NFE 107
-
Ubanisation des Systèmes d'Information

RPC / MOM
Comparaison

Veille technologique réalisée par
Dorian AROD

Centre d'Enseignement CNAM de Grenoble
CUEFA
Domaine Universitaire
701, rue de la piscine – BP 81
38402 Saint Martin d'Hères Cedex

Table des matières

1) Présentation de RPC

RPC (Remote Procedure Call) est un protocole permettant de faire des appels de procédures sur un ordinateur distant à l'aide d'un serveur d'application. Ce protocole est utilisé dans le modèle client-serveur et permet de gérer les différents messages entre ces entités.

On peut utiliser les RPC pour toutes sortes d'applications distribuées, par exemple l'utilisation d'un ordinateur très puissant pour des calculs intensifs (décryptage de données, calculs numériques, ...). Cet ordinateur sera donc le serveur. Un autre ordinateur sera le client et appelera la procédure distante pour commander des calculs au serveur et récupérer le résultat.

L'idée de RPC (Remote Procedure Call) n'est pas nouvelle et date environ de 1976, comme le décit la RFC 707 (http://tools.ietf.org/html/rfc707). Le premier usage commerciale d'RPC, sous le nom de « Courier », a été fait par la société Xerox en 1981.

L'implementation de RPC la plus connue et la plus utilisée sous Unix est Sun RPC, aujourd'hui connu sous le nom de ONC RPC, notamment utilisé pour le système NFS (Network File System, développé en 1984) de SUN. Aujourd'hui, ONC RPC est utilisé sur de nombreuses plateformes.

Une autre implémentation de RPC sous Unix est NCS (Network Computing System), d'Apollo Computer. NCS a été utilisé pour la création de DCE/RPC. Plus tard, Microsoft adopta DCE/RPC en tant que base à MSRPC, le méchanisme RPC propre à Microsoft. Par la suite, Microsoft s'est basé sur MSRPC pour implémenter DCOM.

On peut suivre l'évolution du protocole RPC à travers les différentes RFC (Request For Comments) qui le définissent :

- RFC 707 (idées de bases) : janvier 1976
- RFC 1050 (version 1) : avril 1988
- RFC 1087 (version 2) : juin 1988
- RFC 1831 (mise à jour version 2) : août 1995

2) Présentation de MOM

Le terme Message-Oriented Middleware (MOM) désigne une famille de logiciels qui permettent l'échange de messages entre les applications présentes sur un réseau informatique. Les MOM font partie des éléments techniques de base des architectures informatiques. Ils permettent une forme de couplage faible entre applications.

Les Middleware Orientés Messages sont des systèmes fournissant à leurs clients un service de Peer to peer.

La définition standard du terme de Peer to Peer désigne sur Internet un échange de données ou de fichiers d'un machine à une autre (en Français d'égal à égal), un MOM est un logiciel serveur dont le rôle est de fédérer l'envoi et la réception de ces messages entre les différents types d'applications.

La communication de deux applications via un Message Oriented Middleware est complètement asynchrone, c'est à dire que l'émetteur et le destinataire n'ont pas besoin d'être connectés simultanément lorsqu'ils communiquent.

La communication n'est synchrone qu'entre l'emetteur et le MOM d'une part, et le MOM et le destinataire d'autre part.

