Systèmes répartis : les *Remote Procedure Calls*

Fabrice Rossi

http://apiacoa.org/contact.html.

Université Paris-IX Dauphine

Les *Remote Procedure Calls*

Principe de base : transparence réseau pour les appels de fonctions. Composants :

- un langage de description des données et des fonctions
- une méthode de représentation portable des données
- un protocole d'appel :
  - transmission des paramètres de l'appel du client vers le serveur
  - transmission du résultat du serveur vers le client
- un protocole de publication et de recherche d'un service

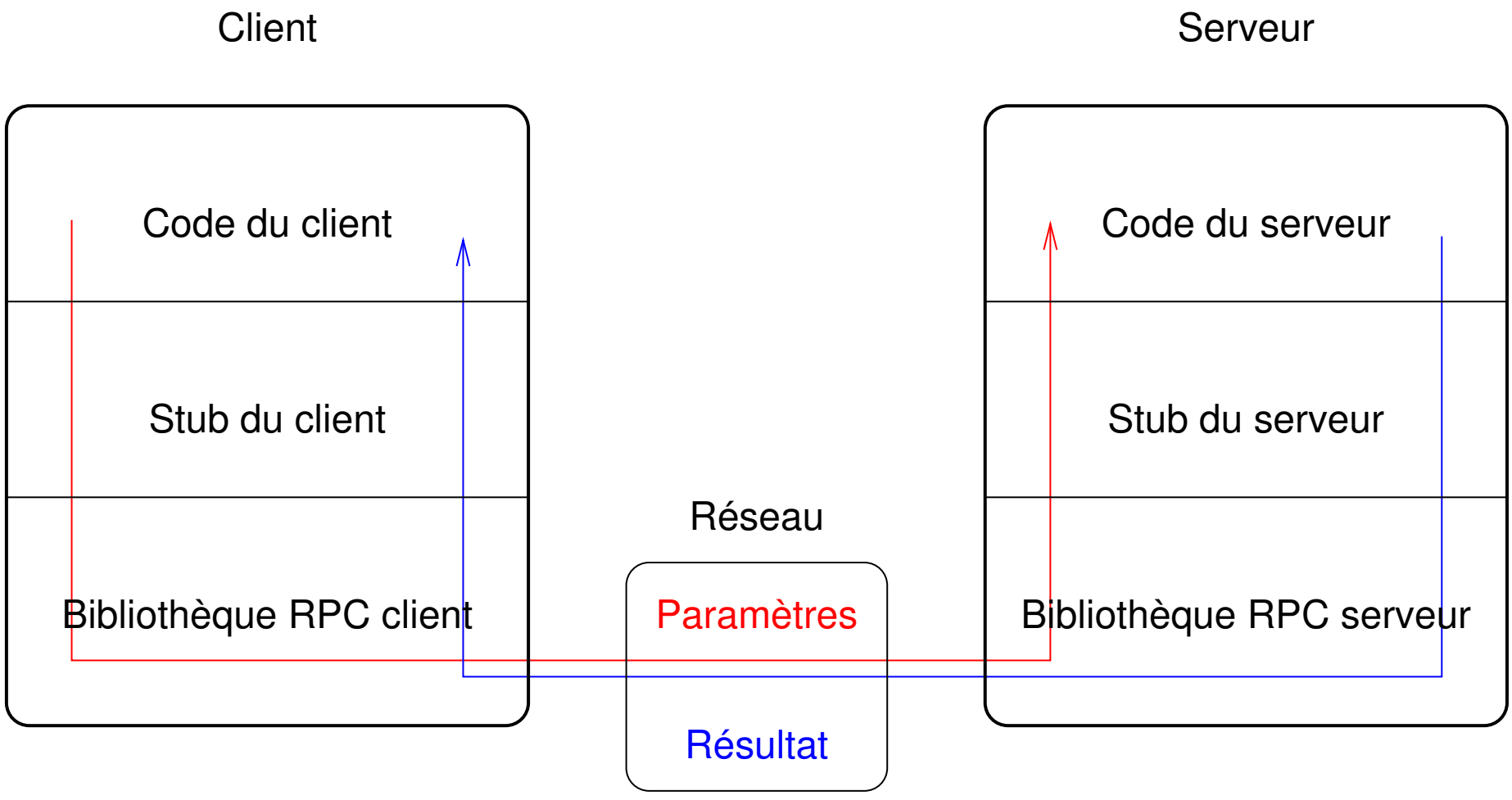
Deux grandes normes :

