivants : l'accès à distance et la distribution du serveur. Ensuite, nous justifions le choix du schéma de fonctionnement du serveur d'Information Bibliographique.

### 3.1.1. Principe général

Le serveur d'information bibliographique est vu comme un **outil**, au même titre qu'Oracle.

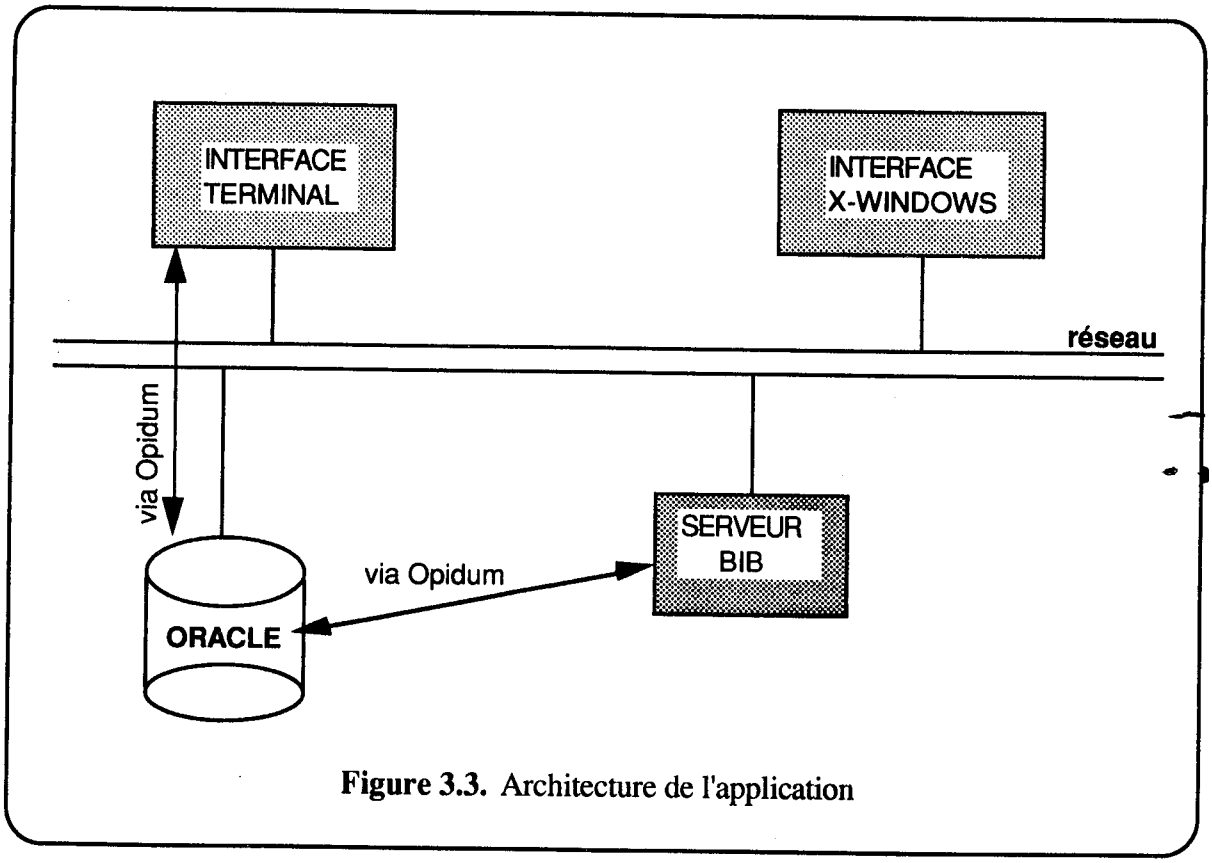



**Figure 3.3.** Architecture de l'application

Le serveur BIB exporte les fonctions de base pour l'administration, la mise à jour et la consultation de la base bibliographique, qui servent de support au développement d'interfaces.

L'interface offre à l'utilisateur une façon agréable d'introduire ou d'extraire les notices, elle est indépendante du type des terminaux, elle utilise pour ce faire les librairies Unix standards de gestion de terminaux. Une interface graphique de type X-WINDOWS peut être envisagée.

Le flux des données entre l'application et le serveur suit les règles imposées par l'utilisation du le logiciel RPC/Sun, et est indépendant du type d'interface mise en oeuvre.

Le but est de réaliser la distribution totale de l'application : par exemple, le SGBD Oracle et le logiciel Opidum sont implantés sur la machine la plus puissante du réseau (SPS9), le serveur BIB est présent sur une ou plusieurs machines, et les applications clientes sont dispersées sur le réseau.

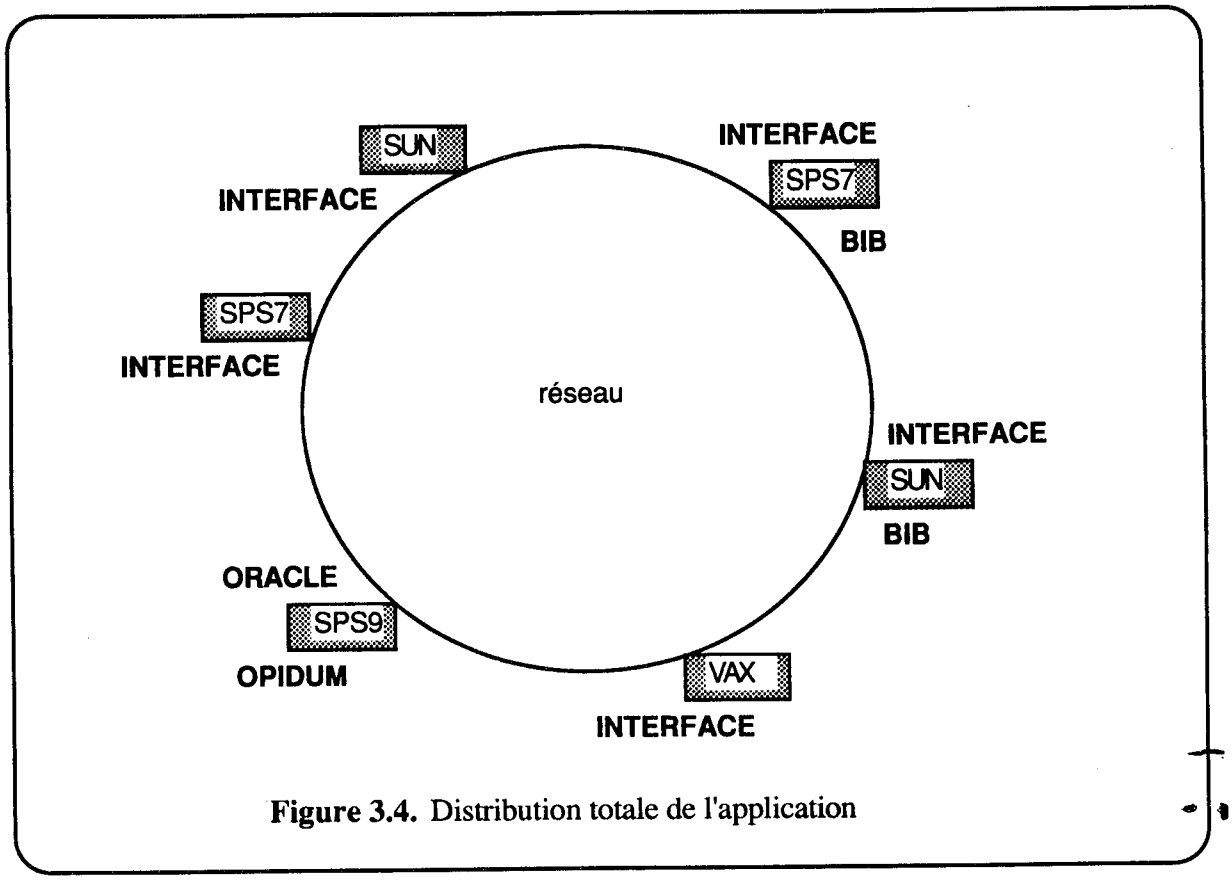



**Figure 3.4.** Distribution totale de l'application

### 3.1.2. Choix du schéma de fonctionnement

Plusieurs fonctionnements en mode client/serveur sont envisageables :

**a) un serveur BIB unique pour l'ensemble des clients**

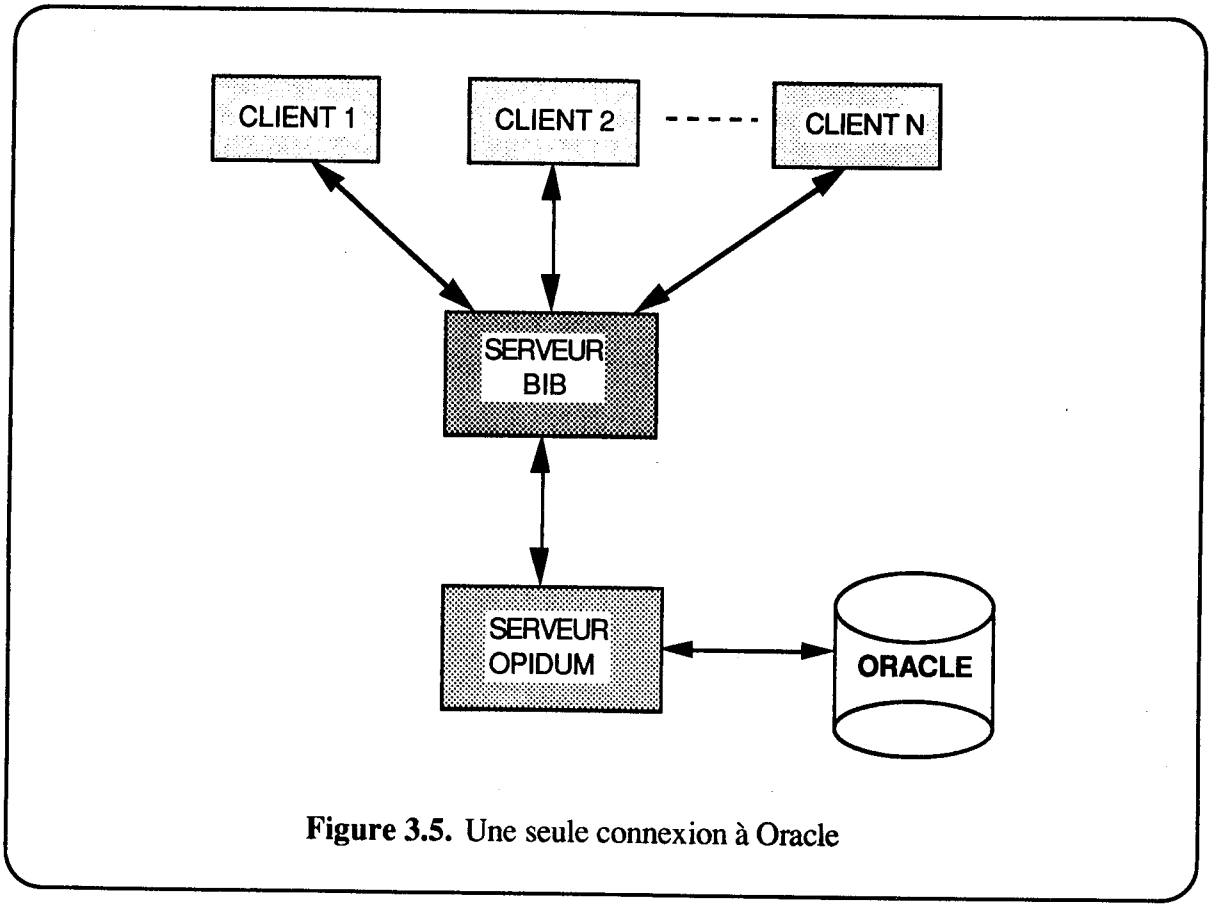



**Figure 3.5.** Une seule connexion à Oracle

Un serveur BIB unique pour une machine hôte, lancé en arrière plan, centralise les appels des clients : une seule connexion à Oracle est demandée. Au début de son activation, le processus serveur BIB se connecte à Oracle sous le nom d'usager du Super Administrateur, puis il gère lui-même la communication entre le SGBD et l'ensemble des clients, via l'interface procédurale Opidum. L'accès aux bases utilisateurs se fait donc par l'intermédiaire de l'usager Super Administrateur, qui possède les privilèges suffisants sur les tables utilisateurs. Les appels aux fonctions serveur sont paramétrées avec l'identificateur du client et le nom de la base usager ciblée, de sorte que le serveur puisse contrôler les autorisations d'accès aux données. Les ordres finaux parviennent au SGBD de manière séquentielle (pas d'accès concurrents à la base).

Ce type d'architecture complique nettement les fonctionnalités du serveur BIB parce qu'en plus de ses fonctions de base, il doit prendre en charge la communication des données entre Oracle et plusieurs programmes clients et assurer les contrôles d'accès aux données.

**b) un serveur BIB unique et un serveur Opidum par client**

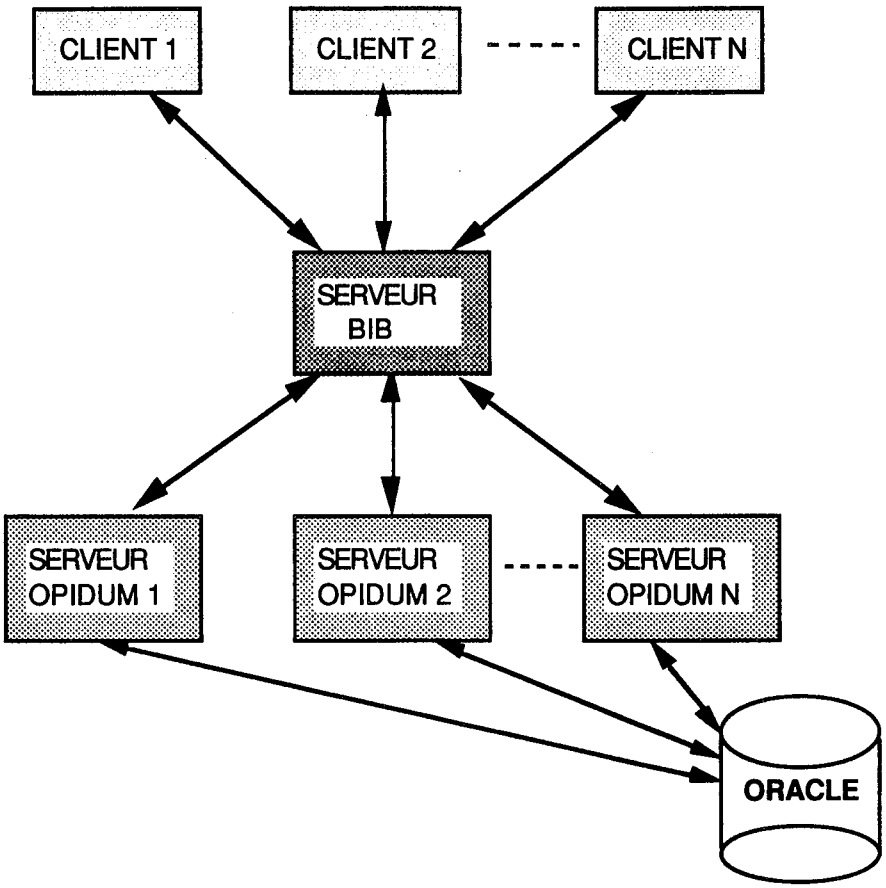



**Figure 3.6.** Un serveur BIB unique et un serveur Opidum par client

Chaque client se connecte à Oracle sous le nom de sa base utilisateur. Contrairement à la première solution, les contrôles d'accès aux données sont assurés par le SGBD. Par contre, le serveur BIB est responsable de la cohérence des échanges entre les clients et l'ensemble des serveurs Opidum, ce qui complique son rôle.