3) Caractéristiques et comparaisons

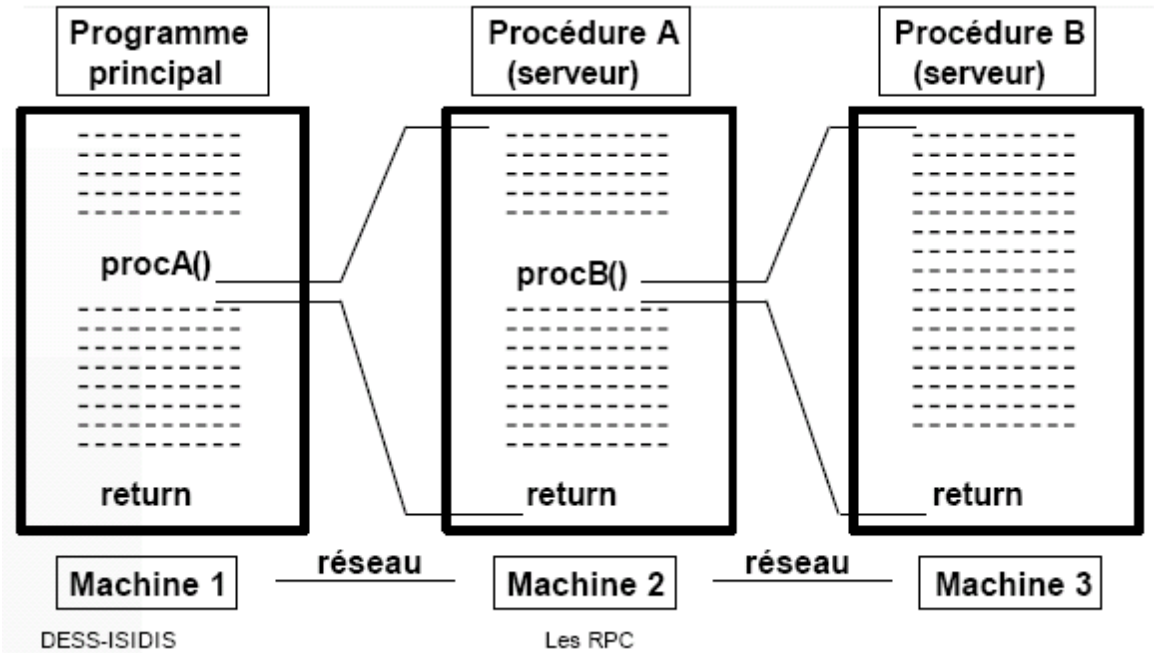
Caractéristique	MOM	RPC
Métaphore	Courier	Téléphone (sans répondeur)
Relation temporelle entre client et serveur	**Asynchrone**	**Synchrone**
Nature du dialogue	File d'attente	Requête - Réponse
Etat opérationnel du serveur	Pas nécessaire	Obligatoire
Equilibrage de charge	Politique d'extraction des messages (système de priorités)	Au moyen d'un moniteur transactionnel
Support des transactions	Dépend du produit	Dépend du produit (nécessité d'un RPC transactionnel)
Filtrage des messages	Possible	Non
Performances	Lent en cas de sécurisation des messages par écriture sur disque	Plus efficace que MOM car pas de sauvegarde

On voit donc grâce à ce tableau que la différence majeure entre RPC et MOM est que l'un, RPC, fonctionne en mode synchrone, tandis que l'autre, MOM, fonctionne en mode asynchrone.

Un fonctionnement synchrone définit quelque chose qui se passe en même temps. Ici, dans le cas de RPC, le client et le serveur doivent être connecté en même temps pour que l'appel de procédure à distance puisse fonctionner.

Dans le cas de MOM, il s'agit d'un fonctionnement asynchrone. Ici, les deux composants qui échangent n'ont pas besoin d'être connectés en même temps grâce au système de file d'attente des MOM.

4) Fonctionnement de RPC



Principe de fonctionnement des RPC

Le système RPC s'efforce de maintenir le plus possible la sémantique habituelle des appels de fonction, autrement dit tout doit être le plus transparent possible pour le programmeur. Pour que cela ressemble à un appel de fonction local, il existe dans le programme client une fonction locale qui a le même nom que la fonction distante et qui, en réalité, appelle d'autres fonctions de la bibliothèque RPC qui prennent en charge les connexions réseaux, le passage des paramètres et le retour des résultats. De même, côté serveur il suffira (à quelques exceptions près) d'écrire une fonction comme on en écrit tous les jours, un processus se chargeant d'attendre les connexions clientes et d'appeler votre fonction avec les bons paramètres. Il se chargera ensuite de renvoyer les résultats. Les fonctions qui prennent en charge les connexions réseaux sont des "stub". Il faut donc écrire un stub client et un stub serveur en plus du programme client et de la fonction distante.

Le travail nécessaire à la construction des stubs client et serveur sera automatisé grâce au programme rpcgen qui produira du code C qu'il suffira alors de compiler. Il ne restera plus qu'à écrire le programme client qui appelle la fonction et en utilise le résultat (par exemple en l'affichant) et la fonction elle-même.

5) Fonctionnement de MOM

Les MOM utilisent des files d'attentes ou queues par lesquelles transitent les messages. Lorsqu'un applicatif envoie un message, il se connecte au broker de messages (courtier de messages) à qui il envoie le message en précisant l'identifiant de la file d'attente. Quand le destinataire du message se connecte à son tour à l'agent de gestion des messages, le message lui est alors délivré lorsqu'il lit la file d'attente en question.