- RPC ONC (Sun)
- RPC DCE (Open Group)

Mais aussi :

- XML RPC
- SOAP

Schéma de principe



Rôle des *Stubs* (Souches)

Les souches :

- *stub*
- souvent appelé squelette (*skeleton*) du côté serveur
- sur le client :
  - paramètres $\rightarrow$ format portable (*marshalling*)
  - format portable $\rightarrow$ résultat (*unmarshalling*)
  - appel de la bonne fonction
- sur le serveur :
  - traductions inverses de celles du client
  - appel de la bonne fonction
- en général engendrées automatiquement

Rôle des bibliothèques

- fonctions de bas niveau pour le *(un)marshalling*
- établissement des connexions réseau
- mise en œuvre du protocole :
  - identification du programme
  - authentification (éventuelle)
  - gestion de certaines erreurs (de protocole, de version, etc.)
  - transmission effective des données (en particulier la gestion de la mémoire)
  - publication et découverte d'un service
- fournies par le "vendeur" de la solution RPC

Code à écrire

Il reste quand même des choses à écrire (!) :

- dépend de la solution retenue
- RPC ONC :
  - le code de la fonction pour le serveur (le code de démarrage est engendré par les outils)
  - le code du client :
    - connexion au serveur
    - appel de la fonction
  - les outils RPC ONC proposent en général une version à compléter de tout ce que le programmeur doit fournir
- autres solutions : en gros la même chose

Etude d'un exemple : RPC ONC

Les RPC ONC (Sun) :

- rfc 1831
- solution classique disponible sur tous les Unix (incluse dans le système en général) et sous windows
- comparable aux DCE RPC en "gratuit"
- langage de description des données : XDR (rfc 1832)
- technologie utilisée par *Network File System*
- outil de base `rpcgen` :
  - engendre des convertisseurs (*(un)marshalling*)
  - engendre les souches (serveurs et clients)
  - propose des modèles pour le client et le serveur
  - engendre un `Makefile`
- utilise un *lookup service* pour la publication et la découverte d'un service (rfc 1833)
- les RPC ONC peuvent être implantées sur n'importe quelle couche transport (en général TCP et UDP)

Le protocole RPC (rfc 1831)

Principes de base :

- un service réseau est fournis par plusieurs programmes
- un programme fournis plusieurs fonctions (ou procédures)
- un client appelle une procédure d'un programme
- une procédure est identifiée par trois entiers (positifs) :
  - le numéro de programme
  - le numéro de version du programme
  - le numéro de procédure au sein du programme
- affectation des numéros de programme :
  - de `0x0` à `0x1fffffff` : affectation par SUN
  - de `0x20000000` à `0x3fffffff` : affectation par le programmeur
  - de `0x40000000` à `0xffffffff` : réservé
- le protocole est basé sur l'échange de messages décrits en XDR

eXternal Data Representation (rfc 1832)

Représentation portable des données :

- codage *big endian* avec des mots de 4 octets (complétion par des zéros)
- permet de définir des **types de données** associés à une **représentation binaire standard**
- orienté C :
  - syntaxe proche du C
  - types fondamentaux classiques (`int`, `double`, etc.)
  - énumération
  - `struct` et `union`
  - tableaux
- `rpcgen` engendre des fonctions de traduction à partir d'un fichier XDR (et des bibliothèques de conversion de bas niveau)

Exemple

person.x

```
1  enum type_gender {
2    FEMALE=0,
3    MALE=1
4  };
5
6  struct person {
7    string firstname<>;
8    string lastname<>;
9    type_gender gender;
10 };
```

On définit :

- `type_gender` : un énuméré pour le genre
- `person` : une `struct` contenant deux chaînes de caractères et un genre

Les types

Type	identificateur	taille
Entier relatif	`int`	4 octets
Entier positif	`unsigned int`	4 octets
Enumération	`enum {identifier=constant,...}`	`int`
Booléen	`bool`	`enum {FALSE=0, TRUE=1}`
Entier long relatif	`hyper`	8 octets
Entier long positif	`unsigned hyper`	8 octets