### c) un serveur BIB par client

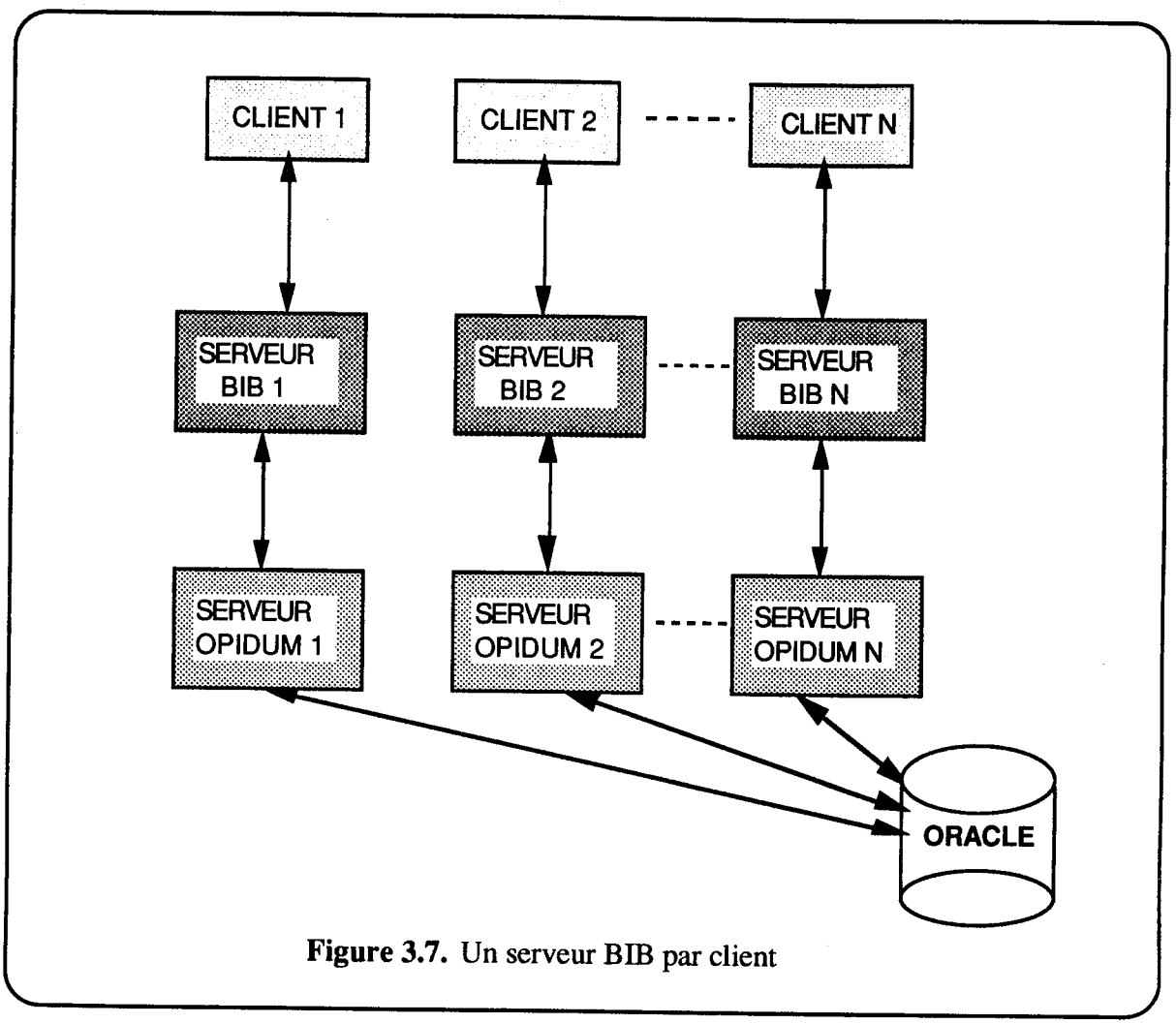



**Figure 3.7.** Un serveur BIB par client

De la même façon que dans la solution précédente, chaque client se connecte à Oracle sous le nom de sa base utilisateur. Un processus Oracle par client est lancé, qui prend en charge la communication des données. Dans le cas où plusieurs clients font des accès parallèles aux données, Oracle se charge de gérer la concurrence. Dans ce cas, le serveur BIB joue le rôle simple d'une interface procédurale. La fonction de gestion bibliographique étant relativement peu utilisée, la duplication du serveur BIB autant de fois que de clients, n'est pas franchement pénalisante.

Le principe d'implémentation de cette solution, est celui utilisé pour le logiciel Opidum : un serveur général centralise les appels des clients, puis active un serveur BIB à chaque appel d'un client.

C'est la solution **un serveur BIB par client**, qui a été retenue, à cause de sa simplicité et de son efficacité.

## 3.2. Le stockage des données

Le modèle des données a une structure hiérarchique à trois niveaux : la base (ex.: BD), relative à un thème, une équipe ou un projet regroupe un ensemble de types de document (ex.: article, livre, ...), à chacun de ces types de document correspond une description bibliographique constituée d'une liste de rubriques (ex.: un article est identifié par un titre, un auteur, une date de parution, ...).

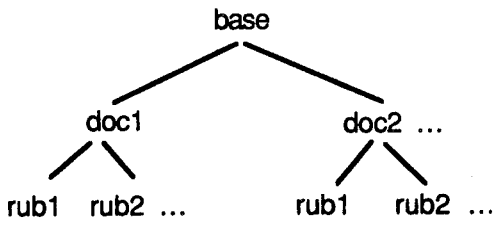



**Figure 3.8.** Structure hiérarchique du modèle des données

### 3.2.1. Les types internes à l'application

A la création de la base, l'administrateur désigné par les utilisateurs, définit pour chaque type de document, le format des notices associé. Pour ce faire, il fournit au système une liste des types de document et une liste de rubriques, pour chaque rubrique, il spécifie son type choisi parmi l'ensemble des types internes à l'application bibliographique et éventuellement sa taille, et aussi sa nature, c'est à dire *standard* ou *utilisateur*. Une rubrique est qualifiée de rubrique standard, si elle sert à décrire plusieurs types de document, sinon elle est dite utilisateur.

L'application prévoit cinq types BIB différents, chacun de ces types correspond à un type de données proposé par Oracle.

| Type BIB | Longueur | Type Oracle |
|---|---|---|
| STRING | N <= 240 | CHAR (N) |
| TEXT | - | LONG |
| INTEGER | - | NUMBER (4) |
| REAL | - | NUMBER (6, 2) |
| DATE | - | DATE |

**Figure 3.9.** Types internes à l'application

Pour chaque type de document, l'administrateur constitue ensuite sa description bibliographique, qui est une combinaison des rubriques déjà définies, pour chaque rubrique il précise son poids, c'est à dire si la rubrique est *obligatoire* ou *optionnelle*, lors de la création d'une nouvelle notice.

### 3.2.2. Les méthodes des signatures

Les rubriques de type TEXT, qui contiennent plutôt des informations relatives au contenu d'un document primaire, telles que le résumé ou la table des matières, posent certains problèmes de gestion avec Oracle. En effet, le type TEXT est stocké sous la forme d'un LONG (65535 caractères maximum), au sens Oracle. Or le type LONG est soumis à des contraintes importantes : une seule colonne de type LONG est autorisée par table, les colonnes de type LONG ne peuvent pas être indexées, et ne peuvent pas être référencées dans les clauses WHERE, GROUP BY, CONNECT BY et DISTINCT. La restriction la plus gênante dans notre cas, est qu'aucune opération de recherche sur la base ne peut être effectuée par rapport aux valeurs d'une rubrique de type TEXT. Le système doit donc apporter une solution à la recherche par rapport aux rubriques de type TEXT, en supplément des possibilités offertes par le SGBD Oracle.

Dans le cadre des environnements bureautiques, trois approches fondamentales pour les méthodes d'accès associées aux documents, et en particulier à leur contenu textuel sont proposées :

- **le balayage du texte** : c'est la méthode la plus simple et générale, d'habitude fondée sur la théorie des automates d'états finis. Elle n'a pas besoin d'espace additionnel à celui nécessaire pour le stockage du texte. Par contre, le temps requis pour les opérations de recherche est proportionnel à la taille du texte, et donc prohibitif dans le cas de bases textuelles de grosse ou moyenne taille. Les machines spécialisées en filtrage de texte sont souvent utilisées pour améliorer le temps d'évaluation.

- **les fichiers inverses** : c'est la méthode la plus traditionnelle dans les systèmes commerciaux. C'est la plus performante lors de l'interrogation par mots-clés, mais elle présente trois grands inconvénients : l'index de mots-clés peut occuper jusqu'à 300% de la taille de la base, le vocabulaire indexé est contrôlé, ou du moins très couteux à mettre à jour, et la recherche par parties de mots n'est pas toujours possible (avec des performances acceptables).

- **les fichiers signatures** : cette méthode est fondée sur la construction, pour chaque partie textuelle, d'une image compressée de son contenu. L'évaluation de la requête est faite sur cette image, ce qui permet une sélection des documents plus rapide et selon l'algorithme de construction de la signature, peut aussi faciliter l'évaluation d'expressions logiques. Beaucoup d'algorithmes de construction de signatures sont proposés dans la littérature, les plus connus étant les Signatures par Mots (WS) et par Codage superposé (CS). Un autre intérêt de ces méthodes est le peu de place nécessaire pour stocker les signatures et les gains en temps de réponse. Il est à noter que ce temps reste, de toutes façons, moins performant que celui des fichiers inverses.

Dans le cadre des applications bureautiques, les méthodes de signatures sont les plus connues et jusqu'à maintenant les seules qui aient répondu aux exigences. En effet, elles offrent un bon compromis entre la taille de la place mémoire additionnelle pour les supporter et le temps d'accès nécessaire pour l'évaluation de requêtes. Nous allons décrire le principe des méthodes de calcul de signatures : les signatures par mot, puis les signatures par codage superposé, qui sont les plus répandues.

### 1) La méthode de Signatures par Mot (WS)

Dans cette méthode, on extrait les mots significatifs par élimination des mots non significatifs qui sont stockés dans une liste prédéfinie de "mots vides". Chaque mot est transformé en une chaîne de bits, qui peut être, soit de longueur fixe, soit de longueur variable avec un entête. Cette chaîne constitue la *signature d'un mot* . Toutes les chaînes sont concaténées, les unes après les autres (dans l'ordre du document) pour former la *signature du document* .

Exemple :

| **Texte** | **Signature** |
|---|---|
| | (Signatures par mot, de longueur fixe = 5 bits) |
| Processus | 01001 |
| Communicant | 10100 |
| Parallèle | 00101 |
| **Signature du texte** | 01001 10100 00101 |

Afin de permettre la recherche de parties de mots, il faut que la fonction de transformation préserve l'ordre des sous-chaînes de caractères. Dans [Fal 86] est proposée une méthode fondée sur des triplets de caractères consécutifs dans le mot. La signature d'un mot est ainsi générée par la concaténation des signatures des triplets.

Exemple :

Pour une signature de longueur 3 bits par triplet de caractères, la signature du mot "Expert" serait :

| | | | | | | |
|---|---|---|---|---|---|---|
| **Triplets possibles** | : "_Ex" | "Exp" | "xpe" | "per" | "ert" | "rt_" |
| **Signature des triplets** | : 100 | 001 | 101 | 110 | 010 | 110 |

Signature de "Expert" : 100001101110010110

Une qualité de cette méthode de calcul de la signature d'un texte, est qu'elle préserve les images des mots individuels et l'ordre de leurs composants. Malgré des bons rapports de compression, la taille du fichier des signatures reste proportionnelle à celle du fichier texte original. En fait, les mots qui apparaissent plusieurs fois occasionnent la génération d'autant de signatures répétées.

### 2) La méthode des signatures par Codage Superposé (CS)

Le codage superposé comme méthode de compression est fondé sur l'idée de faire correspondre, à un ensemble de données, une image sur un bloc de bits de taille fixe (donc contrôlée).

Le principe de la méthode est la superposition d'un ensemble de signatures individuelles de mots, dans un seul bloc de bits. Le texte est divisé en blocs d'un nombre fixe $D$ de mots non-vides (une liste des mots non porteurs d'information doit être construite, et ces mots éliminés du bloc en question). A chaque bloc ainsi défini est associée une chaîne de bits de taille fixe $N$, initialisée à zéro. Pour chaque mot dans le bloc de texte, un nombre M de bits dans le bloc est mis à 1, avec $M << N$ . La signature du bloc est formée par le $OU$ logique des signatures individuelles des mots.

Exemple :

Si le texte est "Bases Bibliographiques Intégrées", sa signature CS avec les paramètres D = 3, N = 8, M = 2, est :

| **Texte** | **Signature (CS)** |
|---|---|
| Bases | 00101000 |
| Bibliographiques | 01000010 |
| Intégrées | 10000100 |
| Signature du texte complet | 11101110 |

Pour l'interrogation, la signature de la requête S(Q) est calculée avec le même algorithme de transformation, donc un bloc de N bits initialement à 0, et M bits par mot sont mis à 1. S(Q) est comparée aux signatures des blocs de texte, par de simples opérations logiques sur des chaînes de bits : le bloc du texte est sélectionné si tous les bits "1" dans le bloc signature de la requête

sont aussi mis à "1" dans le bloc signature du texte. Les opérations logiques exprimées dans la requête peuvent aussi être examinées sur les signatures, par des opérations sur les bits.

Exemple :

Dans l'exemple précédent, si la requête porte sur le mot "bibliographique", la signature de la requête est :

**Q** = "Bibliographiques" **S(Q)** = 01000010

et le contenu du texte a comme signature : 11101110
Les positions du bloc de bits à examiner sont la 2ème et la 7ème, car elles sont à "1" dans la signature de la requête. Le texte est sélectionné car les bits correspondants de sa signature sont aussi à "1".

**Q** = "Intelligent AND Expert"

**S** (Intelligent) = 0001001100

**S** (Expert) = 0100010100

**S** (Q) = 0101011100

Signature du document :

**S** (Bloc1) = 0001011110

**S** (Bloc2) = 0101011101

Dans ce cas, le bloc2 est sélectionné, mais pas le bloc1, son deuxième bit étant "0".

Il faut noter que, pour un mot donné dans la requête, sa signature peut former une configuration de bits S(Q) telle que un bloc soit sélectionné même si le mot n'est pas dans le texte. Ceci est appelé une **collision** et elle est due à la méthode de transformation et par le *OU* logique des signatures individuelles.

Les collisions peuvent être, soit tolérées par l'application, soit éliminées dans une étape postérieure de balayage séquentiel des blocs de texte sélectionnés. Avec un bon choix des paramètres N et M, le CS produit une signature très compacte, d'environ 10% de la taille du texte original, avec environ 10% de collisions.

Une bonne caractéristique de CS est que les mots répétés sont automatiquement transformés vers le même ensemble de positions dans le bloc signature, donc leurs signatures ne sont stockées qu'une seule fois au moment de faire le OU logique pour former la signature du bloc.

Pour permettre la recherche de sous-chaînes de caractères (parties de mots), de manière similaire à la méthode WS, les mots sont considérés comme un ensemble de triplets de caractères consécutifs, et les M bits de la signature d'un mot donné sont calculés par la mise à "1" de un bit par triplet. Si un mot contient moins de M triplets, les bits qui manquent peuvent

être ajoutés par des méthodes de hachage. Si le mot contient plus de M triplets, seule les premiers M seraient pris en compte[CF 84].

Exemple :

Pour un bloc de signature de taille N = 20, et M = 6, la signature du mot "Expert" serait :

**Triplets possibles** : "_Ex" "Exp" "xpe" "per" "er" "rt_"

**Signature des triplets** : bit 5 bit 12 bit 8 bit 2 bit 18 bit 15

Signature de "Expert" : 01001001000100100100

La taille du fichier signature, ainsi que la performance globale du système, sont fortement liées au bon choix des paramètres de la méthode. Néanmoins, le processus de filtrage est toujours dépendant (proportionnel en temps) de la taille des documents [CF 84].