Une file d'attente peut aussi être utilisée pour plusieurs couples d'applicatifs (pas besoin de dédier une file par liaison applicative) puisque les MOM comportent différents critères de sélection de messages lors de la lecture.

Par ailleurs, comme c'est le cas pour une table d'une base de données, les messages peuvent être aussi consultés sans être lus, c'est ce qu'on appele le mode "browse".

1. Définitions relatives aux MOM

Les termes suivants sont utilisés dans les Middleware Orientés Message:

- **Middleware Orienté Message - Store and Forward**
  Un Middleware Orienté Message est un système qui permet de faire communiquer deux applicatifs par son intermédiaire, et ce de manière asynchrone.

  Dans une telle architecture où les messages sont échangés par un intermédiaire logiciel, on parle aussi de Store and Forward: le MOM stocke le message et le route par la suite à son destinataire lorsque celui-ci le demande.

  Le délai entre la phase "Store" et la phase "Forward" ne dépend que de l'application destinataire.

  Ce mode de fonctionnement est à l'opposé des modes connectés de communication entre deux applications.

- **Provider**
  Un Provider dans le domaine des Message Oriented Middleware est un produit, un logiciel, qui fournit des services d'envoi/réception de messages en mode asynchrone.

  Par extension, on parle de Provider JMS pour un Middleware Orienté Message qui implémente la norme JMS définie par Sun.

  On utilise aussi parfois le terme de Broker, mais son utilisation ne se limite pas aux MOM: la plupart des produits d'EAI sont aussi appelés Brokers.

- **Message**
  Le message représente les informations échangées par deux applications via le MOM. Il est constitué de trois parties distinctes:

  - Les données elles mêmes, on parle aussi de charge utile (ou payload)
  - L'entête du message, qui comporte ses caractéristiques techniques (son identifiant, date de dépot ...)
  - Ses propriétés, les caractéristiques fonctionnelles du message, qui sont différentes pour chaque application émettrice
  - Données du message

Les données d'un message sont les données applicatives échangées entre les deux logiciels qui communiquent ensemble.

Certains messages peuvent être sans données, avec seulement une entête et des propriétés. D'autres peuvent contenir des messages textes, ou des données sérialisées comme c'est le cas avec les ObjectMessage de JMS.

Dans tous les cas, le rôle du MOM se borne à véhiculer ces données, en aucun cas il ne doit les modifier ou les altérer.

- Entête de message
  Dans l'entête du message, on trouve: la date de dépot, le nom de la file d'attente destinataire, l'identifiant du message, et éventuellement: le nom du queue manager destinataire, l'identifiant de corrélation, la date de disponibilité, la date d'expiration, ...
- Propriétés des messages
  Les propriétés d'un message sont un ensemble de (clés, valeurs) de différents types (entier, chaine, booléen ...) qui apportent des précisions sur le message.

- **File d'attente (En Anglais "queue")**
  Une file d'attente est un espace de stockage pour les messages, limité ou non.
- **Topic (En Français "sujet")**
  Un Topic est une file d'attente particulière qui est destinée à recevoir les messages utilisés dans le mode Publish/Subscribe.
- **Gestionnaire de files d'attentes (Queue manager)**
  Un queue manager est un ensemble de files d'attentes formant un tout cohérent et géré par une ou plusieurs instances d'un MOM. Le queue manager est l'équivalent dans le monde des MOM d'une base de données dans le monde des SGBD.
- **Canal (Channel)**
  On appelle canal un lien entre deux queue manager qui communiquent et s'envoient des messages.
- **Noeud**
  Lors de la communication entre deux gestionnaires de files d'attente par le biais d'un canal, on parle de communication entre deux noeuds.
- **Routage**
  Quand un message qui est destiné à un MOM situé par exemple à New York est envoyé via le noeud du MOM situé à Paris, on parle de routage.

  Le message est alors transféré par le Middleware Orienté Message vers le noeud Américain de façon transparente pour l'applicatif émetteur.

  Généralement, un message devant être routé sur un autre noeud possède des caractéristiques spéciales dans son entête permettant au MOM de savoir qu'un routage est demandé.