Les types (2)

Type	identificateur	taille
Réels simples	`float`	4 octets (IEEE)
Réels doubles	`double`	8 octets (IEEE)
Réels longs	`quadruple`	16 octets (IEEE)
Rien	`void`	0 octet

Les types (3)

- Données à taille fixe ou variable :
  - `[n]` correspond à une taille fixe égale à `n`
  - `<>` correspond à une taille variable
  - `<m>` correspond à une taille variable inférieure à `m`
  - exemple, `int<15>` : liste d'entiers de longueur au plus 15.
  - Cas particuliers :

Identificateur	contenu
`opaque`	liste d'octets
`string`	chaîne de caractères

  Attention, pas de taille fixe pour les chaînes de caractères.
- Comme en C :
  - on peut définir des types par `typedef`
  - on peut définir des constantes

Les types (4)

Comme en C :

- `struct` permet d'agréger des types pour fabriquer un type plus complexe (cf exemple d'introduction)
- `union` permet de decrire des types variables. Exemple :

person.x

```
1  enum type_gender {
2    FEMALE=0,
3    MALE=1
4  };
5  union type_maidenname switch (type_gender gender) {
6    case FEMALE:
7      string maidenname<>;
8    case MALE:
9      void;
10 };
```

Définition d'un programme

Pour décrire un programme RPC, on utilise une extension de XDR :

calculation.x

```
1  struct two_int {
2      int a;
3      int b;
4  };
5  program CALCULATION_PROG {
6      version CALCULATION_VERS_BASE {
7          int sum(two_int)=1;
8      }=1;
9  }=0x20000000;
```

Principe :

- `program identificateur {version...}=constante ;`
- `version identificateur {procedure...}=constante ;`
- `procedure type identificateur (type,...)=constante ;`

Programmation

On soumet `calculation.x` au programme `rpcgen`. On obtient (sous Linux) :

- `calculation.h` : fichier d'en-tête à inclure dans les différents programmes
- `calculation_xdr.c` : les fonctions de conversion en XDR des types définis dans `calculation.x`
- `calculation_clnt.c` : le *stub* côté client
- `calculation_svc.c` : le *stub* côté serveur (par défaut, support de `udp` et de `tcp`)
- de façon optionnelle :
  - un `Makefile`
  - des exemples à compléter pour le client et le serveur
- nombreuses options (mode de démarrage du serveur, couche de transport, etc.)

calculation_server.c

calculation_server.c

```
1  #include "calculation.h"
2
3  int *
4  sum_1_svc(two_int *argp, struct svc_req *rqstp)
5  {
6          static int  result;
7
8          /* partie ajoutée ! */
9          result=argp->a+argp->b;
10         /* fin de l'ajout */
11         return &result;
12 }
```

- totalement basique
- toute la complexité est dans `calculation_svc.c`

Synopsis de la partie cliente

Dans un client :

1. on établit la connexion (`clnt_create`, ligne 14). On doit préciser :
   - la machine cible (`host`)
   - le numéro du programme et son numéro de version (macros engendrées par `rpcgen`)
   - le transport utilisé
2. on effectue l'appel qui est local, cf ligne 20 (seule astuce : le dernier paramètre contient les informations de connexion)
3. on coupe la connexion (`clnt_destroy`, ligne 30)

Avec un transport connecté (type `tcp`), on peut conserver la connexion ouverte pendant un certain temps.

calculation_client.c

calculation_client.c

```
1  #include "calculation.h"
2
3  void calculation_prog_1(char *host)
4  {
5      CLIENT *clnt;
6      int  *result_1;
7      two_int  sum_1_arg;
8
9      /* partie ajoutée */
10     sum_1_arg.a=15;
11     sum_1_arg.b=35;
12     /* fin de l'ajout */
13 #ifndef        DEBUG
14     clnt = clnt_create(host,CALCULATION_PROG,CALCULATION_VERS_BASE,"udp");
15     if (clnt == NULL) {
16         clnt_pcreateerror (host);
17         exit (1);
18     }
19 #endif        /* DEBUG */
```

calculation_client.c (2)

calculation_client.c

```
20      result_1 = sum_1(&sum_1_arg, clnt);
21      if (result_1 == (int *) NULL) {
22          clnt_perror (clnt, "call failed");
23      }
24      /* partie ajoutée */
25      else {
26          printf("%d+%d=%d\n",sum_1_arg.a,sum_1_arg.b,*result_1);
27      }
28      /* fin de l'ajout */
29  #ifndef         DEBUG
30      clnt_destroy (clnt);
31  #endif          /* DEBUG */
32  }
```