Pour effectuer la recherche textuelle sur les notices bibliographiques, c'est la méthode par Codage Superposé (CS) qui a été retenue en raison de sa simplicité de mise en oeuvre :

- les phases de calcul des signatures et d'évaluation des requêtes de sélection sont faciles à implémenter;
- CS offre un très bon rapport de compression : en raison de la superposition des codages, le codage des mots répétés dans un bloc de signature n'est stocké qu'une seule fois;
- de plus, CS a été intégré dans le Serveur d'Information Bureautique réalisé dans le cadre du projet DOEOIS, l'ensemble des fonctions qui ont été codées pour le SIB sont réutilisables.

### 3.2.3. La représentation relationnelle

Le modèle de données proposé dans le modèle relationnel consiste à percevoir l'ensemble des données comme des tableaux organisés en lignes et colonnes. Nous présentons le détail des relations (ou tables) qui regroupent les données relatives à l'application de telle façon qu'elles puissent être exploitées de manière efficace par les programmes de traitement (nous avons appliqué la décomposition des relations en respectant les formes normales).

| NOM RELATION | NOM ATTRIBUT | TYPE ATTRIBUT | COMMENTAIRE ATTRIBUT | COMMENTAIRE RELATION |
|---|---|---|---|---|
| BASE | nombase | char (30) | son libellé | table des bases consultables |
| DOCUMENT | idfdoc<br>typedoc<br>typedocangl | number (2)<br>char (30)<br>char (30) | idf. type de document<br>son libellé<br>son libellé en anglais | table des types de document |
| RUBRIQUE | idfrub<br>nomrub<br>nomrubangl<br>typebib<br>lngbib<br>typerub | number (3)<br>char (30)<br>char (30)<br>char (7)<br>number (3)<br>char (1) | idf. rubrique<br>son libellé<br>son libellé en anglais<br>type BIB<br>longueur<br>nature (s, u) | table des rubriques |
| DESCRIPTION | idfdoc<br>idfrub<br>poidsrub | number (2)<br>number (3)<br>char (3) | idf. type de document<br>idf. rubrique<br>poids rubrique<br>(obl, opt) | table des descriptions |
| NOTICE | nrnot<br>idfdoc | number (6)<br>number (2) | n° notice<br>idf. type de document | table des notices |
| CONTENU | idfcont<br>contenu | number (6)<br>long | idf. contenu<br>texte original | table des contenus |
| COMPTEUR | cptdoc<br>cptrub<br>cptnot<br>cptcont | number (2)<br>number (3)<br>number (6)<br>number (6) | cpt. type de document<br>cpt. rubrique<br>cpt. notice<br>cpt. contenu | table des compteurs |

**Figure 3.10.** Détail des tables générales de l'application

La table BASE contient les libellés des bases utilisateurs, accessibles en lecture par l'usager courant. Par défaut, l'utilisateur qui crée une table est considéré comme le propriétaire. Il a tous les droits sur cette table et son contenu. En revanche, les autres utilisateurs n'ont aucun droit (ni lecture, ni modification) sur cette table, à moins que le propriétaire leur donne explicitement ces droits.

Les tables DOCUMENT et RUBRIQUE servent à stocker respectivement les types de document et les rubriques nécessaires à leur identification.

La table DESCRIPTION permet de mémoriser les descriptions bibliographiques pour chaque type de document. Une rubrique standard n'est donc définie qu'une seule fois dans la table RUBRIQUE (pour éviter les redondances).

La table NOTICE a pour but de permettre l'optimisation des opérations de recherche sur la base, par le numéro de notice.

La table CONTENU est dédiée au stockage des parties textuelles utilisées dans l'identification des documents.

La table COMPTEUR sert à mémoriser la valeur courante, des identificateurs de type de document, de rubrique, de contenu et du numéro de notice. Chacun de ces attributs voit sa valeur incrémentée de un, à chaque insertion d'un nouveau tuple dans la table. Ainsi, chaque valeur prise est garantie unique.

A partir des lignes des tables DOCUMENT, RUBRIQUE et DESCRIPTION, le système génère autant de tables spécifiques à la base que de types de document créés, chacune de ces tables sert à stocker les notices bibliographiques relatives à un type de document. Chaque table spécifique contient l'ensemble des éléments bibliographiques identifiant le document primaire, y compris le numéro de notice. Pour les données bibliographiques textuelles, elle contient la signature CS du texte, calculée lors de la mise à jour de la notice, et l'index du texte original qui est stocké dans la table CONTENU. Une rubrique de type TEXT figurant dans la description bibliographique d'un type de document, entraîne donc la création de deux colonnes dans la table relative au type de document : une colonne de type binaire (RAW(38)) pour le stockage de la signature CS (se référer au paragraphe 3.2.2), et une colonne numérique contenant l'index du contenu original. Ainsi, plusieurs rubriques de type TEXT peuvent décrire en même temps un type de document. La figure 3.11 illustre cette situation.

**DOCUMENT**

| idfdoc | typedoc | typedocangl |
|---|---|---|
| 1 | livre | book |
| ... | ... | ... |

**RUBRIQUE**

| idfrub | nomrub | nomrubangl | typebib | lngbib | typerub |
|---|---|---|---|---|---|
| 1 | auteurs | authors | string | 80 | s |
| 2 | titre | title | string | 80 | s |
| 3 | éditeur | publisher | string | 40 | s |
| 4 | adresse | address | string | 40 | s |
| 5 | date | date | date | - | s |
| 6 | résumé | abstract | text | - | s |
| ... | ... | ... | ... | ... | ... |

**DESCRIPTION**

| idfdoc | idfrub | poidsrub |
|---|---|---|
| 1 | 1 | obl |
| 1 | 2 | obl |
| 1 | 3 | opt |
| 1 | 4 | opt |
| 1 | 5 | opt |
| 1 | 6 | opt |
| ... | ... | ... |

**NOTICE**

| nrnot | idfdoc |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 1 |
| ... | ... |

**LIVRE**

| nrnot | auteurs# | titre# | éditeur# | adresse# | date# | résumé# | idfres |
|---|---|---|---|---|---|---|---|
| ... | ... | | | | | | |
| 3 | S. Bourne | Unix | - | - | - | 01011... | 4 |
| ... | ... | ... | | | | | |

**CONTENU**

| idfcont | contenu |
|---|---|
| 1 | blabla |
| 2 | blabla |
| 3 | blabla |
| 4 | blabla |
| ... | ... |

**Figure 3.11.** Exemple de configuration de la base

Le système doit assurer que l'information stockée soit cohérente, c'est à dire conforme à la réalité qu'elle représente. Pour ce faire, il doit prendre en compte des contraintes dites **contraintes d'intégrité** (CI). Le SGBD Oracle fournit des mécanismes pour vérifier que la base est cohérente vis-à-vis de certaines contraintes, par exemple, l'unicité de la clé d'une relation est exprimable grâce à la commande CREATE UNIQUE INDEX. Mais d'autres contraintes doivent être garanties par les traitements, par exemple les contraintes que doivent vérifier individuellement un tuple (plage ou liste de valeurs, contraintes entre attributs,...) sont vérifiées lors des opérations de mise à jour des tuples.

Dans notre cas, les contraintes d'intégrité que le système doit prendre en compte sont les suivantes :

- les libellés *nombase*, *typedoc* et *rubrique* sont composés de caractères tous différents du caractère blanc; (le nom d'usager, le nom d'une table et le nom d'une colonne ne doivent pas contenir de caractère blanc);
- la liste des valeurs autorisées pour l'attribut *typebib* est (STRING, TEXT, INTEGER, REAL, DATE);
- la valeur du constituant *lngbib* est requise pour le type TEXT uniquement, elle est comprise entre 1 et 240 inclus;
- l'attribut *typerub* prend les valeurs "s" ou "u";
- l'attribut *poidsrub* prend les valeurs "obl" ou "opt";
- tout identificateur de type de document *idfdoc*, apparaissant dans les relations DESCRIPTION et NOTICE, est décrit dans la relation DOCUMENT;
- tout identificateur de rubrique *idfrub*, apparaissant dans la relation DESCRIPTION, est décrit dans la relation RUBRIQUE;
- la description d'un type de document est constituée de une ou plusieurs rubriques;
- les modifications de tuples dans les relations DOCUMENT, RUBRIQUE et DESCRIPTION sont reportées sur le format des tables relatives à chaque type de document;
- un contenu identifié par *idfcont* est toujours rattaché à une notice, à la suppression d'une notice, le contenu est aussi supprimé dans la table CONTENU;
- une notice supprimée dans la table NOTICE, l'est aussi dans la table associée au type de document.

D'autres tables relatives à l'affichage, ou à l'édition des notices bibliographiques peuvent être créées, leur forme est spécifique à l'interface qui les utilise.

## 3.3. Les fonctions de consultation

Le serveur BIB exporte des fonctions qui servent de support pour la construction d'applications de gestion bibliographique : administration de la base, mise à jour et consultation de la bibliographie. Les procédures mises à la disposition du programmeur sont implémentées de manière à offrir des temps de réponse acceptables au niveau des interfaces, à vérifier la validité des paramètres d'appel qui leur sont transmis par l'application, et à conserver l'état cohérent de la base. Nous détaillons plus particulièrement les fonctions d'interrogation qui nous semblent plus significatives du travail réalisé.

### 3.3.1. Le langage d'interrogation

Nous proposons aux utilisateurs deux formules de consultation :

- **par type de document** : la sélection porte sur un ensemble restreint de notices, qui identifie certains types de document choisis par l'utilisateur. Une requête définit une sélection sur les rubriques relatives aux types de document ciblés.
- **par rubrique** : la recherche porte sur la totalité des notices quel que soit le type de document décrit. Une requête définit alors une sélection sur les rubriques communes à tous les types de documents.

Pour l'implantation de ces formules de recherche, nous avons défini une syntaxe permettant d'exprimer les opérations et les objets sur lesquels portent la requête de sélection. Ceci est réalisé par l'introduction d'opérateurs de relation (=, !=, <,>, LIKE) rattachés aux types internes BIB, et d'opérateurs logiques (AND, OR). Les exemples cités ont trait à la formule de consultation par rubrique.

Exemple :

auteurs = "Delobel" AND date > "01011982"

Dans cet énoncé, nous voulons les notices qui identifient les documents primaires dont l'auteur est "Delobel" et qui ont été publiés après le 1er Janvier 1982.

Dans la définition des règles du langage d'interrogation, nous avons tenu compte du fait que les futurs utilisateurs sont des habitués du système Unix, plutôt que des spécialistes en Bases de Données :

- L'opérateur de relation "=" appliqué sur le type STRING réalise l'opération de recherche d'une chaîne de caractères à l'intérieur de chaînes de caractères (même principe que la commande *grep* du système Unix) :

Exemple :

titre = "système"

Cet énoncé provoque la sélection de toutes les notices associées aux documents dont le titre inclut le mot "système".

- Les expressions de recherche sont évaluées de gauche à droite de la même façon que dans le langage de programmation C.

- L'opérateur AND est prioritaire sur l'opérateur OR (de même que dans le langage C).

- Les parenthèses peuvent être utilisées pour imposer une priorité dans l'évaluation d'une expression de sélection ou simplement pour rendre la requête plus claire :

Exemple :

auteurs = "Delobel" AND titre = "relationnel" OR titre = "système"

Cette requête sélectionne les notices qui identifient à la fois les documents dont l'auteur est "Delobel" et dont le titre contient la chaîne "relationnel", et les documents qui traitent de "système".

(auteurs = "Delobel") AND (titre = "relationnel" OR titre = "système")

Cet énoncé sélectionne les notices relatives aux documents dont l'auteur est "Delobel" et dont le titre contient les mots "relationnel" ou "système".

- Pour exprimer la recherche textuelle qui porte sur les rubriques de type TEXT, nous avons défini l'opérateur de relation LIKE. Il permet la sélection des notices identifiant les documents dont le contenu correspond à une condition donnée par une expression de filtrage.

Le tableau de la figure 3.12. exprime les règles d'utilisation des opérateurs de relation.

| TYPE / OP. | INTEGER | REAL | DATE | STRING | TEXT |
|---|---|---|---|---|---|
| = | oui | oui | oui | oui | non |
| != | oui | oui | oui | oui | non |
| > | oui | oui | oui | non | non |
| < | oui | oui | oui | non | non |
| >= | oui | oui | oui | non | non |
| <= | oui | oui | oui | non | non |
| LIKE | non | non | non | non | oui |

**Figure 3.12.** Règles d'utilisation des opérateurs de relation

La syntaxe générale d'un énoncé de recherche par rapport au type **STRING** est la suivante :

<rubrique> <opérateur> <chaîne>  où <rubrique> est un nom de rubrique.

<opérateur> ::= "=", "!="

<chaîne> ::= <sous-chaîne> {<joker> <sous-chaîne>}

<joker> ::= "%", "_"

Il est possible de spécifier partiellement une chaîne, en utilisant des opérateurs "joker" qui peuvent être placés à l'intérieur de la chaîne. Deux types de joker sont définis :

- l'opérateur joker "**%**", qui remplace une sous-chaîne de longueur quelconque, y compris de longueur nulle.

  Exemple :

  titre = "syst%Unix"

  Dans cet énoncé, nous voulons les notices identifiant les documents dont le titre est "Le système Unix", "Le système d'exploitation Unix", etc.

- l'opérateur joker "_", qui remplace un caractère seulement.

Exemple :

auteurs = "L_mbert"

Cette requête provoque la sélection des notices relatives aux documents dont l'auteur est "Lambert", "Lembert", etc.

La syntaxe générale d'un énoncé de **recherche textuelle** est la suivante [Jimenez 89]:

<rubrique> **LIKE** <expression_de_filtrage> où <rubrique> est un nom de rubrique.

<expression_de_filtrage> ::= ( <expr_and> { **OR** <expr_and}*)

<expr_and> ::= <terme> { **AND** <terme> }*

<terme> ::= { NOT } <chaîne>

<chaîne> ::= {"*"} <sous-chaîne>{<joker><sous-chaîne>{"!"{[n]}}}* {"*"}

<joker> ::= "*", "$", "!"{[n]}

Dans une expression de filtrage nous pouvons donc spécifier des combinaisons logiques de chaînes de caractères via les opérateurs logiques AND, OR et NOT. Ces expressions logiques sont en forme disjonctive, la précédence des opérateurs étant implicite. Nous pouvons aussi spécifier partiellement une chaîne, en utilisant des opérateurs "joker". Trois types de joker sont définis :

- l'opérateur joker **"*"**, qui peut être placé n'importe où dans la chaîne. Sa portée est le texte complet.

  Exemple :

  résumé LIKE ("syst*infor*" AND "*expert*")

  Dans cet énoncé, nous voulons les notices identifiant les documents qui parlent de "systèmes d'information pour experts", "systèmes experts d'information", "experts en systèmes d'information", etc.

- l'opérateur joker **"$"**, qui peut être placé entre deux sous-chaînes. Sa portée est défini comme étant limitée à une phrase dans un texte.

- l'opérateur joker **"!"**[n], peut être placé entre deux sous-chaînes, ou à la fin de la chaînes de caractères. Sa portée est limitée à maximum n caractères dans un mot du texte. Si n n'est pas spécifié, la valeur n = 1 est prise par défaut.

Exemple :

résumé LIKE ("impressionnis!3")

permet de retrouver les notices relatives aux documents qui parlent d'"impressionnisme" ou d'"impressionnistes".

### 3.3.2. Implantation de la recherche textuelle

Au niveau du serveur BIB, deux fonctions sont définies : une pour calculer la signature CS du texte d'une rubrique de type TEXT lors de la mise à jour d'une notice, et une fonction d'évaluation d'un énoncé de recherche.

Les fonctions de base pour la recherche textuelle sont :

1) **CREER_SIGNAT** (texte)

La signature CS d'un texte (ST) est formée par la superposition des signatures individuelles des mots qui le composent. Une ST est un bloc de bits de taille fixe (N) représentant la chaîne de caractères de longueur maximum de 64 Koctets (c'est la taille d'un LONG). Dans le calcul de ST sont codés tous les mots, ainsi que les séparateurs entre les mots (contrairement aux méthodes classiques qui prennent des blocs de taille fixe, éliminent les mots non significatifs de la langue et ne gardent pas les séparateurs).