2. Fonctionnalités offertes par les MOM

Les Middleware Orientés Message, outre les services d'acheminement (envoi, réception), de stockage, et de recherche des messages etc ..., offrent des services plus évolués comme:

- Rendre certains messages plus prioritaires que d'autres
- Compresser les données utiles du messsage

- Faire expirer un message à une date donnée
- Ne rendre un message disponible qu'à partir d'une certaine date (sur certains MOM uniquement)
- Des services de routage des messages d'un noeud à l'autre (un peu à la manière des serveurs de mails)
- Des fonctionnalités de triggering: lancement d'applications lorsque des messages sont disponibles pour elle
- Des possibilités d'alertes suivant la présence de messages dans une file donnée ou suivant un nombre de messages donné.

Attention, la compression n'est pas implémentée par tous les MOM. Par exemple XMS (maintenant Synchrony Messaging (Axway)) et ActiveMQ permettent de compresser les messages, mais MQ Series requiert que ce soit l'application qui se charge de compresser le message si elle le souhaite.

Les messages disponibles à partir d'une certaine date constituent également une fonctionnalité optionnelle offerte par certains MOM comme Synchrony Messaging.

3. Domaines d'utilisation

Les Middleware Orientés Message sont très utilisés dans le domaine de l'EAI (Enterprise Application Integration) ainsi que dans les ESB (Enterprise Service Bus).

Les autres secteurs utilisateurs de MOM incluent, par exemple, le Data Warehouse, les messageries inter-bancaires (par exemple le broker de messages open source AMQ) ainsi que la diffusion d'informations.

6) Application RPC

1. Interface

Pour comprendre le fonctionnement des RPC, nous allons donc écrire, à titre d'exemple, un programme simple mais néanmoins complet . Il y a en fait deux programmes, un programme client et un programme serveur. La fonction distante prendra deux nombres en paramètres et renverra leur somme ainsi qu'un code d'erreur indiquant s'il y a eu un overflow ou non. Le travail commence donc par la définition de l'interface, celle-ci étant écrite en utilisant l'IDL (Interface Definition Language) du système RPC (proche du C).

```
struct data {
  unsigned int arg1;
  unsigned int arg2;
};

typedef struct data data;

struct reponse {
  unsigned int somme;
  int errno;
};

typedef struct reponse reponse;

program CALCUL{
  version VERSION_UN{
    void CALCUL_NULL(void) = 0;
    reponse CALCUL_ADDITION(data) = 1;
  } = 1;
} = 0x20000001;
```

Cette définition est enregistrée dans le fichier calcul.x . Ce fichier décrit notre programme et les fonctions qu'il contient. La définition parle d'elle-même. Notre programme s'appelle CALCUL et dans sa version VERSION_UN contient deux procédures: CALCUL_NULL et CALCUL_ADDITION.

Le programme a un nom (ici CALCUL) et un numéro (ici 0x20000001), ce numéro identifie ce programme de manière unique dans le monde. C'est pratique pour des programmes comme le daemon NFS. Dans notre cas, le numéro est choisi dans l'intervalle allant de 0x20000000 à 0x3FFFFFFF, réservé pour les utilisateurs et ne risque pas d'entrer en conflit avec des programmes tournant déjà sur votre machine.

Ensuite, pour un programme donné, on peut avoir plusieurs versions, ceci afin de pouvoir offrir de nouvelles fonctions dans la nouvelle version tout en conservant les versions précédentes (pour les anciens programmes clients). Ici, nous avons donc une version appelée VERSION_UN et qui a pour numéro 1.

Vient ensuite, la liste des procédures que le serveur implémente. Chaque procédure a un nom et un numéro. Une procédure de numéro 0 et qui ne fait rien (ici CALCUL_NULL) est toujours requise. Ceci afin de tester si le système marche (une sorte de ping en quelque sorte). On peut l'appeler afin de vérifier que le réseau fonctionne et que le serveur tourne. La seconde procédure décrite est notre procédure d'addition. Elle prend un argument de type data et renvoie un argument de type reponse. Le système RPC n'autorise qu'un seul argument en paramètre et un seul

en retour, pour passer plusieurs arguments (dans notre cas les deux nombres à additionner), on utilise donc une structure. De même pour renvoyer plusieurs valeurs (dans notre cas le résultat de l'addition et un code d'erreur), on utilise une structure .

