

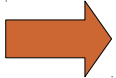
« Remote Procedure Call » (RPC)

Module FAR

Polytech Montpellier – IG3

David Delahaye

Motivations du RPC

- On sait faire discuter deux hôtes entre eux (adresses IP, protocoles jusqu'à la couche réseau IP).
- On sait faire discuter deux processus sur deux hôtes distants (adresses IP et ports, protocoles de la couche transport TCP et UDP).
- Comment appeler une procédure distante (c'est-à-dire sur un hôte distant) et ce, de manière transparente pour le développeur ?
-  La technique RPC apporte une solution

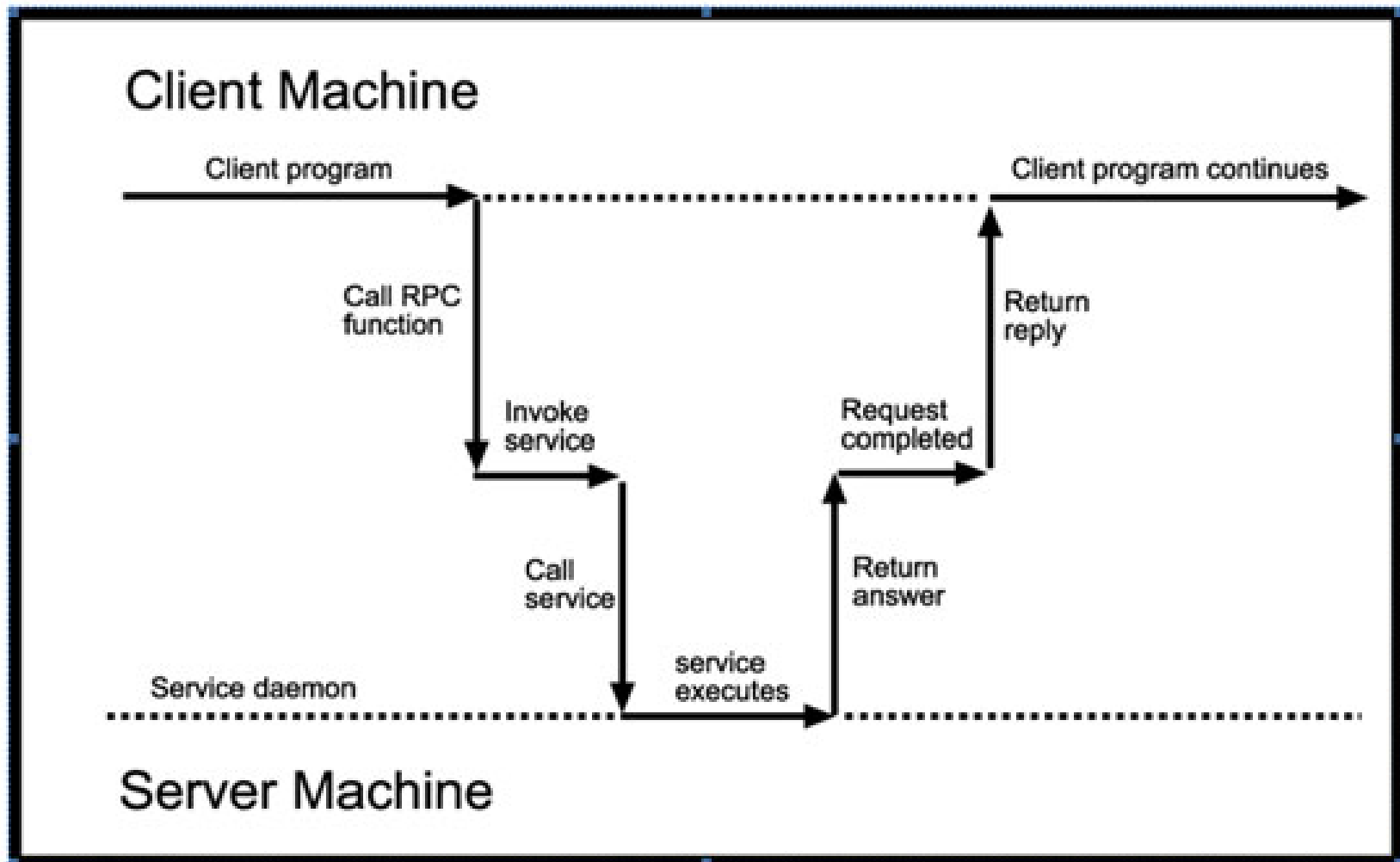
Un peu d'histoire

- Idée qui date de la fin des années 70 (RFC 707).
- Pas de standard (beaucoup de versions différentes et incompatibles entre elles).
- Implantation Unix due à Sun (« Sun/ONC RPC »).
- Spécifications dans le domaine public et disponible en standard sous tous les systèmes Unix.
- Développé initialement pour servir de base au système NFS (« Network File System »), très utilisé sous Unix, et permettant d'accéder à des fichiers distants (sur un autre disque réseau) de manière transparente pour l'utilisateur.