La valeur de N a été choisie de façon à optimiser à la fois la place occupée par la signature du texte et la probabilité de collision (le bit i de ST est mis à "1" par au moins deux mots différents). Dans notre cas, le nombre N de bits par bloc a été fixé à 300. La signature est donc stockée dans la base sous la forme d'une chaîne d'octets de longueur (300-1)/8+1 = 38, qui correspond au type Oracle, RAW(38).

Pour permettre la spécification de parties de mots dans les expressions de filtrage, la fonction de transformation effectue le découpage des mots en triplets superposés (se reporter au paragraphe 3.2.2). Les triplets sont formés en tenant compte de tous les caractères du texte (les caractères multiples éventuels de séparation entre les mots sont réduits à un seul). Ceci permet aussi de préserver l'ordre entre les mots, en utilisant le triplet avec le séparateur et les triplets avec les caractères de fin et début de mot. La fonction de transformation calcule **le codage de chaque triplet** dans le texte T, vers un bit dans ST. Le vocabulaire traité est donc l'ensemble des triplets possibles (et non pas les mots de texte comme c'est le cas classique).

La fonction CREER_SIGNAT(texte) qui calcule la signature d'un texte comporte donc plusieurs étapes :

- **Identification des triplets de caractères** : la formation des triplets tient compte de tous les mots dans le texte.

- **Application de la fonction de hachage** : chaque triplet est considéré comme un chiffre d'un nombre en base 127 ($2^7$-1), calculé à partir de la représentation hexadécimale de chaque caractère, puis il est transformé en base décimale et converti enfin en un entier dans l'intervalle [0 ...N-1] en calculant son modulo à N. Le chiffre ainsi obtenu est l'image du triplet, c'est à dire la signature du triplet.

  Exemple :

  Pour une taille de bloc N = 199 bits,

  l'image du triplet "abc" est 1056834 mod 199 = 144

  et l'image du triplet "bca" est 1073088 mod 199 = 80

- **Formation du bloc de signatures** : le bloc signature ST d'un texte est initialisé à "0" et pour chaque triplet, le bit correspondant à la signature du triplet est mis à "1". Ceci constitue en fait le OR des signatures des triplets individuels.

  Dans l'exemple précédent, les bits des positions 80 et 144 dans ST ont alors la valeur "1".

Cette fonction est appelée de manière systématique lors de la mise à jour d'une rubrique de type TEXT. La signature ainsi calculée est ensuite stockée dans la base et elle est prête à être filtrée lors de l'évaluation des expressions de recherche.

2) **MATCH_BIB** (base, nrnot, idfdoc, idfrub, requête, flag)

Cette fonction nous permet d'évaluer une expression de filtrage *requête* sur le texte de la rubrique *idfrub*, de la notice *nrnot* appartenant à la base de l'usager *base*. La notice étudiée a le format associé au type de document *idfdoc*. MATCH_BIB effectue le processus d'élimination des collisions si la valeur de *flag* est à "1".

Son algorithme simplifié est le suivant :

```
MATCH_BIB (base, nrnot, idfdoc, idfrub, requête, flag)
début
    Analyse_syntaxique (requête)
    Sreq = Génération_SignReq (requête)
    (ST, idfcont) = Récup_Sign (base, nrnot, idfdoc, idfrub)
    Rés = Eval_Signature (Sreq, ST)
    si (Rés = faux)
    alors
        return (nrnot n'est pas sélectionnée)
    sinon
        si (flag = 1)
        alors
            T = Récup_Texte (base, idfcont)
            Génération_Automate (requête)
            Rés = Exécution_Automate (requête, T)
            si (Rés = vrai)
            alors
                return (nrnot est sélectionnée)
            sinon
                return (nrnot n'est pas sélectionnée)
            finsi
        sinon
            return (nrnot est sélectionnée)
        finsi
    finsi
fin
```

La méthode de calcul et d'évaluation de la signature de la requête envisage quatre cas distincts :

1) la requête est **une chaîne de caractères** :

La signature de la requête Sreq est un bloc de bits de taille N, tel qu'à chaque triplet de la requête correspond un bit dans Sreq. Dans un bloc de signature d'un texte est perdue la notion de phrases et de mots; en fait tous les triplets sont codés dans le même ensemble de bits, de manière homogène. Les jokers "$" et "!" sont donc traités au niveau de l'évaluation de la même manière que "*". C'est seulement lors de l'élimination des collisions par l'automate qu'ils peuvent être distingués. Donc, les signatures de toutes les sous-chaînes de la requête doivent apparaître dans la signature du texte de la rubrique, pour que la notice soit sélectionnée.

2) la requête est **la négation d'une chaîne de caractères :**

Dans ce cas, une notice est sélectionnée que si le texte de la rubrique ne correspond pas à la requête. Sreq est calculée de la même façon que dans le cas sans négation, c'est seulement le critère de sélection du texte qui change.

3) la requête est **une conjonction de chaînes de caractères** (c'est une expr_and, voir le paragraphe 3.1.1) :

La notice est sélectionnée si chacune des signatures de chaînes a été vérifiée dans la signature du texte de la rubrique.

4) la requête est **une disjonction d'expressions conjonctives** (c'est une expr_or) :

Sreq est formée par un arbre, où chaque fils de la racine représente la signature d'une expression conjonctive, celle-ci étant à son tour composée d'autant de blocs qu'elle contient de chaînes de caractères (termes). Une notice est sélectionnée si le texte de la rubrique satisfait au moins une expr_and de la requête.

Si l'application désire une réponse exacte, il est nécessaire de procéder au filtrage du texte lui-même. A chaque requête correspond un automate augmenté d'une mémoire qui permet de la reconnaître dans le texte pré-sélectionné [Jimenez 89].

### 3.3.3. Implantation de la formule de consultation par rubrique

Le serveur BIB fournit deux fonctions qui permettent l'interrogation de la base suivant la formule de consultation par rubrique : une pour déterminer les notices qui vérifient une requête de sélection, et une fonction qui récupère les numéros des notices qui ont été sélectionnées.

1) **SELECT_RUBCOMM** (base, requête)

Cette fonction permet de sélectionner les notices d'une base de nom d'usager *base*, qui vérifient la requête de sélection *requête*; elle comporte quatre étapes principales :

- Analyse syntaxique de la requête de sélection.
- Construction de l'arbre abstrait de représentation, avec vérification de la sémantique de la requête (vérification des noms de rubrique, compatibilité entre les types : la rubrique, l'opérateur et la valeur doivent être compatibles).
- Application de la distributivité de l'opérateur AND sur l'opérateur OR avec la transformation de l'arbre abstrait.

• Construction de l'arbre des instructions.

Nous présentons un exemple pour expliquer la méthode employée :

Exemple :

(auteurs = "Delobel" AND date > "01011982") **AND**
(résumé LIKE ("*relationnelle*") OR mots-clés = "relationnelle" )

L'arbre abstrait correspondant est de la forme :

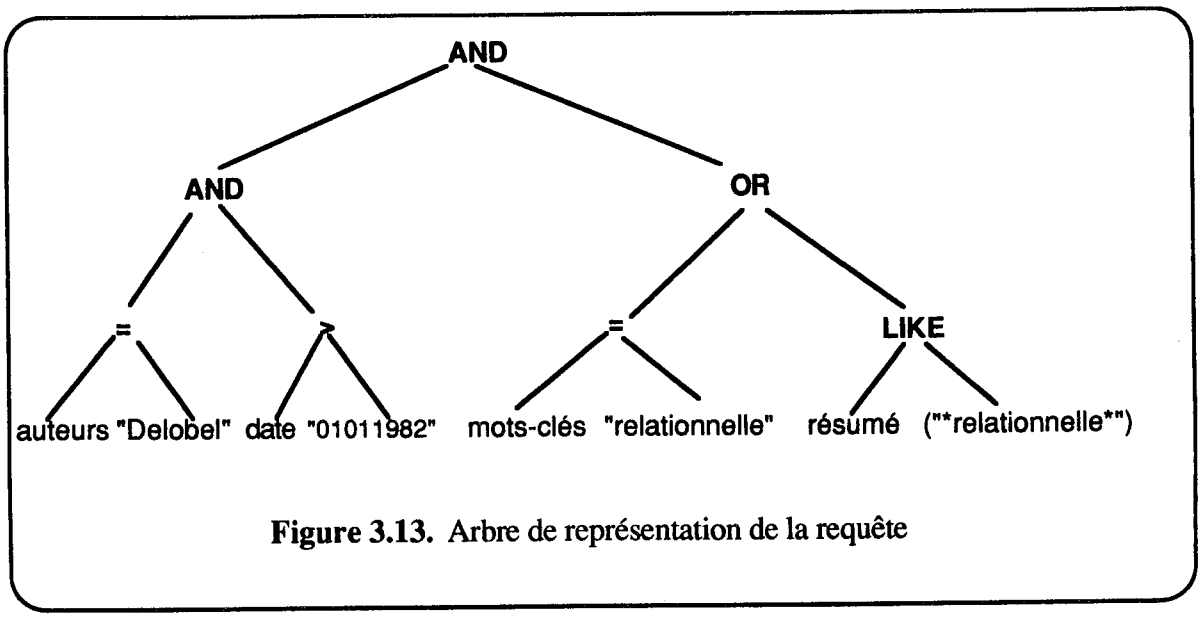



**Figure 3.13.** Arbre de représentation de la requête

Si nous appliquons la distributivité de l'opérateur AND sur l'opérateur OR, la requête résultante est une disjonction d'expressions conjonctives de la forme :

(auteurs = "Delobel" AND date > "01011982" AND résumé LIKE ("*relationnelle*"))
**OR** (auteurs = "Delobel" AND date > "01011982" AND mots-clés = "relationnelle")

L'arbre de représentation est alors de la forme :

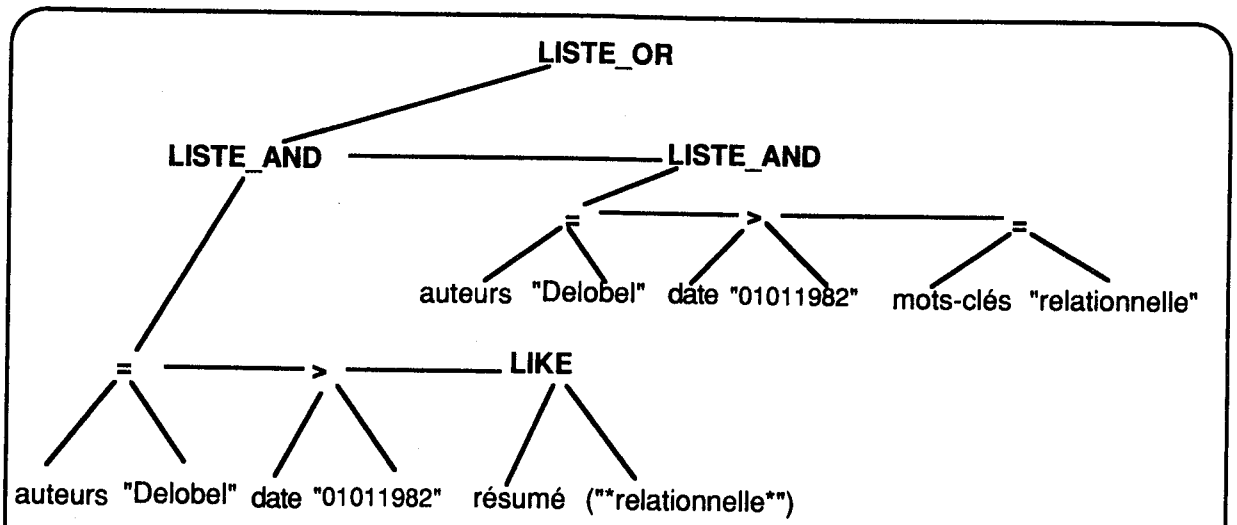



**Figure 3.14.** Arbre final de représentation de la requête

A partir de cet arbre, il est possible de construire l'arbre des instructions :

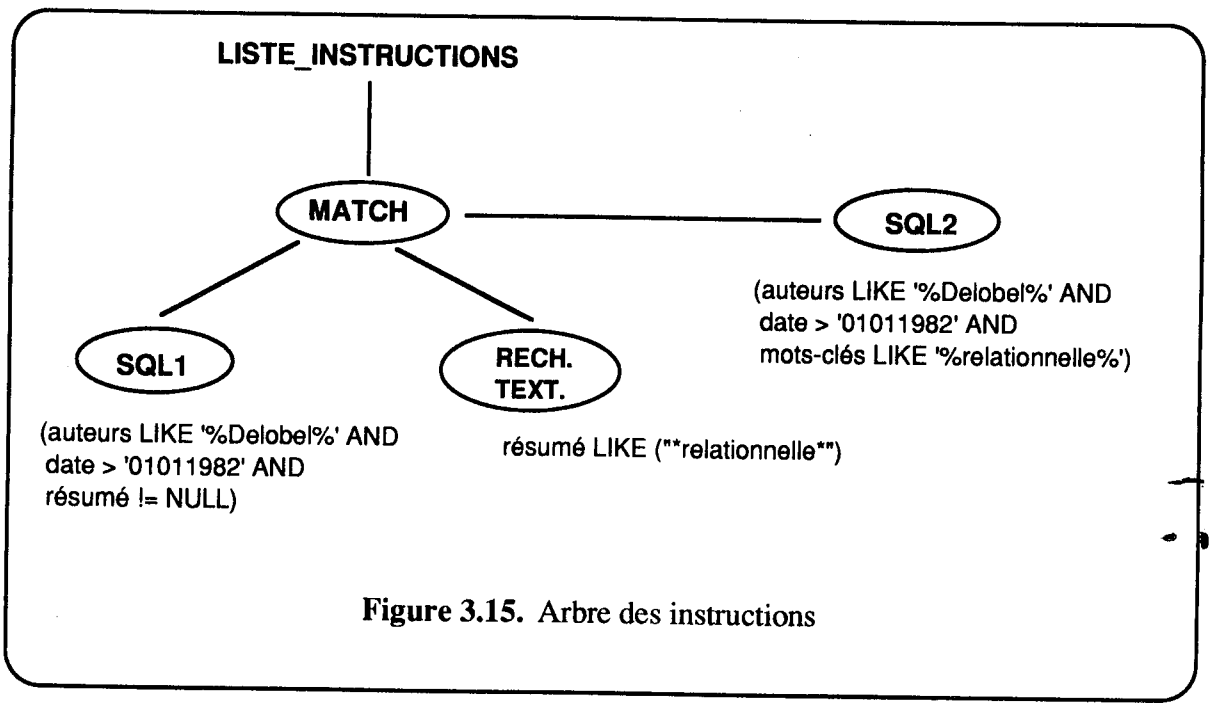



**Figure 3.15.** Arbre des instructions

L'instruction SQL1 correspond à la transformation de l'expression formulée dans la requête en un prédicat SQL tel que :

(auteurs LIKE '%Delobel%' AND date > '01011982' AND résumé != NULL)

Ce prédicat est évalué pour chaque ligne de toutes les tables contenant les notices, quel que soit le type de document.

Cet énoncé sélectionne donc les notices (repérées par leur numéro) qui identifient les documents dont l'auteur est "Delobel", le titre comporte le mot "système", et le résumé est quelconque pourvu qu'il soit renseigné. Ensuite pour affiner la sélection, la recherche textuelle est effectuée sur l'ensemble des notices pré-sélectionnées par l'instruction SQL1. L'expression de filtrage ("*relationnelle*") est évaluée sur le texte de la rubrique *résumé*. Le résultat est l'ensemble des notices satisfaisant la première expression conjonctive, (auteurs = "Delobel" AND date > "01011982" AND résumé LIKE ("*relationnelle*")).