Ce fichier de définition est ensuite traité par l'utilitaire rpcgen (RPC program generator), il suffit de taper sur la ligne de commande:

```
rpcgen -a calcul.x
```

L'option -a permet de produire un squelette pour notre programme client (calcul_client.c) et un squelette pour la fonction distante (calcul_server.c). Avec ou sans cette option, les fichiers suivants sont également produits: calcul.h (entête), calcul_clnt.c (stub client), calcul_svc.c (stub serveur) et calcul_xdr.c (routines XDR).

Le format XDR (eXternal Data Representation) définit les types utilisés pour l'échange de variables entre le client et le serveur. Il est possible que les processus serveur et client ne tournent pas sur la même plateforme, il est donc indispensable de parler une langue commune. Ainsi ce format définit précisément un codage pour les entiers, les flottants... Les structures plus complexes utilisent les types de base. Ainsi, les types que nous avons nous-mêmes définis nécessitent un filtre XDR, c'est le rôle des fonctions définies dans le fichier calcul_xdr.c, à compiler puis à lier avec le client et le serveur. La compilation peut être faite maintenant (c'est déjà ça) et produit un calcul_xdr.o :

```
gcc -c calcul_xdr.c
```

Les stubs client et serveur sont complets et peuvent déjà être compilés. La seule connaissance de l'interface suffit à rpcgen pour les générer. Donc, nous pouvons compiler pour produire les fichiers calcul_clnt.o et calcul_svc.o :

```
gcc -c calcul_clnt.c
gcc -c calcul_svc.c
```

2. Processus serveur

Ceci étant fait, écrivons maintenant la fonction distante qui effectue réellement le travail. Grâce à l'option -a de rpcgen nous avons le squelette de fonction suivant dans le fichier calcul_server.c:

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calcul.h"

void *
calcul_null_1_svc(void *argp, struct svc_req *rqstp)
{

  static char* result;

  /*
   * insert server code here
   */

  return((void*) &result);
```

```
}

reponse *
calcul_addition_1_svc(data *argp, struct svc_req *rqstp)
{

  static reponse  result;

  /*
   * insert server code here
   */

  return(&result);
}
```

Après examen du fichier produit par rpcgen, quelques explications s'imposent, car on peut remarquer quelques différences par rapport à notre définition. Les fonctions ont deux arguments, le deuxième n'a pas d'utilité pour notre exemple. Au lieu de récupérer une variable du type demandé dans la définition, on doit en fait passer par un pointeur sur cette variable (ceci évite une copie des arguments et donc il y a un gain de temps et de mémoire). Pour le retour, c'est également un pointeur qui est renvoyé. Comme on utilise des pointeurs, la variable dans laquelle on met la valeur de retour est déclarée 'static' car il faut bien évidemment passer l'adresse d'une variable qui existe encore après la fin de la fonction.

Comme vous le voyez, il n'y a pas de fonction main. La fonction main est située dans le stub serveur. Le stub serveur s'occupe de recevoir et de "dispatcher" les appels aux fonctions adéquates. Notre seul travail est d'écrire les fonctions. Donc, après modifications, le fichier calcul_server.c devient:

```
#include "calcul.h"

void *
calcul_null_1_svc(void *argp, struct svc_req *rqstp)
{
  static char* result;
  /* Ne rien faire */
  return((void*) &result);
}

reponse *
calcul_addition_1_svc(data *argp, struct svc_req *rqstp)
{
  static reponse  result;
  unsigned int max;

  result.errno = 0; /* Pas d'erreur */

  /* Prend le max */
  max = argp->arg1 > argp->arg2 ? argp->arg1 : argp->arg2;

  /* On additionne */
  result.somme = argp->arg1 + argp->arg2;

  /* Overflow ? */
  if ( result.somme < max ) {
    result.errno = 1;
  }

  return(&result);
```

```
}
```

Nous pouvons alors le compiler en faisant:

```
gcc -c calcul_server.c
```

Puis pour obtenir le programme serveur complet, il faut lier calcul_svc.o, calcul_server.o et calcul_xdr.o ensemble:

```
gcc -o server calcul_svc.o calcul_server.o calcul_xdr.o
```

A ce stade, nous avons terminé la partie serveur de notre application. On peut alors démarrer le serveur, puis utiliser rpcinfo pour vérifier qu'il tourne:

```
> ./server &
[2] 209
> rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  rpcbind
    100000    2   udp    111  rpcbind
    100005    1   udp    673  mountd
    100005    2   udp    673  mountd
    100005    1   tcp    676  mountd
    100005    2   tcp    676  mountd
    100003    2   udp   2049  nfs
    100003    2   tcp   2049  nfs
 536870913    1   udp    897
 536870913    1   tcp    899
> rpcinfo -u localhost 536870913
program 536870913 version 1 ready and waiting
```

L'option -p de rpcinfo permet de connaître la liste des programmes RPC actuellement enregistrés sur la machine. Pour chaque programme, on obtient le numéro de version, le protocole et le port utilisés. Veuillez noter, que les numéros de programmes sont affichés en décimal . Sachant que 536870913 est l'équivalent décimal de 0x20000001, tout va bien notre programme est bien enregistré. L'option -u quant à elle, permet de tester le programme indiqué en appelant sa procédure 0 (rappelez vous qu'il est obligatoire d'avoir au moins une procédure de numéro 0). Pour plus d'information, vous pouvez consulter la page man de rpcinfo.

3. Processus client

Là encore, pour simplifier, on utilise le squelette que nous a fourni rpcgen en produisant le fichier calcul_client.c :

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "calcul.h"


void
calcul_1( char* host )
{
  CLIENT *clnt;
  void  *result_1;
  char*  calcul_null_1_arg;
```

```
  reponse  *result_2;
  data  calcul_addition_1_arg;
  clnt = clnt_create(host, CALCUL, VERSION_UN, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror(host);
    exit(1);
  }
  result_1 = calcul_null_1((void*)&calcul_null_1_arg, clnt);
  if (result_1 == NULL) {
    clnt_perror(clnt, "call failed:");
  }
  result_2 = calcul_addition_1(&calcul_addition_1_arg, clnt);
  if (result_2 == NULL) {
    clnt_perror(clnt, "call failed:");
  }
  clnt_destroy( clnt );
}


main( int argc, char* argv[] )
{
  char *host;

  if(argc < 2) {
    printf("usage: %s server_host\n", argv[0]);
    exit(1);
  }
  host = argv[1];
  calcul_1( host );
}
```

Des variables sont déclarées pour les arguments et les valeurs de retour. Comme vous pouvez le constater le programme squelette généré comprend un appel à chacune des fonctions définies dans l'interface (ici CALCUL_NULL et CALCUL_ADDITION). On peut remarquer que chaque appel est suivi d'un test qui détecte les erreurs de niveau "RPC" (serveur ne répondant pas, machine inexistante, ...). L'erreur éventuelle est alors explicitée par la fonction clnt_perror(). Quand une erreur de niveau "RPC" se produit, le pointeur renvoyé est à NULL. Dans vos fonctions, vous ne devez donc pas mettre le pointeur à NULL pour spécifier une erreur. Pour un niveau d'erreur autre que RPC, vous pouvez utiliser une valeur particulière de la valeur de retour (comme un nombre négatif par exemple) et renvoyer tout à fait normalement le pointeur sur cette variable. Dans l'exemple, le choix a été fait d'utiliser une deuxième variable (champ errno de la structure reponse) pour spécifier s'il y a eu une erreur ou non (car l'utilisation de valeurs particulières est discutable).

Pour faire un vrai programme, il nous faut donner des valeurs aux paramètres et il faut utiliser effectivement les résultats des appels distants (par exemple en les affichant à l'écran). C'est ce que nous faisons avec notre programme client (dans lequel il n'y a volontairement pas d'appel à la procédure CALCUL_NULL) :

```
#include <limits.h>
#include "calcul.h"

CLIENT *clnt;

void
test_addition (uint param1, uint param2)
{
  reponse  *resultat;
  data  parametre;
```

```
  /* 1. Preparer les arguments */

  parametre.arg1 = param1;
  parametre.arg2 = param2;
  printf("Appel de la fonction CALCUL_ADDITION avec les parametres: %u et
%u \n", parametre.arg1,parametre.arg2);

  /* 2. Appel de la fonction distante */

  resultat = calcul_addition_1 (&parametre, clnt);
  if (resultat == (reponse *) NULL) {
    clnt_perror (clnt, "call failed");
    clnt_destroy (clnt);
    exit(EXIT_FAILURE);
  }
  else if ( resultat->errno == 0 ) {
    printf("Le resultat de l'addition est: %u \n\n",resultat->somme);
  } else {
    printf("La fonction distante ne peut faire l'addition a cause d'un
overflow \n\n");
  }

}

int
main (int argc, char *argv[])
{
  char *host;

  if (argc < 2) {
    printf ("usage: %s server_host\n", argv[0]);
    exit (1);
  }
  host = argv[1];

  clnt = clnt_create (host, CALCUL, VERSION_UN, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }

  test_addition ( UINT_MAX - 15, 10 );
  test_addition ( UINT_MAX, 10 );

  clnt_destroy (clnt);
  exit(EXIT_SUCCESS);
}
```