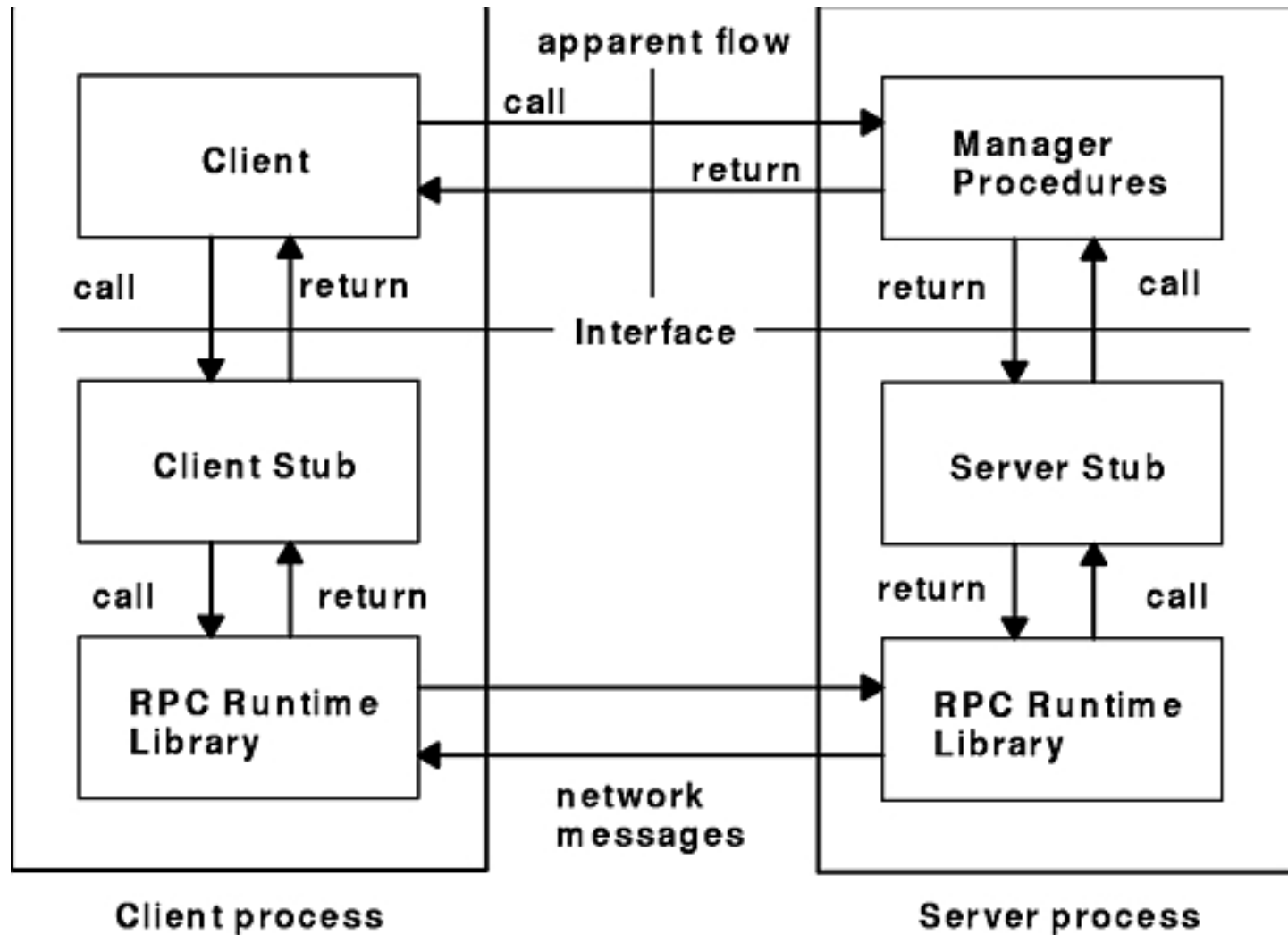
Principe d'un appel distant RPC

- Architecture client/serveur.
- Le client effectue une requête d'appel à une fonction offerte par le serveur.
- Le serveur distant effectue l'appel de fonction, puis envoie une réponse au client.
- Le client est bloqué le temps de la requête et tant qu'il n'a pas reçu de réponse.
- Le serveur peut répondre à plusieurs requêtes en même temps (principe même du serveur).

Principe d'un appel distant RPC



Principe d'un appel distant RPC



Remote Procedure Call Flow

Principe d'un appel distant RPC

- Côté serveur, on écrit la fonction qui va être appelée.
- Côté client, on appelle la fonction avec son nom.
- La transmission réseau (TCP/UDP) est prise en charge par des « stubs ».
- La transmission des paramètres de l'appel et sa réponse aux « stubs » est appelé « marshalling ».
- Une grande partie du processus va être gérée automatiquement (par « rpcgen ») et les parties client/serveur seront développées en même temps.