L'instruction SQL2 exécute la requête SQL qui spécifie le prédicat, constitué à partir de la seconde expression conjonctive :

(auteurs LIKE '%Delobel%' AND date > '01011982' AND mots-clés LIKE '%relationnelle%')

Ainsi, le parcours de l'arbre des instructions de la gauche vers la droite donne comme résultat deux ensembles de notices pas forcément disjoints, que nous devons regrouper en un ensemble unique après avoir éliminé les numéros de notices en double. Les numéros de notices ainsi obtenus sont stockés dans une table unique. La fonction SELECT_RUBCOMM retourne à l'application le nombre de notices sélectionnées.

Le mécanisme, appliqué ici sur un exemple simple est valable quel que soit le degré de complexité de la requête de sélection.

L'analyse syntaxique de la requête de sélection a été mise en oeuvre à l'aide de l'outil YACC, qui permet la génération d'analyseurs syntaxiques. YACC est un module d'Unix, qui utilise comme données une grammaire ainsi que des fragments de programmes C pour produire un programme qui analyse des données en accord avec cette grammaire. Lorsqu'une règle est reconnue, le fragment de programme associé est exécuté. L'analyse lexicale effectuée avant la vérification syntaxique est réalisée avec un autre module Unix, LEX. Le programme LEX possède une interface adaptée à YACC.

La manipulation des arbres est modélisée grâce à un outil de gestion de structures arborescentes attribuées, EMIR (Easy Manipulation Internal Representation), développé au Centre de recherche Bull de Grenoble. Sa mise en oeuvre est rapide et facile (description simple des arbres, visualisation des sous-arbres, ...). EMIR permet la manipulation symbolique et fonctionnelle des arbres, grâce à la définition de concepts (noeuds, classes de noeuds, attributs, symboles) et de fonctions de haut niveau.

2) NOTICE_SUIV ()

Cette fonction retourne à chaque appel le numéro de notice suivant stocké dans la table, qui mémorise l'ensemble des numéros de notices sélectionnées lors de l'appel à la fonction SELECT_RUBCOMM. La fonction, qui récupère les informations contenues dans une notice à partir de son numéro doit alors être appelée par l'application.

## 4. Conclusion

Un des avantages importants du système de gestion bibliographique proposé ici, concerne la généricité des formats des notices bibliographiques : un groupe d'utilisateurs a la possibilité d'adapter à son goût les descriptions bibliographiques des documents primaires. Un autre avantage intéressant est le langage d'interrogation qui permet de constituer des requêtes de sélection combinant n'importe quelles rubriques, y compris des rubriques textuelles.

Le serveur BIB offre des fonctions pour la gestion de la bibliographie qui donne au programmeur les possibilités suffisantes pour construire des interfaces conviviales, et qui lui cache les mécanismes internes liés au modèle de données sous-jacent. Chaque fonction BIB retourne des codes d'erreurs, qui signalent à l'application appelante les anomalies détectées sur les données passées à l'appel. C'est à l'application de gérer le dialogue avec l'usager.

# Conclusion

Les premières conclusions concernent la réalisation et les résultats obtenus. Le serveur Opidum, permettant l'accès à distance à Oracle, a été utilisé à plusieurs reprises lors du développement d'applications s'appuyant sur le SGBD. Une partie de notre travail a donc consisté à assurer le support aux programmeurs. Nous avons pu constater que cette première réalisation a produit un logiciel fiable et opérationnel, répondant aux spécifications.

Le serveur Opidum est une couche logicielle construite sur l'interface HLI d'Oracle, par conséquent il devra être maintenu à chaque changement de version du SGBD, entraînant des modifications du code HLI.

La deuxième réalisation n'est pas arrivée à son terme. En effet, toutes les fonctionnalités du serveur d'Information Bibliographique n'ont pas été implémentées. Les fonctions de mise à jour et consultation ont été codées qu'en local; la mise au point des agents client et serveur demande une lourde programmation à caractère répétitif.

En effet, le principe général d'implémentation du modèle "un serveur par client" avec le logiciel d'appel de procédure à distance, RPC de Sun, a été défini lors de la réalisation du serveur Opidum. Désormais, la méthode établie est appliquée à chaque nouvelle implantation du même type. Il serait donc souhaitable d'envisager la mise au point d'un outil qui génère de façon automatique, les modules serveur général, agents client et serveur.

De toute manière, le serveur BIB est facilement extensible; le codage de la fonction d'administration de la base n'est pas urgent car l'administrateur dispose actuellement d'un programme, qui lit des fichiers décrivant le format des notices bibliographiques à inclure dans la base. Des interfaces plus conviviales (par exemple de type X-WINDOWS) devront être développées si l'on veut que l'outil soit largement utilisé.

Le travail effectué dans le cadre de ce mémoire est très important pour nous car il correspond à une ouverture à de nouvelles compétences dans plusieurs domaines d'actualité : les SGBD relationnels, la communication , Unix et la programmation C.

La réalisation du serveur Opidum nous a apporté une bonne connaissance du langage C et du système Unix, notamment de la gestion des processus et de la communication inter-processus.

L'emploi du mécanisme d'appel de procédure à distance nous a permis de nous rendre compte de la simplification qu'il apporte dans l'écriture d'applications distribuées et de l'étendue de son champ d'application.

Le serveur d'Information Bibliographique constitue une bonne expérience en tant que conception d'une application répartie et aussi en tant qu'utilisation du SGBD Oracle.

Enfin, nous avons beaucoup apprécié l'environnement riche en compétences multiples, au sein duquel nous avons évolué pendant une année.

# Annexe 1

# Les protocoles TCP/IP

### *Le protocole TCP*

TCP est un protocole an niveau de la couche transport du modèle OSI. Il est orienté *flot d'octets* : il transmet autant d'octets qu'il le peut au sein d'une connexion donnée, avec concaténation possible des messages émis.

TCP travaille en mode connecté. La communication se fait en mode point à point.

Les octets sont rassemblés dans une série de *segments* dont la longueur maximale est fixée pour une connexion donnée, par négociation entre les deux machines établissant cette connexion.

La fiabilité du transfert est assurée par acquittement des segments et retransmission si nécessaire. De plus, il y a un contrôle de flux basé sur le système de fenêtrage en émission et en réception. TCP est le protocole de transport le plus utilisé.

### *Le protocole IP*

IP est le protocole de réseau sur lequel s'appuie TCP pour transmettre ses segments. Ce protocole remplit deux grandes fonctions :

- le routage,
- la fragmentation.

Rappel :

Une *passerelle* est un système qui connecte un réseau avec un ou plusieurs autres réseaux. Les passerelles sont soit des ordinateurs non dédiés ayant plusieurs contrôleurs de communication, soit des dispositifs spécialisés.

Le *routage* détermine la machine passerelle qui connaît un chemin vers la machine de destination, lorsque cette dernière n'est pas accessible directement.

Le routage dans IP, est basé entièrement sur le numéro de réseau des adresses de destination. Chaque ordinateur a une table des numéros de réseau; pour chaque numéro de réseau une passerelle est indiquée : c'est la passerelle à emprunter pour aller sur ce réseau. Afin que deux machines de réseaux différents puissent communiquer, on passe par une ou plusieurs passerelles comme le montre la figure A1.1.

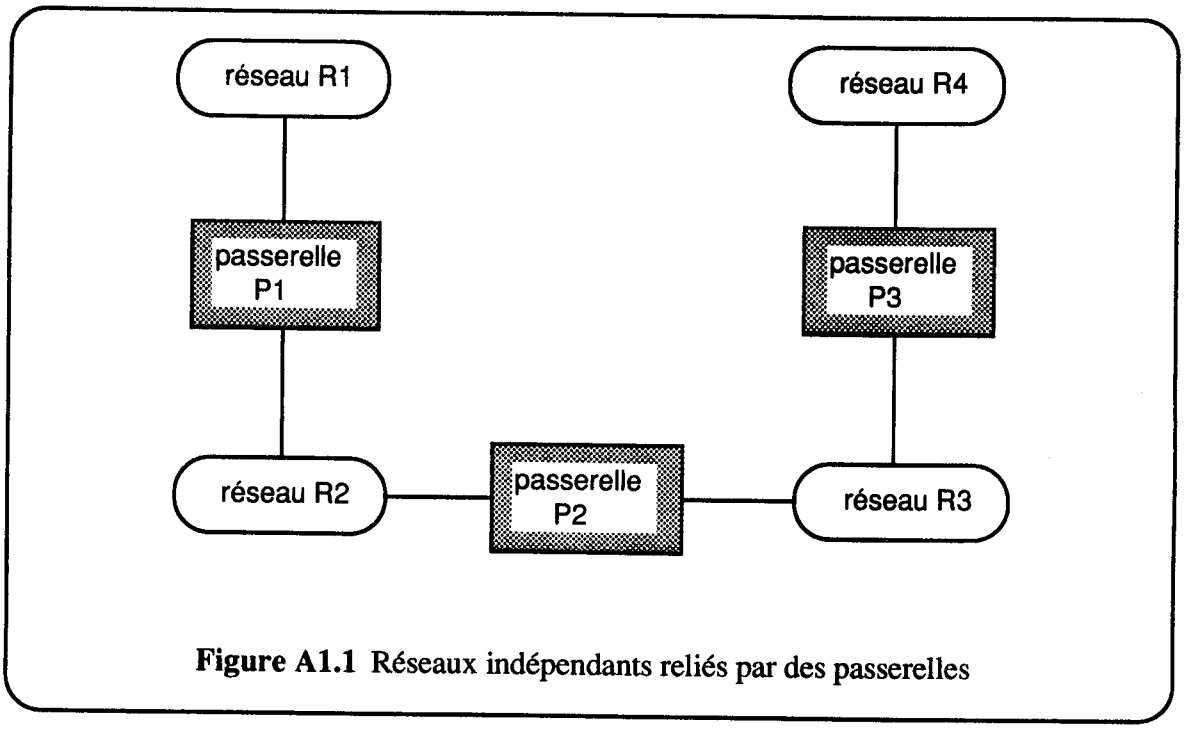



**Figure A1.1** Réseaux indépendants reliés par des passerelles

Grâce à une table de routage, on saura que :

- pour aller du réseau R1, il faut envoyer à la passerelle P1,
- P1 va recevoir le paquet d'informations de R1 et l'envoyer à R2,
- etc ...

Le routage s'effectue dans les couches basses de TCP/IP, et reste invisible pour les utilisateurs. En effet, le routage se fait de proche en proche et au niveau de IP. Dans les machines intermédiaires, on ne passe que par IP; ce n'est que dans la machine destinataire que TCP reçoit le paquet.

La *fragmentation* est la phase qui suit le routage. Au départ de la machine émettrice et de chaque passerelle, les segments TCP sont découpés en fragments IP de taille maximale égale à l'unité maximum de transmission (MTU : Maximum Transmit Unit) de l'interface choisie pour le routage.

A l'arrivée sur chaque passerelle et sur la machine destinataire, IP réassemble les fragments pour recomposer le segment initial. Les fragments d'un segment incomplet sont détruits après expiration d'un "timeout" spécifié dans l'entête de chaque fragment. Ainsi, IP ne conserve jamais de fragment parasite.

Si un fragment est perdu, TCP reémet le segment en entier. Chaque segment contient les adresses des machines source et destinataire.

L'un des intérêts de IP est la faculté de s'adapter aux possibilités de tout média car non seulement il fragmente un segment à la dimension des unités de transfert du média, mais il réassemble les fragments d'un même segment arrivés dans n'importe quel ordre.

Les protocoles TCP/IP sont actuellement le standard le plus répandu sur leur principal support : Ethernet.

# Annexe 2

# Exemple d'utilisation du logiciel RPC/Sun

Nous décrivons dans cette annexe, l'utilisation du logiciel RPC/Sun dans le cas d'un serveur d'addition, implanté sur la machine "toutatis". Les différents modules sont écrits en langage C :

- add.x : interface RPC/Sun
- addsv.c : programme serveur
- addcl.c : programme client implanté sur la machine "idefix"
- add.h : déclarations communes générées par RPCGEN
- add_svc.c : stub serveur généré par RPCGEN
- add_clnt.c : stub client généré par RPCGEN
- add_xdr.c : fonctions XDR de conversion

---

**add.x :** fichier regroupant les déclarations de la structure passée en appel, et de la procédure distante, svadd(). Ce fichier est à définir par le programmeur avant d'exécuter RPCGEN.

---

```
struct nbres {
   int opd1;
   int opd2;
};
program ADDPROG {
   version ADDVERS {
       int SVADD (nbres) = 1;
   } = 1;
} = 91;
```

## addsv.c : programme serveur, définition de la procédure, svadd().

```
#include <sys/types.h>
#include <rpc/rpc.h>
#include "add.h"  /* déclarations communes générées par rpcgen */
int *
svadd_1 (operat)
  nbres *operat;
  {
      static int result;      /* doit être déclaré en static*/
      result = operat -> opd1 + operat -> opd2;
      return (&result);
  }
```

## addcl.c : programme client, appel à la procédure distante svadd_1().

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
#include "add.h"  /* déclarations communes générées par rpcgen */
main (argc, argv)
int argc;
char *argv[];
{
   CLIENT *cl;
   int *result;
   int opd1, opd2;
   nbres calcul;
   char *serveur = "toutatis";      /* machine du serveur */
   if (argc > 3) {
       fprintf (stderr, "trop d'arguments passes en parametres\n"");
       exit (1);
   }
   opd1 = atoi (argv [1]);
   opd2 = atoi (argv [2]);
   /* affectation des deux nombres dans la structure de communication */
   calcul.opd1 = opd1;
   calcul.opd2 = opd2;
   /*creation du lien avec le serveur */
   cl = clnt_create (serveur, ADDPROG, ADDVERS, "tcp");
   if (cl == NULL) {    /* impossible d'établir la communication */
       clnt_pcreateerror (serveur);
       exit (1);
   }
   result = svadd_1 (&calcul, cl); /*on passe l'adresse de la structure */
   if (result == NULL) {
       clnt_perror (cl, serveur);
       exit (1); }
   printf (" resultat de l'addition = %d \n", *result);
}
```

## add.h : fichier des déclarations communes généré par rpcgen.