- solution très basique (création de `clnt` à chaque appel par exemple)
- utilise la partie bilbiothèque des RPC (e.g., `clnt_create`)
- l'appel proprement dit est local (ligne 20)

calculation_client.c (3)

calculation_client.c

```
33  int main (int argc, char *argv[])
34  {
35      char *host;
36
37      if (argc < 2) {
38          printf ("usage: %s server_host\n", argv[0]);
39          exit (1);
40      }
41      host = argv[1];
42      calculation_prog_1 (host);
43      exit (0);
44  }
```

Publication et découverte

Un problème standard des systèmes distribués :

- pour le serveur, se faire connaître (publication)
- pour le client, trouver le serveur (découverte et résolution)

Plusieurs niveaux :

1. partie facile : nom de la machine du serveur + DNS $\rightarrow$ connexion possible (TCP ou UDP par exemple)
2. quelle couche transport ?
3. pour TCP/UDP, quel numéro de port ?

Solution proposée par les RPC ONC, un programme de *binding* (RFC 1823) :

- le *portmapper* (version 2) : spécifique à TCP/UDP
- *rpcbind* (versions 3 et 4) : indépendant du transport

Binding RPC

Principes communs aux différentes versions :

- le *binder* est un programme RPC de numéro `100000`
- il écoute toujours sur les ports 111 UDP et TCP

Le `portmapper` offre différentes fonctions :

- publication d'un programme (décrit par son numéro de programme et son numéro de version)
- suppression d'un programme
- découverte (*résolution*) d'un programme, c'est-à-dire obtention du port associé à un programme
- obtention de la liste des programmes enregistrés
- appel direct d'une fonction distante

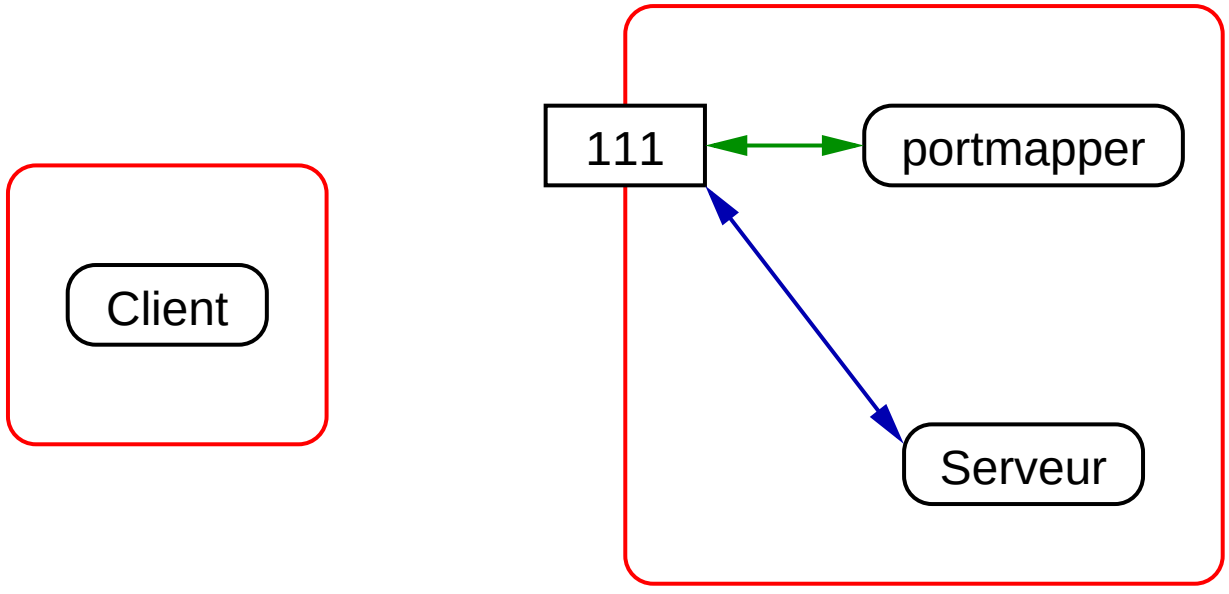
Le `portmapper` est limité à TCP et UDP.