Interface (« calcul.x »)

```
struct data {  
    int arg1;  
    int arg2;  
};  
typedef struct data data;  
struct reponse {  
    int somme;  
    int errno;  
};  
    ↪ peu savoir si erreur (-1, 0, ...)  
typedef struct reponse reponse;
```

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


Génération automatique des squelettes de fichiers

- Générés automatiquement avec la commande :
 - `rpcgen -a calcul.x`
- Génère les fichiers (ainsi que « `Makefile.calcul` ») :
 - « `calcul_client.c` » (squelette client)
 - « `calcul_server.c` » (squelette serveur)
 - « `calcul_clnt.c` », « `calcul_svc.c` » (« stubs » client et serveur, ne pas toucher)
 - « `calcul_xdr.c` » (routines XDR)

Format XDR

- XDR = « eXternal Data Representation ».
- Définit les types utilisés pour l'échange de variables entre le client et le serveur.
- Permet de s'assurer que l'échange de structures de données entre les deux plate-formes est correct.
- Les types définis dans le fichier « .x » nécessitent un filtre XDR (fonctions définies dans « calcul_xdr.c »).
- Compilation :
 - gcc -c calcul_xdr.c

« Stubs » client et serveur

- Les « stubs » gèrent les connexions réseaux (complètement transparent pour l'utilisateur).
- Deux modes possibles : TCP ou UDP.
- Une fois générés, les « stubs » ne sont pas à modifier.
- On peut donc les compiler de suite :
 - `gcc -c calcul_clnt.c`
 - `gcc -c calcul.svc.c`

Processus serveur

- Fichier « calcul_server.c » (squelette) :

```
#include "calcul.h"
```

```
void *calcul_null_1_svc(void *argp, struct svc_req *rqstp)
{
    static char * result;

    /*
     * insert server code here
     */

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

Processus serveur

```
reponse *calcul_addition_1_svc(data *argp, struct svc_req *rqstp)
{
    static reponse result;

    /*
     * insert server code here
     */

    return &result;
}
```

Processus serveur

- Modification (fonction d'addition) :

```
reponse *calcul_addition_1_svc(data *argp, struct svc_req *rqstp)
{
    static reponse result; pas pointeur
    result.somme = argp->arg1 + argp->arg2;
    return &result;
}
```

- Compilation et génération de l'exécutable :
 - gcc -c calcul_server.c
 - gcc -o calcul_server calcul_svc.o calcul_server.o calcul_xdr.o

Processus serveur

- Lancement du serveur :
 - `$./calcul_server &`
`[1] 26864`
 - `$ rpcinfo -p`

...
 - `536870913 1 udp 41592`
`536870913 1 tcp 51586`
 - `$ rpcinfo -u localhost 536870913`
`program 536870913 version 1 ready and waiting`

Processus client

- Fichier « calcul_client.c » (squelette) :

```
#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;
```


Processus client

```
void calcul_1(char *host)
{ ...
    result_1 = calcul_null_1((void*)&calcul_null_1_arg, clnt);
    if (result_1 == (void *) NULL) {
        clnt_perror (clnt, "call failed");
    }
    result_2 = calcul_addition_1(&calcul_addition_1_arg, clnt);
    if (result_2 == (reponse *) NULL) {
        clnt_perror (clnt, "call failed");
    }
}
```

Processus client

```
int 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);
    exit (0);
}
```

Processus client

- Modification (fonction d'addition et appel) :

```
CLIENT *clnt;
```

```
int addition (int a, int b) {  
    reponse *resultat;  
    data parametre;  
    parametre.arg1 = a;  
    parametre.arg2 = b;  
    resultat = calcul_addition_1 (&parametre, clnt);  
    return resultat->somme;  
} //addition
```

Processus client

```
int main (int argc, char *argv[])
{
    char *host;
    int a, b;
    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    clnt = clnt_create (host, CALCUL, VERSION_UN, "tcp");
    printf("Entier 1 : ");
    scanf("%d", &a);
    printf("Entier 2 : ");
    scanf("%d", &b);
    printf("Somme = %d\n", addition(a, b));
    exit (0);
}
```

Processus client

- Compilation et génération de l'exécutable :
 - `gcc -c calcul_client.c`
 - `gcc -o client calcul_client.o calcul_clnt.o calcul_xdr.o`
- Exécution :
 - `$./calcul_client localhost`
Entier 1 : 1
Entier 2 : 2
Somme = 3

Utilisation de RPC

- Approches à RPC traditionnelles :
 - Sun/ONC RPC, Open Network Computing, Remote Procedure Call
 - OSF DCE, Open Software Foundation – Distributed Computing Environment
 - Systèmes de gestion de bases de données : procédures stockées.

Utilisation de RPC

- Approches à RPC intégrées dans les systèmes d'objets répartis :
 - OMG CORBA, Object Management Group – Common Object Request Broker Architecture
 - SUN Java RMI, Remote Method Invocation
 - Microsoft – DCOM, Distributed Component Object Model

Utilisation de RPC

- Approches à RPC intégrées dans les systèmes de composants :
 - SUN J2EE EJB, Java 2 (Platform) Enterprise Edition – Enterprise Java Beans
 - OMG CCM, Object Management Group – Corba Component Model
 - WS-SOAP, Web Services – Simple Object Access Protocol