```
struct nbres {
  int opd1;
  int opd2;
};
typedef struct nbres nbres;
bool_t xdr_nbres();

#define ADDPROG ((u_long)91)
#define ADDVERS ((u_long)1)
#define SVADD ((u_long)1)
extern int *svadd_1();
```

## add_svc.c : squelette du serveur généré par rpcgen.

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
#include "add.h"

static void addprog1_1();

main()
{
  SVXPRT *transp;
  pmap_unset (ADDPROG, ADDVERS);
  transp = svcudp_create (RPC_ANYSOCK);
  if (transp == NULL) {
      fprintf (stderr, "cannot create udp service.\n");
      exit (1);
  }
  if (!svc_register (transp, ADDPROG, ADDVERS, addprog1_1, IPROTO_UDP)) {
      fprintf (stderr, "unable to register (ADDPROG, ADDVERS, udp). \n");
      exit (1);
  }
  transp = svcudp_create (RPC_ANYSOCK, 0, 0);
  if (transp == NULL) {
      fprintf (stderr, "cannot create udp service.\n");
      exit (1);
  }
  if (!svc_register (transp, ADDPROG, ADDVERS, addprog1_1, IPROTO_TCP)) {
      fprintf (stderr, "unable to register (ADDPROG, ADDVERS, tcp). \n");
      exit (1);
  }
  svc_run();
  fprintf (stderr, "svc_run returned\n");
  exit (1);
}
```

```
static void
addprog1_1 (rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
  union {
     nbres svadd_1_arg;
  } argument;
  char *result;
  bool_t (*xdr_argument) (), (*xdr_result) ();
  char *(*local) ();

  switch (rqstp -> rq_proc) {
     case NULLPROC :
           svc_sendreply (transp, xdr_void, NULL);
           return;
     case SVADD :
           xdr_argument = xdr_nbres;
           xdr_result = xdr_int;
           local = (char *(*) () ) svadd_1;
           break;
     default :
           svcerr_noproc (transp);
           return;
  }
  bzero (&argument, sizeof (argument));
  if (!svc_getargs (transp, xdr_argument, &argument)) {
     sverr_decode (transp);
     return;
  }
  result = (*local) (&argument, rqstp);
  if (result != NULL && !svc_sendreply (transp,xdr_result, result)) {
     svcerr_systemerr (transp);
  }
  if (!svc_freeargs (transp, xdr_argument, &argument)) {
     fprintf (stderr, "unable to free arguments\n");
     exit (1);
  }
}
```

## add_clnt.c : stub client généré par rpcgen.

```
#include <sys/types.h>
#include <rpc/rpc.h>
#include <sys/time.h>
#include "add.h"

static struct TIMEOUT = {25, 0};

int *
svadd_1 (argp, clnt)
nbres *argp;
CLIENT *clnt;
{
   static int res;

   bzero (&res, sizeof (res));
   if (clnt_call (clnt, SVADD, xdr_nbres, argp, xdr_int, &res, TIMEOUT)
       != RPC_SUCCESS)
    {
       return (NULL);
    }
   return (&res);
}
```

## add_xdr.c : fonctions XDR de conversion générées par rpcgen.

```
#include <sys/types.h>
#include <rpc/rpc.h>

bool_t
xdr_nbres (xdrs, objp)
XDR *xdrs;
nbres *objp;
{
   if (!xdr_int (xdrs, &objp -> opd1)) {
       return (FALSE);
   }
   if (!xdr_int (xdrs, &objp -> opd2)) {
       return (FALSE);
   }
   return (true);
}
```

Les directives de compilation et d'éxecution sur la machine "toutatis" sont les suivantes :

toutatis% **rpcgen add.x**
toutatis% **cc addsv.c add_svc.c add_xdr.c -o sv_add**
toutatis% **sv_add &**

Les directives de compilation et d'éxecution sur la machine "idefix" sont les suivantes :

idefix% **rpcgen add.x**
idefix% **cc addcl.c add_clnt.c add_xdr.c -o add**
idefix% **add (2, 44, 56)**

Le serveur retourne la valeur "100", le programe client affiche alors le résultat de l'addition à l'écran.

# Annexe 3

# Manuel d'utilisation du logiciel Opidum

**Centre de Recherche Groupe**

Division Organisation des Systèmes d'Information

BULL c/o IMAG
BP 53X
38041 GRENOBLE CEDEX

# OPIDUM

# UNE INTERFACE PROCEDURALE DE HAUT NIVEAU POUR ORACLE DANS UN ENVIRONNEMENT DISTRIBUE

**Marie-Christine PAJON**

**DSG/CRG/OSI**

## RESUME

Le développement de la nouvelle version du logiciel SOUPE est une retombée du projet DOEOIS (Design and Operational Evaluation of Office Information Servers), mené par le Centre de Recherche Bull de 1985 à 1988. La particularité de cette nouvelle version renommée OPIDUM est son aspect **distribué**. **OPIDUM**, qui signifie **O**racle **P**rocedural **I**nterface for **D**istributed **U**nix **M**achines, offre une interface Oracle de haut niveau pour la programmation en langage C, réunissant les avantages du précompilateur Oracle (expression des requêtes en clair) et de l'interface HLI (élimination de la phase de précompilation). Le cycle de développement des logiciels se trouvent ainsi écourté. L'accès à Oracle se fait par appel de fonctions acceptant en paramètre la chaîne en clair de la requête SQL. Par un mécanisme de substitution de variables à l'appel, les requêtes peuvent être paramétrées. La lisibilité des programmes est ainsi accrue, et la maintenance facilitée. L'utilisation d'OPIDUM permet à une application résidente sur une machine, d'accéder à un SGBD Oracle implanté sur toute machine, connectée sur le même réseau de communication.



## ABSTRACT

The new release of the software package SOUPE, named OPIDUM, is a consequence of the OIS project which was managed by the Bull Research Center beetwen 1985 and 1988. OPIDUM is a software layer to provide a **distributed processing environment** for the applications using the Oracle DBMS. **OPIDUM** which means **O**racle **P**rocedural **I**nterface for **D**istributed **U**nix **M**achines, provides a high-level Oracle programming interface for the language C, having both the advantages of the precompiler (queries simply expressed in SQL) and HLI (no precompilation phase). In this way the software development cycle is shortened. Access to Oracle is done by function calls where SQL statements are given as parameter. SQL statements themselves can contain variables thus allowing parameterized queries. The use of OPIDUM increases program readibility and maintainability. OPIDUM allows an application running on one machine on a network to access and manipulate data in an Oracle database that resides an other machine anywhere on that network.

**MOTS-CLE**

ORACLE
Interface pour la programmation
SQL
SGBD
Appel de procédure à distance

**KEY-WORDS**

ORACLE
Programming Interface
SQL
DBMS
Remote procedure call

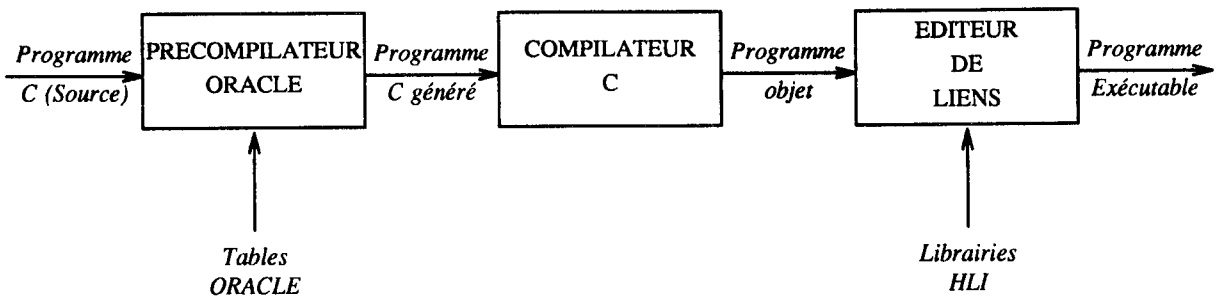
# 1. PRESENTATION

## 1.1 Caractéristiques

Le développement de la nouvelle version du logiciel SOUPE est une retombée du projet DOEOIS (Design and Operational Evaluation of Office Information Servers), mené par le Centre de Recherche Bull de 1985 à 1988. La particularité du nouveau SOUPE renommé **OPIDUM** est son aspect **distribué.**

**OPIDUM**, qui signifie **O**racle **P**rocedural **I**nterface for **D**istributed **U**nix **M**achines, offre une interface Oracle de haut niveau pour la programmation en langage C. Sa réalisation a été motivée principalement, par le désir d'éliminer la phase de précompilation des programmes contenant des instructions en langage SQL, pour les accès au SGBD Oracle. En effet, dans les cas de développement de logiciels complexes avec de nombreux accès au SGBD Oracle, la première possibilité est l'utilisation du précompilateur C d'Oracle (PRO*C), qui complique l'organisation des fichiers, multiplie le nombre de fichiers et augmente significativement les temps de compilation (précompilation plus compilation proprement dite). Le schéma ci-dessous, illustre cette situation :

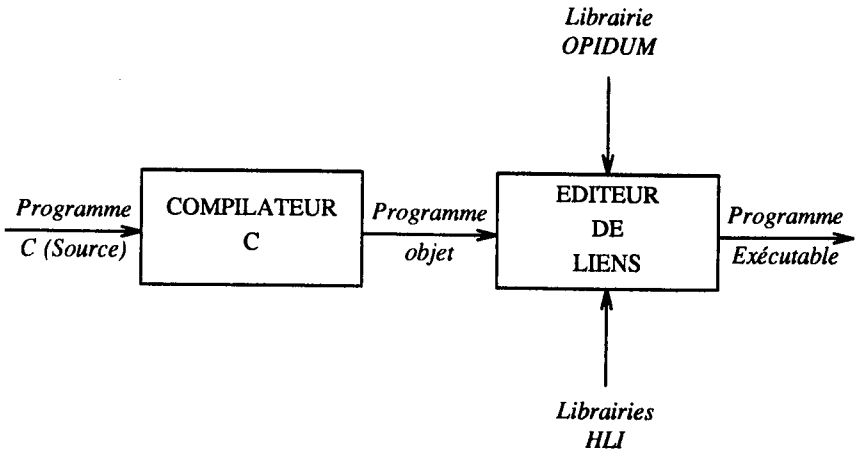
**Chaine de développement avec précompilateur C**




La seconde possibilité consiste à utiliser l'interface HLI (High Level Interface) d'Oracle, qui supprime la phase de précompilation. L'écriture des programmes devient alors plus complexe, à cause des multiples appels de fonctions nécessaires et des nombreuses structures de données à connaître et à manipuler.

**OPIDUM** est en fait une couche logicielle construite sur l'interface HLI d'Oracle, et propose une syntaxe de programmation proche des fonctions *printf* et *scanf* de la librairie standard du langage C. Le logiciel OPIDUM réunit donc les avantages du précompilateur Oracle (expression des requêtes en clair) et de l'interface HLI (élimination de la phase de précompilation). Le cycle de développement des logiciels se trouvent ainsi écourté. Le schéma ci-dessous illustre cette situation :

**Chaine de développement utilisant OPIDUM C**




L'autre intérêt d'OPIDUM est son caractère **distribué** : une application, résidente sur une machine, peut accéder à un SGBD Oracle implanté sur toute autre machine, connectée sur le même réseau de communication. L'accès aux données Oracle est donc indépendant de la localisation de l'application. La communication inter-machines dans OPIDUM fonctionne avec le logiciel d'appel de procédure à distance, **RPC** (Remote Procedure Call) de la société Sun. L'utilisation d'OPIDUM est identique quel que soit le mode, local ou distribué.
L'outil SQL*NET d'Oracle version 5 utilisé avec le précompilateur PRO*C, propose un service équivalent à OPIDUM réparti, mais toujours avec l'inconvénient de la phase de précompilation.

### 1.2 Principe

Le logiciel **OPIDUM** met à la disposition du programmeur treize fonctions, qui suffisent pour la plupart des applications de bases de données.
La requête SQL en clair, est passée en paramètre de la fonction OPIDUM, qui enchaîne les différents appels aux fonctions HLI nécessaires à l'exécution de la requête. Par un mécanisme de substitution de variables à l'appel, les requêtes peuvent être paramétrées.
La lisibilité des programmes est ainsi accrue, et la maintenance facilitée. L'utilisation de ces fonctions est simple et nécessite peu de temps d'apprentissage.
L'exemple de programme présenté ci-contre montre différentes utilisations de la fonction de plus haut niveau, **sql** :

```
main()
{
char nom[21], prenom[21];
int age;
 ...
if ( !connection(1, "test", "test"))    /* connexion au SGBD Oracle */
      {
      printf("Connexion impossible\n");
      exit(2);
      }

/* récupération automatique de résultats a l'écran      */
age = 30;
sql("select * from demopidum where age < %i", age);

/* mise a jour de l'âge de la personne "DUPONT"         */

sql("update demopidum set age = %i where nom = 'DUPONT'", 40);

/* récupération de valeurs dans des variables et affichage des résultats
   par fonction */

sql("select nom &20s, prenom &20s, age &i from demopidum {f}",
               nom,          prenom,    &age,              traiter_tuple);
 ...

/* déconnexion d'Oracle                                  */

connection(0);
 ...
}

traiter_tuple()
{
printf("nom : %s prenom : %s (%d)\n", nom, prenom, age);
return(1);
}
 ...
```

La description et l'utilisation des fonctions OPIDUM sont explicitées dans le chapitre suivant.

## 2. LES FONCTIONS

### 2.1 La fonction sql

La fonction sql permet d'exécuter n'importe quelle requête SQL. Dans le cas d'un SELECT, elle réalise la récupération automatique des tuples sélectionnés.

#### *2.1.1 Description*

L'interface de la fonction sql est la suivante :

```
int sql (requête_sql [, paramètres] ... )
char *requête_sql;
```

Le premier paramètre d'appel est la requête SQL à exécuter, elle peut contenir des caractères spéciaux tels que "%", "&", ... qui décrivent le type des paramètres suivants. Trois types de paramètres sont prévus :

1. Variable du programme d'application, à prendre en entrée, dans la requête SQL.
2. **Adresse** d'une variable globale du programme d'application, cette variable est destinée à recevoir la valeur d'un attribut sélectionné lors d'un SELECT.
3. Adresse d'une fonction de type entier de l'application, exécutée lors de la récupération de chaque tuple par un SELECT.

Il existe donc trois types de descripteurs de paramètres, pouvant être insérés dans la requête SQL :

1. **%t** où **t** indique le type de variable du programme, à prendre en entrée. Le type **t** est défini dans le tableau qui suit.
   Pour le type **b**, **%nt** est utilisé, où **n** indique le nombre de bits à charger.
2. **&nt** où **t** indique le type de la variable de l'application, qui reçoit la valeur d'un attribut selectionné, lors d'un SELECT.
   **n** est un entier, donnant le nombre de caractères de la valeur de l'attribut, à récupérer dans la variable de sortie. **n** est demandé pour les types, chaîne de caractères et chaîne d'octets seulement. Le tableau qui suit décrit l'ensemble des valeurs de **t**, et le type C correspondant :

| Valeur de t | Signification | Type C |
|---|---|---|
| a, A<br>non valable pour % | adresse d'une chaîne de caractères pour nom d'attribut | char[n] |
| c | caractère contenant une valeur d'attribut | char |
| C | caractère | char |
| f, F | réel | float |
| i, I, d, D | entier | int |
| l, L | adresse d'un long | t_ora_long *<br>(décrit dans 5.2.1). |
| s | adresse d'une chaîne de caractères contenant une valeur d'attribut | char[n] |
| S | adresse d'une chaîne de caractères | char[n] |
| b, B | adresse d'une chaîne d'octets sous sa forme binaire<br>(pas de caractère *nul* en fin de chaîne) | char[n] |
| r, R | adresse d'une chaîne d'octets sous sa forme hexadécimale | char[n] |

3. **{f}** indique que l'adresse d'une fonction de type entier de l'application est passée en paramètre. Une et une seule fonction est autorisée dans une requête SQL. Cette fonction sera, suivant les besoins du programmeur, de l'une des formes ci-dessous :

```
• int fonction ()
  {...}
```

```
• int fonction (c_err)
  int c_err;     /* code d'erreur Oracle */
  {...}
```

L'argument **c_err** reçoit le code d'erreur Oracle chargé par OPIDUM lors de l'appel de la fonction.

```
• int fonction (c_err, nb_car);
  int c_err, nb_car;
  {...}
```

**nb_car** est utilisé seulement avec **&na**, et reçoit la longueur de l'attribut

retourné.

Dans le cas où l'on désire stopper la récupération des tuples, cette fonction doit retourner une valeur strictement négative, l'exécution de SELECT est alors abandonnée.

Les trois types de paramètres décrits précédemment, peuvent être utilisés en nombre variable et indépendamment les uns des autres (excepté dans le cas cité dans le paragraphe 2.1.2 Exemple 3). Toutefois, à chaque descripteur inséré dans la requête, doit correspondre un paramètre. Les paramètres doivent être positionnés dans le même ordre que les descripteurs, qui e ux-mêmes sont placés de manière logique.

### *2.1.2 Exemples d'utilisation*

Ces exemples mettent en évidence les fonctionnalités de la fonction sql.