rpcbind

Remplace le `portmapper`. Principaux avantages :

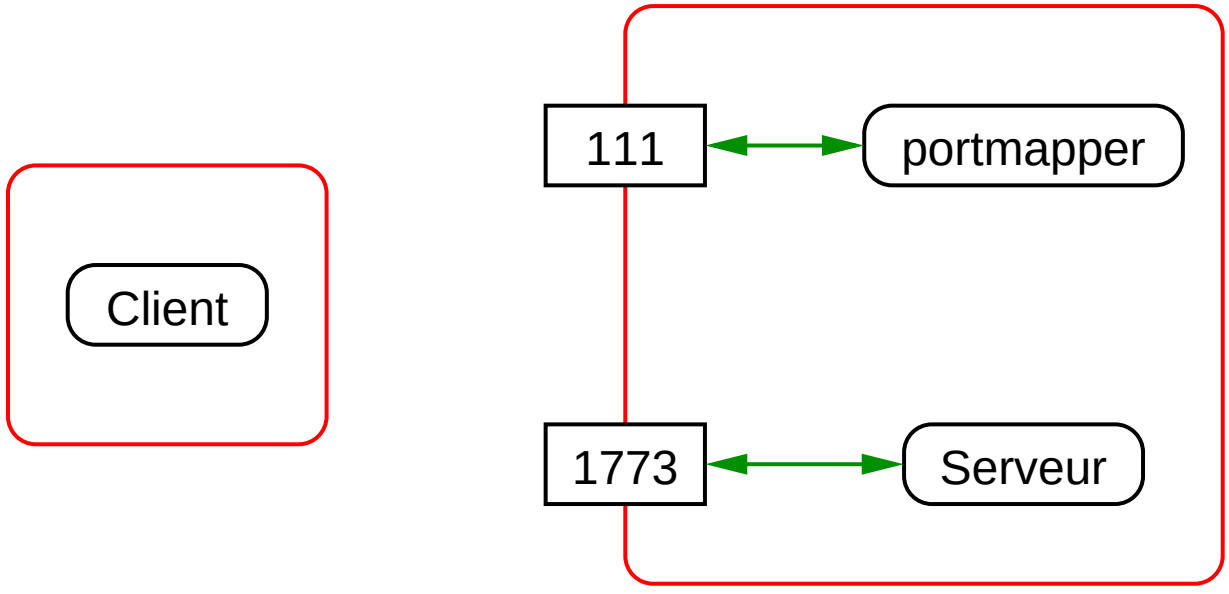
- indépendant du transport
- permet un enregistrement et une résolution incluant les informations de transport
- offre quelques nouvelles fonctions :
  - horloge
  - statistiques d'utilisation
  - un programme peut enregistrer plusieurs instances correspondant à différents transport (avec le `portmapper`, le transport est automatique celui utilisé pour contacter le `portmapper` lui-même)