Nous pouvons alors compiler puis lier avec calcul_clnt.o et calcul_xdr.o pour produire le client. Enfin, on peut tester le résultat final (en ayant démarré le processus serveur avant).

```
> gcc -c calcul_client.c
> gcc -o client calcul_client.o calcul_clnt.o calcul_xdr.o

> ./client localhost
Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967280 et 10
Le resultat de l'addition est: 4294967290

Appel de la fonction CALCUL_ADDITION avec les parametres: 4294967295 et 10
La fonction distante ne peut faire l'addition a cause d'un overflow
>
```

Le client prend en paramètre le nom de la machine serveur (ici localhost car le processus serveur a été démarré sur la même machine). Le nom peut être court (machine) pour une machine du même domaine que le client ou complet (machine.domaine.com).

7) Application MOM

1. Quelques Message Oriented Middleware

MOM commercialisés :

- IBM MQSeries (ObjectWeb)
- MSMQ de Microsoft
- TIBCO Rendezvous de Tibco Software
- Synchrony Messaging d'Axway
- SonicMQ de Progress Software

MOM Open-Source :

- JORAM (ObjectWeb)
- ActiveMQ (Apache Software Foundation)
- OpenJMS (OpenJMS Group
- Jboss Messaging (Jboss)

2. Java Message Service

2.1 Qu'est-ce que l'API Java Message Service?

JMS (Java Message Service), est la spécification de Sun pour la gestion des messages de la plate-forme Java EE (Nouvelle dénomination de J2EE). En peu de mots, JMS est aux Middleware Orientés Message ce que JDBC est aux bases de données relationnelles.

On parle ainsi d'API JMS, mais au strict sens du terme, ça n'existe pas.

Sun a spécifié les besoins d'une API en mode message, a défini les obligations de la part du provider implémentant la spécification, ainsi que celles du client, et ensuite les éditeurs de MOM comme IBM, BEA ou Tibco ont implémenté cette spécification.

Donc, ce que l'on désigne généralement par API JMS est en fait une implémentation de la spécification JMS de Sun.

Par la suite Sun a écrit lui aussi une implémentation de sa spécification JMS 1.1 avec son produit Sun Java System Message Queue.

On appelle alors provider un éditeur de MOM qui a implémenté une API conformément aux recommandations de la norme JMS de Sun Microsystems.

Pour vérifier que le provider respecte bien la norme, celui-ci doit exécuter des tests de conformités avec un outil spécialement développé pour cela, il y a notamment celui de Sun (mais il faut alors être éditeur d'un serveur d'applications), ou celui d'OpenJMS, le JMS CTS (JMS Compliance Test Suite).

L'API JMS existe depuis 1998, la dernière spécification de JMS est la version 1.1 qui date d'Avril 2002.

2.3 Une API standard d'accès aux MOM

JMS est considérée comme une API standard d'accès aux MOM car JMS n'est en réalité pas une API, mais une spécification, et fait partie de la spécification Java EE.

Tout utilisateur d'une API conforme à JMS sait que son applicatif se comportera (il le doit car le provider respecte la norme !) de la même façon avec WebSphere MQ ou avec Joram par exemple.

La spécification de Sun décrit très précisément comment doit se comporter un provider JMS, et comme "l'API" JMS définie par Sun ne comporte quasiment que des interfaces Java, on comprend facilement que le travail du programmeur développant une application JMS est facilité.

Comme JMS est une API Java, l'envoi et la réception de messages sont grandement facilités par le modèle objet de Java, et les types de messages qui existent dans JMS vont du message sans données utiles (mais avec une entête et éventuellement des propriétés), au message texte, au flux de bytes, ... jusqu'à l'objet Java sérialisé.