**Exemple 1 : substitution de variables en entrée.**

```
...
char nom_table[31];
char nom[21];
int age;
strcpy (nom_table, "demopidum");
strcpy (nom, "DUPONT");
age = 40;
sql("select * from %S where nom = %s and age < %i ",
                  nom_table,     nom,          age);
...
```

La commande sql à exécuter, contient le descripteur "%S", qui précise à OPIDUM qu'un paramètre en entrée de type chaîne de caractères, doit être substitué avant l'exécution de la requête par Oracle. La nom de la table contenu dans la variable **nom_table**, se substituera au "%S". Elle contient aussi les descripteurs "%s" et "%i", qui précisent à OPIDUM que respectivement un paramètre en entrée de type chaîne de caractères (contenant la valeur de l'attribut *nom*) et un paramètre de type entier, doivent aussi être substitués. Les valeurs des variables **nom** et **age**, se substitueront respectivement à "%s" et "%i". La requête SQL envoyée à Oracle est donc :

```
"select * from demopidum where nom = 'DUPONT' and age < 40"
```

L'exécution de cette commande a les trois effets suivants :

- Le résultat de la requête s'affiche automatiquement sur l'écran, dans un format standard.
- La valeur rendue par la fonction sql est soit 0, s'il y a eu un problème, soit 1, si tout s'est bien déroulé.
- La variable globale **sqlstatus**, définie dans la librairie OPIDUM, de la manière suivante :

```
typedef struct {
    tab_colstatus *colstatus ;/* référence à un  tableau de colstatus     */
    short typeft ;            /* type de la fonction sql                  */
    unsigned short errnum ;   /* code d'erreur oracle                     */
    short int_err ;           /* code d'erreur interne                    */
    short rows ;              /* nombre de tuples (non valide pour select)*/
    short columns ;           /* nombre de colonnes (pour select seult)   */
} t_sqlstatus ;

t_sqlstatus sqlstatus;
```

rend des informations sur le déroulement de la requête. Ici, la variable **sqlstatus.columns** contient le nombre d'attributs de la table *demopidum.*

**Exemple 2 : récupération de valeurs dans des variables.**

```
...
char nom[21];
int age;
int imprimer(c_err)
      int c_err;
      {
      printf("nom : %s age : %d\n",nom,age);
      return(1);  /* pour continuer la récupération des tuples */
      }
sql("select {f} nom   &20s, age &i from demopidum",
           imprimer, nom,      &age);
...
```

La commande sql à exécuter contient plusieurs descripteurs :

{f} qui indique que le paramètre **imprimer** est l'adresse d'une fonction, de type entier,

&20s qui précise que le paramètre **nom** est l'adresse d'une chaîne de caractères d'une longueur maximum de 20 octets, destinée à recevoir la valeur de l'attribut *nom* selectionné dans la table *demopidum*,

&i qui indique que le dernier paramètre **&age**, est l'adresse d'une variable entière, destinée à recevoir la valeur de l'attribut *age* sélectionné dans la table *demopidum.*

L'exécution de cette commande a pour effet :

- Pas d'affichage direct sur l'écran, (fait par la fonction imprimer).
- La valeur rendue est 0 ou 1.
- Pour chaque tuple trouvé, les variables **nom** et **age** reçoivent respectivement les valeurs des attributs *nom* et *age* sélectionnés dans la table *demopidum.*
- La fonction **imprimer**, décrite par l'utilisateur, est appelée à la restitution de chaque tuple, après le chargement des variables de sortie. **c_err** est le code d'erreur Oracle.
- Comme dans l'exemple 1, la structure globale **sqlstatus** rend des informations sur le déroulement de la commande.

Si le résultat de la requête est un tuple unique, il est possible de récupérer sa valeur de la manière suivante :

```
sql("select age &i from demopidum where nom = %s",
                &age,                          nom);
...
```

Pour le tuple unique trouvé, la variable **age** reçoit la valeur de l'attribut *age* de la table *demopidum.*

### Exemple 3 : récupération des noms de colonnes d'une table

```
...
char nom_colonne[31];
int imprimer(c_err, lg_attribut)
      int c_err, lg_attribut;
      {
      printf("%s (%d)\n",nom_colonne,lg_attribut);
      return(1);
      }
sql("select * {f}      &30a from demopidum",
             imprimer,nom_colonne);
...
```

Dans le cas de la récupération des noms de colonnes d'une table, la requête doit toujours être de la forme :

**"select * {f} &na from table"**

L'exécution de la commande de l'exemple a pour effet :

- Pas d'affichage direct sur l'écran.
- La valeur rendue par sql est 0 ou 1.
- La fonction **imprimer** est appelée à la restitution de chaque nom d'attribut de la table *demopidum*, après le chargement de la variable de sortie **nom_colonne**. Elle rend à la fois, le code d'erreur Oracle et la longueur de l'attribut courant.
- Le type Oracle de l'attribut est donné dans la variable **(*sqlstatus.colstatus)[i]**, dont le type est décrit ci-dessous, **i** est le numéro de la colonne.

```
typedef struct {
    short errnum ;            /* code d'erreur Oracle sur colonne */
    unsigned short length ;   /* longueur de l'attribut           */
    short check ;             /* non utilisé                      */
    short type ;              /* type d'attribut                  */
} t_colstatus ;

typedef t_colstatus tab_colstatus[MAX_COLS + 1];
```

Dans le cas de requêtes SQL de mise à jour telles que INSERT, UPDATE, DELETE, ..., les paramètres de sortie (descripteur "&nt") ne sont pas utilisés, car ces requêtes ne

retournent pas de valeurs d'attributs.

### 2.2 La fonction sqli

L'interface de la fonction sqli est la suivante :

```
int sqli (requête_sql,paramètre)
char *requête_sql;
int *paramètre[];
```

La fonction sqli est une variante de sql avec un nombre fixe de paramètres (deux au total). Cette fonction permet notamment l'interfaçage avec le langage Pascal où un nombre variable de paramètres n'est pas possible. Le premier paramètre d'appel est une requête SQL contenant des descripteurs de paramètres. Le second paramètre est l'adresse d'un tableau dont la dimension est fixée par le programme appelant, et où **paramètre[i-1]** contient l'adresse du $i^{ème}$ paramètre décrit dans la requête SQL.

### 2.3 La fonction sqlo

L'interface de la fonction sqlo est la suivante :

```
int sqlo (requête_sql)
char *requête_sql;
```

sqlo est une variante de sql pour les requêtes sans descripteur de paramètres. Sa raison d'être est l'efficacité car aucune substitution n'est à faire et son traitement est ainsi optimisé.

### 2.4 Les fonctions de base

#### *2.4.1 Connection*

La fonction connection permet de se connecter, ou de se déconnecter d'Oracle.

Elle est définie comme suit :

**int connection (1/0, user_name[@host], password)**
**char *user_name, *host, *password;**

et elle rend les valeurs 0 ou 1. Le paramètre optionnel **host** doit être renseigné dans le cas d'une utilisation d'OPIDUM en réparti. **host** est alors le nom de la machine sur laquelle le SGBD Oracle est implanté.

Connection retourne la valeur 0 si une erreur a été détectée. Dans ce cas, s'il s'agit d'une erreur Oracle, la variable **sqlstatus.errnum** contient le code d'erreur Oracle différent de 0, sinon s'il s'agit d'une erreur interne à OPIDUM, la valeur de **sqlstatus.errnum** est alors nulle, et **sqlstatus.int_err** contient le code d'erreur correspondant.

**Exemples :**

```
connection(1,"test","test");  /* connexion en local     */
...
connection(0);                /* déconnexion            */

connection(1,"scott@idefix","tiger"); /* connexion en réparti */
...
connection(0);                        /* déconnexion          */
```

**Important :** connection(1, user_name, password) place Oracle en AUTOCOMMIT OFF.

*2.4.2 Les fonctions de commit*

Deux fonctions permettent le contrôle du COMMIT :

**int autocommit (1/0)**
**int commit (1/0)**

et elles rendent les valeurs 0 ou 1.

La variable **sqlstatus.errnum** contient le code d'erreur de la fonction HLI activée.

| Appel | Action |
|---|---|
| autocommit(1) | COMMIT ON |
| autocommit(0) | COMMIT OFF |
| commit(1) | COMMIT |
| commit(0) | ROLLBACK |

### *2.4.3 sqlb, sqls, sql_fetch, sql_abort*

Grâce à ces fonctions, le programmeur peut, lors d'une requête SELECT, gérer lui-même la récupération des tuples sélectionnés.

**Rappel :** La fonction sql, décrite précédemment, provoque au contraire la restitution automatique des tuples.

Les deux phases mises en jeu sont :

- la compilation et l'exécution de la requête, après substitution éventuelle des paramètres en entrée : c'est le rôle de la fonction **sqlb** (pas de variable du programme à substituer dans la requête) ou **sqls** (variables du programme à substituer dans la requête),
- la récupération des tuples sélectionnés : c'est le rôle de la fonction **sql_fetch**.

Les fonctions qui réalisent la compilation et l'exécution de la requête, sont décritent comme suit.

L'interface de la fonction sqlb est la suivante :

```
int sqlb (requête_sql, adr_header)
char *requête_sql;
tab_header **adr_header;
```

Le premier paramètre d'appel est une requête SELECT, sans descripteur. Le paramètre **adr_header** est l'adresse d'un pointeur sur un tableau de type tab_header, exploitable à partir de 1 et défini dans le fichier opidum.h de la façon suivante :

```
typedef struct {
            short type ;  /* type Oracle de l'attribut */
            char *name ;  /* nom de l'attribut         */
} t_header;

typedef t_header tab_header [MAX_COLS + 1];
```

**MAX_COLS** représente le nombre maximum d'attributs pouvant être sélectionnés.
Dans le programme d'application, le programmeur définit, par exemple, la variable **header** de type (tab_header *), et lorsque l'adresse de **header** est donnée comme paramètre dans l'appel de la fonction sqlb, les variables **(*header)[i].type** et **(*header)[i].name** contiennent respectivement le type Oracle et le nom du ième attribut sélectionné dans le SELECT.
La variable **(*sqlstatus.colstastus)[i].length** retourne la taille définie dans Oracle du ième attribut.

sqlb retourne 0 si une erreur a été détectée, sinon elle rend un identificateur de la requête. En standard, le nombre maximum de requêtes actives simultanément est 8.

**Remarque** : Les primitives de **connexion** (connection(1,...)) et **déconnexion** (connection(0)) assurent l'allocation et la désallocation d'un nombre de curseurs pré-établi, au total huit, pendant la session, pour permettre la programmation de requêtes imbriquées.

**Exemple :**

```
...
tab_header *header;
int req_idf;
req_idf = sqlb("select nom, age from demopidum",&header);
...
```

On obtient :

(*header)[1].type = 1 (*header)[1].name = "nom"
(*header)[2].type = 2 (*header)[2].name = "age"

(*sqlstatus.colstatus)[1].length = 20
(*sqlstatus.colstatus)[2].length = 22

La fonction sqls est une extension de la fonction sqlb, elle accepte en plus des

paramètre en entrée à substituer dans la requête. La requête SELECT, en paramètre, peut donc contenir le descripteur de type "%t".

L'interface de la fonction sqls est la suivante :

**int sqls (requête_sql, adr_header [, paramètres] ... )**
**char *requête_sql;**
**tab_header **adr_header;**

**Exemple :**

```
...
tab_header *header;
int req_idf;
int age = 40;
req_idf = sqls("select nom, prenom from demopidum where age < %i",
                &header,age);
...
```

La fonction **sql_fetch**, décrite comme suit, réalise la récupération des tuples sélectionnés par la fonction sqlb ou sqls.

L'interface de la fonction sql_fetch est la suivante :

**int sql_fetch (req_idf, adr_tuple)**
**int req_idf;**
**tab_tuple **adr_tuple;**

Le premier paramètre d'appel est l'identificateur de la requête retourné par sqlb ou sqls, dont on veut récupérer les tuples sélectionnés.
Le paramètre **adr_tuple** est l'adresse d'un pointeur sur un tableau de type tab_tuple , exploitable à partir de 1 et défini dans le fichier opidum.h de la manière suivante :

```
typedef struct {
            int length ;     /* longueur du buffer alloué                  */
            char *buf ;      /* toutes les valeurs sont converties en string*/
} t_tuple;

typedef t_tuple tab_tuple [MAX_COLS + 1];
```

Lorsque l'adresse d'une variable **tuple** de type (tab_tuple *), est passée en paramètre dans l'appel de la fonction sql_fetch, les variables **(*tuple)[i].length** et **(*tuple)[i].buf** contiennent respectivement la longueur du buffer alloué, et la valeur du ième attribut sélectionné.

**Important :** Toutes les valeurs d'attribut, quel que soit leur type, sont converties en chaîne de caractères, le programme d'application doit, par conséquent, assurer les

conversions en fonction du type retourné dans la structure t_header.

La variable **(*sqlstatus.colstatus)[i].length** donne la longueur en octets de la valeur courante du ième attribut.

sql_fetch retourne l'identificateur de la requête, s'il reste des tuples, sinon elle rend la valeur END_FETCH. Lorsqu'il n' y a plus de tuple à restituer, elle libère l'espace alloué pour les tableaux de structures t_header et t_tuple. En cas d'erreur, la valeur retournée par sql_fetch est 0.

**Exemple :**

```
 ...
tab_header *header;
tab_tuple *tuple;
int req_idf, ret_fetch;
int age = 40;

req_idf = sqls("select nom, age from demopidum where age < %i",&header,age);

while ((ret_fetch = sql_fetch(req_idf, &tuple)) != END_FETCH) {

            if (ret_fetch == 0) {
                        impr_erreur();
                        break;
            }
            traiter_tuple(tuple,header);
}
 ...
impr_err()
{...}

void traiter_tuple(t,h)
tab_tuple *t;
tab_header *h;
{
            int i;
            for (i = 1; i <= sqlstatus.columns; i++) {

                        switch((*h)[i].type) {

                        case STRING  :
                        printf("%s",(*t)[i].buf);

                        case NUMERIC :
                        printf("%10g",atof((*t)[i].buf));
                        }
            }

}
 ...
```

**Remarque :** La fonction sql est construite à partir des fonctions de "bas niveau" sqlb et sql_fetch.

La fonction **sql_abort** permet, en cas d'arrêt de la demande de tuples, par le programmeur (avant la fin de tuples), de libérer l'espace allouée pour les structures tab_header et tab_tuple.

Son interface est la suivante :

```
sql_abort (req_idf)
int req_idf;
```

Le paramètre **req_idf** est l'identificateur de la requête, retourné par sqlb ou sqls.

### *2.4.4 Options*

La fonction Options reproduit les fonctionnalités de la fonction HLI, OOPT (cursor, rollback, waitop). Le paramètre rollback (qui prend les valeurs 0 ou 2) permet de programmer l'ordre de rollback par ligne, en cas d'erreurs non fatales survenant à la mise à jour. Le paramètre waitopt spécifie, en cas de ressources indisponibles si le programme attend indéfiniment ( waitopt vaut 0) ou si une erreur est signalée (waitopt vaut 4 ). Pour des compléments, il faut se reporter au manuel Oracle intitulé PRO*SQL User Guide.

L'interface de la fonction options est la suivante :

```
int options (optn1, optn2)
int optn1, optn2;
```

Les paramètres optn1 et optn2 correspondent respectivement aux paramètres roolback et waitopt de la fonction OOPT de HLI.
Options retourne 0 en cas d'erreur ou 1 si l'ordre s'est exécuté correctement.

### *2.4.5 La fonction print_set*

Par défaut, des messages (d'états ou d'erreurs) sont affichés à l'écran. Si le programmeur désire supprimer l'affichage de ces messages, il doit faire un appel à la fonction **print_set**.

L'interface de la fonction print_set est la suivante :

```
int print_set (1/0)
```

et elle rend toujours la valeur 1.

| **Appel** | **Action** |
|---|---|
| print_set(1) | affichage des messages OPIDUM |
| print_set(0) | pas d'affichage des messages OPIDUM |

### *2.4.6 Les fonctions de transformation pour le type binaire*

Le type binaire ou RAW est stocké dans le SGBD, comme le type CHAR, sous forme d'une chaîne d'octets de longueur variable. Le programmeur doit savoir que :

- Une donnée de type RAW est rendue par OPIDUM au programme utilisateur sous la forme hexadécimal. Par contre, une donnée de type LONG RAW est retournée sous sa forme binaire.
- Une insertion ou une modification d'une donnée de type RAW ou LONG RAW dans un SGBD doit se faire en spécifiant sa valeur sous sa forme hexadécimale.