Fonctionnement du système



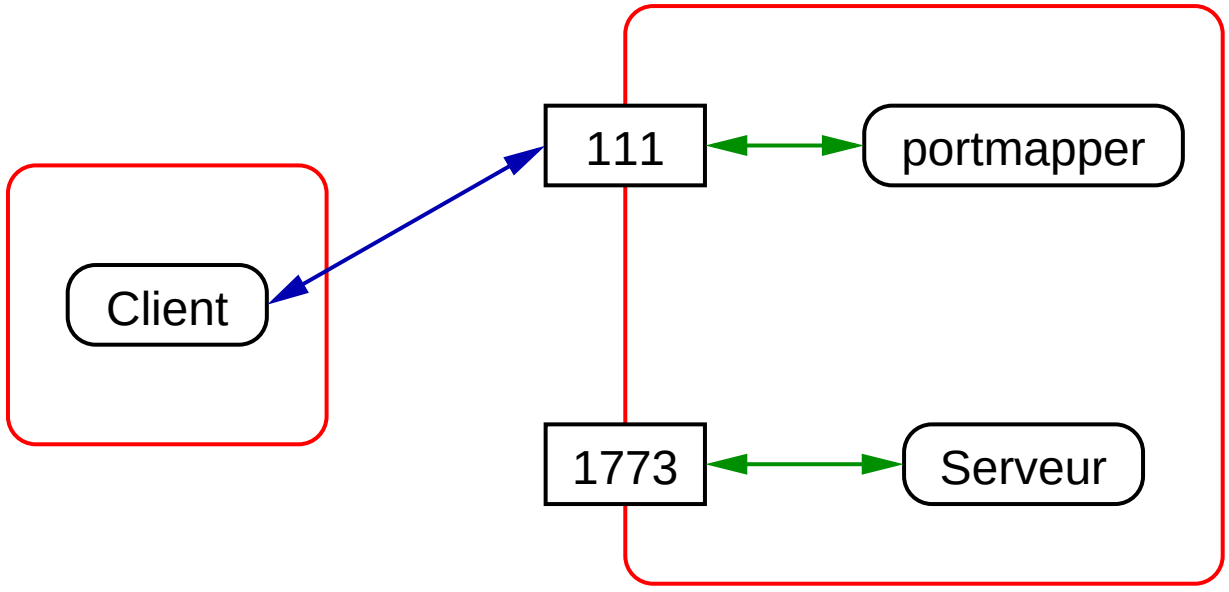
Publication du programme (RPC)

Fonctionnement du système



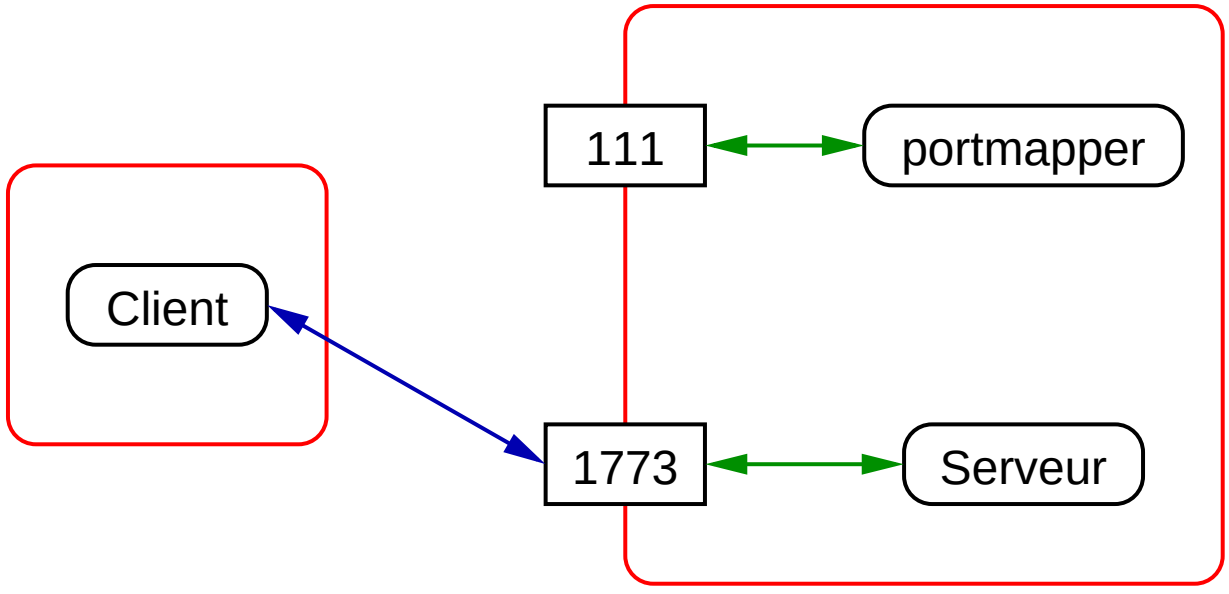
Le programme écoute sur le port 1773

Fonctionnement du système



Demande de résolution (RPC)

Fonctionnement du système



Appel de la fonction recherchée