Les transformations de format sont à la charge du programmeur, qui dispose des deux fonctions **btoh** et **htob**.

L'interface de la fonction btoh est la suivante :

```
char *btoh (strbin, longueur, strhex)
char *strbin, *strhex;
int longueur;
```

Le paramètre **strbin** est l'adresse de la chaîne binaire à transformer, le paramètre **longueur** spécifie sa taille. Le dernier paramètre **strhex** est l'adresse de la chaîne hexadécimale résultante.

L'interface de la fonction htob est la suivante :

```
char *htob (strhex, strbin)
char *strhex, *strbin;
```

Le paramètre **strhex** est l'adresse de la chaîne hexadécimale à convertir. Le second paramètre **strbin** est l'adresse de la chaîne binaire, destinée à recevoir le résultat.

## 3. LES DONNEES ORACLE

Chaque colonne d'une table Oracle est spécifiée comme un des types de données suivants :

- CHAR (N)
- NUMBER
- DATE
- LONG
- RAW (N), LONG RAW

Les variables **(*header)[i].type** et **(*sqlstatus.colstatus)[i].type**, décrites précédemment, précisent le type Oracle du ième attribut sélectionné, dans une requête SELECT. La correspondance entre le type Oracle et la valeur de ces variables est donné dans le tableau ci-dessous. La taille en octets définie dans Oracle est donnée à titre

indicatif pour le formatage des résultats. Elle est contenue dans la variable **(*sqlstatus.colstatus)[i].length**, au retour des fonctions sqlb et sqls (se reporter à 5.3).

| Type Oracle | Valeur | Taille en octets définie dans Oracle |
|---|---|---|
| CHAR | STRING | N <= 240 |
| NUMBER | NUMERIC | 22 |
| RAW | | RAW |
| LONG | LONG | 0 |
| LONG RAW | LONGRAW | 0 |
| DATE | DATE | 8 |
| ROWID | ROWID | 14 |

## 4. LES CODES ERREURS

Les fonctions OPIDUM retourne la valeur 0 si une erreur a été détectée. S'il s'agit d'une erreur Oracle, la variable **sqlstatus.errnum** contient le code d'erreur Oracle (Se reporter à Oracle Error Messages and Codes Manual), sinon s'il s'agit d'une erreur interne à OPIDUM, **sqlstatus.errnum** est nulle, et **sqlstatus.int_err** contient une valeur entière comprise entre 1 et 11.

Les codes d'erreur interne à OPIDUM sont les suivants :

| Code | Signification |
|---|---|
| 1 | erreur d'allocation mémoire |
| 2 | erreur de syntaxe en SQL |
| 3 | trop de colonnes (nombre de colonnes limité à MAX_COLS) |
| 4 | adresse nulle passée en paramètre |
| 5 | {f} utilisé dans une commande autre que SELECT |
| 6 | descripteur {f} incomplet |
| 7 | non utilisé |
| 8 | pour un LONG, longueur supérieure à MAX_LNG |
| 9 | trop de curseurs utilisés (nombre limité à CURSOR_MAX) |
| 10 | RPC indisponible |
| 11 | erreur interne RPC |

La détection d'erreurs d'exécution peut être programmée de la manière suivante :

```
if ((fct_opidum (...) == 0) && (sqlstatus.errnum != 0))
        printf("Erreur Oracle : %d\n", sqlstatus.errnum);
if ((fct_opidum (...) == 0) && (sqlstatus.errnum == 0))
        printf("Erreur interne d'OPIDUM : %d\n", sqlstatus.int_err);
```

# 5. RECAPITULATIF

## 5.1 Les fonctions

```
int sql (requête_sql [, paramètres] ... )
char *requête_sql;

int sqli (requête_sql, paramètres)
char *requête_sql;
int *paramètres[];

int sqlo (requête_sql)
char *requête_sql;

int connection (1/0, user_name, password)
char *user_name, *password;

int autocommit (1/0)
int commit (1/0)

int sqlb (requête_sql, adr_header)
char *requête_sql;
tab_header **adr_header;

int sqls (requête_sql, adr_header [, paramètres] ... )
char *requête_sql;
tab_header **adr_header;

int sql_fetch (req_idf, adr_tuple)
int req_idf;
tab_tuple **adr_tuple;

int sql_abort (req_idf)
int req_idf;

int options (optn1, optn2)
int optn1, optn2;

int print_set (1/0)

char *btoh (strbin, longueur, strhex)
char *strbin, *strhex;
int longueur;

char *htob (strhex, strbin)
char *strhex, *strbin;
```

## 5.2 Les types

### *5.2.1 Le type t_ora_lng*

```
typedef struct {
     char *adresse ;
     int longueur ;
} t_ora_lng;
```

Le type t_ora_lng sert à manipuler les données de type LONG et LONG RAW. Le champ longueur est toujours initialisé par l'application avant l'appel d'une fonction OPIDUM. Lors d'une mise à jour de la base Oracle, **longueur** indique le nombre d'octets à charger à partir de **adresse**. Dans le cas d'une interrogation de la base, **longueur** représente le nombre maximum d'octets à lire (et à écrire ) à partir de **adresse** (la taille réelle de l'attribut est donnée par la variable **(*sqlstatus.colstatus)[i].length**), au retour des fonctions sql, sql, sqlo et sql_fetch. La valeur de longueur est limitée à 65535.

### *5.2.2 Le type tab_header*

```
typedef struct {
     short type ;      /* type Oracle de l'attribut */
     char *name ;      /* nom de l'attribut         */
} t_header;

typedef t_header tab_header [MAX_COLS + 1];
```

Le type tab_header est utile exclusivement lorsque la requête SELECT est passée en paramètre des fonctions sqlb et sqls. **MAX_COLS** (sa valeur est fixée à 40, en standard) représente le nombre maximum d'attributs pouvant être sélectionnés.

### *5.2.3 Le type tab_tuple*

```
typedef struct {
     int length ;     /* longueur du buffer alloué                    */
     char *buf ;      /* toutes les valeurs sont converties en string */
} t_tuple;

typedef t_tuple tab_tuple [MAX_COLS + 1];
```

Le type tab_tuple n'a de sens que pour les requêtes SELECT, et est utilisé pour la récupération des tuples à la demande avec les fonctions sqlb ou sqls, et sql_fetch.

### 5.3 Les variable globale sqlstatus

```
typedef struct {
    short errnum ;              /* code d'erreur Oracle sur colonne */
    unsigned short length ;     /* longueur de l'attribut           */
    short check ;               /* non utilisé                      */
    short type ;                /* type d'attribut                  */
} t_colstatus ;

typedef t_colstatus tab_colstatus[MAX_COLS + 1];

typedef struct {
    tab_colstatus *colstatus ;/* référence à un  tableau de colstatus     */
    short typeft ;              /* type de la fonction sql                 */
    unsigned short errnum ;     /* code d'erreur oracle                    */
    short int_err ;             /* code d'erreur interne                   */
    short rows ;                /* nombre de tuples (non valide pour select)*/
    short columns ;             /* nombre de colonnes (pour select seult)   */
} t_sqlstatus ;

t_sqlstatus sqlstatus;
```

La variable sqlstatus est propre à une requête. Dans le cas de requêtes imbriquées, il existe autant d'exemplaires de la structure t_sqlstatus que d'identificateurs de requête utilisés.

Le champ **typeft** est retourné quel que soit la requête SQL passée en paramètre d'une fonction OPIDUM, et contient son type Oracle.

Le champ **rows** donne, par exemple, le nombre de lignes d'une table Oracle mises à jour, lors d'une requête UPDATE. Sa valeur n'est pas significative lors d'une requête SELECT.

Les champs **columns** et **colstatus** sont exploitables exclusivement quand la requête exécutée est un SELECT. Dans ce cas, le champ **columns** spécifie le nombre d'attributs sélectionnés.
Il existe une structure **t_colstatus** par attribut sélectionné. La variable **(*sqlstatus.colstatus)[i].type** désigne le type Oracle du ième attribut sélectionné. La variable **(sqlstatus.colstatus)[i].length** donne :

- dans le cas des fonctions sqlb et sqls, la taille définie dans Oracle du ième attribut,
- dans le cas des fonctions sql, sqlo, et sql_fetch, la longueur en octets de la valeur courante du ième attribut.

La variable **(*sqlstatus.colstatus)[i].errnum** rend le code d'erreur Oracle éventuel sur le ième attribut.

Seuls les champs **errnum** et **int_err** de la structure t_sqlstatus sont exploitables quelle que soit la fonction OPIDUM appelée, ils contiennent respectivement le code d'erreur

d'Oracle et le code d'erreur interne à OPIDUM.

## 6. MODE D'EMPLOI

### 6.1 En local

Le logiciel OPIDUM est composé la librairie **libopidum.a** qui contient tous les modules objets d'OPIDUM, et d'un fichier include **opidum.h**. Un programme de démonstration (**demopidum** et **demopidum.c**), et un modèle de Makefile (**Makedemo**) sont à la disposition du programmeur.

La figure qui suit décrit l'architecture du serveur OPIDUM en local.

### 6.2 En réparti

Sur le site client, OPIDUM comprend la librairie **libopidumcl.a** qui contient tous les modules objets d'OPIDUM, et le fichier include **opidum.h**. Un modèle de Makefile (**Makedemorep**) est à la disposition du programmeur.
Le site serveur dispose de deux programmes, **daemon_sv** qui doit être lancé préalablement et **opidum_sv**, qui joue le rôle de serveur qui est lancé à la connection de l'application.

La figure qui suit décrit l'architecture du serveur OPIDUM en réparti.

Le fonctionnement du logiciel OPIDUM en mode distribué est représenté par la figure suivante :

## SOMMAIRE


1. PRESENTATION . . . . . 106
   1.1 Caractéristiques . . . . . 106
   1.2 Principe . . . . . 107
2. LES FONCTIONS . . . . . 109
   2.1 La fonction sql . . . . . 109
   2.2 La fonction sqli . . . . . 115
   2.3 La fonction sqlo . . . . . 115
   2.4 Les fonctions de base . . . . . 115
3. LES DONNEES ORACLE . . . . . 122
4. LES CODES ERREURS . . . . . 123
5. RECAPITULATIF . . . . . 124
   5.1 Les fonctions . . . . . 124
   5.2 Les types . . . . . 125
   5.3 Les variable globale sqlstatus . . . . . 126
6. MODE D'EMPLOI . . . . . 127
   6.1 En local . . . . . 127
   6.2 En réparti . . . . . 128

# Bibliographie

[Afnor 87] AFNOR.
*Vocabulaire de la documentation, 2ème édition.*
Les Dossiers de la Normalisation - Paris - 1987.

[Aidel 88] AIDEL.
*SUPERDOC, Progiciel de Gestion de Base de Données Textuelles.*
AIDEL - Goncelin - 1988.

[Alhafez 87] ALHAFEZ Nizar.
*Etude et réalisation d'un mécanisme d'appel de procédure à distance.*
DEA Informatique - Université de Grenoble - 1987.

[Berkeley 86] BERKELEY Software Distribution.
*Unix User's Reference Manual - Virtual VAX-11 Version.*
University of California - Berkeley - 1986.

[Bourne 85] BOURNE Steve.
*Le système Unix.*
InterEditions - Paris - 1985.

[Budd 88] A.BUDD Timothy.
*BIB , A program for formatting bibliographies.*
Department of Computer Science, The University of Arizona - Tuscon Arizona.

[Calixte 85] CALIXTE Jacqueline - MORIN J-Claude.
*Management d'un service d'information documentaire.*
Les Editions d'Organisation - Paris - 1985.

[Cassagne 89] CASSAGNE Bernard.
*Introduction au langage C.*
Laboratoire de Génie Informatique - IMAG - Grenoble - 1989.

[CF 84] CHRISTODOULAKIS S. - FALOUTSOS Ch.
*Signature Files : A Access Method for documents and its Analytical Performance Evaluation.*
ACM Transactions on Office Information Systems, Vol 2, n°.4 - 1984.

[Cooper 83] COOPER E.C.
*Writing Distributed Programs with Courrier.*
University of California - Berkeley - 1982.

[Delobel 82] DELOBEL Claude - ADIBA Michel.
*Bases de données et systèmes relationnels.*
DUNOD Informatique - Poitiers - 1982.

[Deweze 83] DEWEZE A.
*L'accés en ligne aux bases documentaires.*
Masson - Paris - 1983.

[Fal 86] FALOUTSOS Christos.
*Integrated Access Methods for Messages using Signatures Files.*
Methods and Tools for Office Systems - IFIP - 1986.

[Houssain 88] HAJ HOUSSAIN Samer.
*DOSIS : un serveur OSI pour l'ouverture ses systèmes distribués au monde extérieur.*
Thèse de Docteur INPG - 1988.

[Jimenez 89] JIMENEZ GUARIN Claudia.
*Opérations d'accès par le contenu à une base de documents textuels . Application à un environnement de bureau.*
Thèse de Docteur INPG - 1989.

[Latex 86] LAMPORT Leslie.
*LATEX, A document Preparation System.*
Wesley Publishing Company - Addison - 1986.

[Lenne 89] LENNE Christian.
*EMIR : un outil de gestion de structures arborescentes attribuées.*
Centre de recherche Bull - Grenoble - 1989.

[Lesk 88] LESK M.E.
*Some Applications of Inverted Indexes on the Unix System.*
AT&T Laboratories - Murray Hill, New Jersey - 1988.

[Mondain 87] MONDAIN-MONVAL Pierre.
*Conception et vérification d'un service d'appel de procédure à distance, dans les réseaux hétérogènes.*
Thèse de docteur 3ème cycle - Université de Toulouse - 1987.

[Nelson 81] NELSON B.J.
*Remote Procedure Call.*
PHD Thesis - Carnegie Mellon University - 1981.

[Oracle 87] SPIX Oracle.
*DATABASE Administrator's Guide - Vol 1-2.*
*PRO* COMPILERS Programmer's Guide.*
*SQL* PLUS User's Guide - Vol 1-2.*
BULL - Val de Reuil - 1987.

[Paire 89] PAIRE Eric - LAFORGUE Pierre.
Le réseau hétérogène multi-média de l'I.M.A.G.
I.M.A.G. - Grenoble - 1989.

[Ritchie 84] W.KERNIGHAN Brian - M.RITCHIE Dennis.
*Le langage C.*
Masson - Paris - 1984.

[RPC 87] SUN.
*Remote Procedure Call (RPC).*
*eXternal Data Representation (XDR).*
*Network File System (NFS).*
Sun - 1987.

[Silvestre 88] SILVESTRE Danièle.
*Réalisation d'un serveur de communication OSI dans un réseau local Ethernet.*
Mémoire CNAM - Grenoble - 1988.

[Soupe 88] BENNETT M. - BERARD P. - LENNE C. - LOPEZ M.
*SOUPE : une interface de haut niveau pour Oracle.*
Centre de recherche Bull - Grenoble - 1988.

[Spix 86] BULL.
*SPIX System - Programmer Reference Manual - First Edition.*
*ADMINISTRATOR Reference Manual.*
BULL S.A. 1986.

[SQL 88] COLLET Christine - FAUVET M-Christine.
*SQL/UFI, Reference Guide.*
LGI - Grenoble - 1988.

[SystemV 87] UNIX System V - AT&T.
*Manuel de référence du programmeur.*
Masson - Paris - 1987.

[Tuthill 88] TUTHILL Bill.
*Refer , A Bibliography System.*
Computing Services, University of California - Berkeley - 1988.

[Unix 88] J.BACH Maurice.
*The design of the Unix Operating system.*
1988.

[Slype 77] VAN SLYPE G.
*Conception et gestion des systèmes documentaires.*
Les Editions d'Organisation - Paris - 1985.

[Voboril 89] VOBORIL Dominique.
*Conception et réalisation d'un service réparti sur réseau pour la consultation de bases d'informations.*
Mémoire CNAM - Grenoble - 1989.