Tortrat-Gentilhomme Nicolas
Raymond Nicolas
Runarvot Loïc

PROJET SYSTEME

Procédures Distantes

I. PRÉSENTATION DU PROJET

Le projet se divise en trois parties.

La première consiste en un service de sérialisation, qui permet de pouvoir transformer les données à envoyer, et garder une cohérence entre le client et le serveur; et ce même en cas de changements mineurs.

La seconde partie est le serveur en lui-même, s'occupant de recevoir toutes les requêtes et de les traiter, et ce de manière parallèle.

La dernière partie, quant à elle, est le client, qui s'occupera d'envoyer des requêtes au serveur, et d'en récuperer les résultats.

Quelques tests unitaires ont été effectués sur le service de sérialisation, et le service de sauvegarde des fonctions, afin de vérifier que ces services ne comporte pas d'erreurs potentielles.

II. IMPLÉMENTATIONS

Tout le projet repose sur l'utilisation de socket UNIX locale.

1. Sérialisation

De manière plus générale, la structure de données de notre application se trouve dans le fichier u_rpc_data.c et son header. Il possède donc un ensemble de fonctions de sérialisation mais aussi de désérialisation, afin de pouvoir transcrire cette structure en un message, ainsi que le sens inverse.

La raison qui a motivé le choix d'externaliser ces fonctions par rapport au serveur et au client, était tout simplement de pouvoir utiliser les mêmes fonctions pour les deux binaires. Ainsi, lors du développement, il a été beaucoup plus simple de construire nos clients/serveurs, lorsque la (dé)sérialisation a été fonctionnelle.

2. Serveur

Le serveur va donc s'occuper de lire les requêtes, et de les traiter, de manière parallèle. C'est à dire qu'il s'occupera de plusieurs requêtes en même temps.
Pour cela, le serveur fait un simple fork avant de s'occuper de la requête. Comme il le sera expliqué dans la partie suivante, ce client est enregistré au sein d'une structure où un « Garbage Collector » opère.

Voici, dans l'ordre, les étapes effectuées par le serveur :

- accepter le client
- fork
- lire le message
- la fonction existe-t-elle ? *oui: on continue, non: erreur*
- type checking *type compatibles: on continue, sinon erreur.*
- exécution de la fonction
- réponse

Le type checking, qui est ici juste après la vérification de l'existence de la fonction, est utilisé pour vérifier à la fois que le nombre de paramètre donné est correct, mais aussi si leur type correspond; tout ceci permet de vérifier si la fonction est bel et bien exécutable.

Il ne peut, en terme général, pas y avoir d'erreur particulière durant l'exécution d'une fonction. Mais si une erreur inexplicable arrive durant l'application d'une fonction, alors le « Garbage Collector » sera capable d'annoncer l'erreur au client. *(voir 3.c)*

3. Client

Le client a plusieurs fonctionnalités possibles en utilisant le serveur.

La première est tout simplement la demande d'une liste de fonctions disponible sur le serveur, en appelant le client tel :

```
./client –list
```

La seconde est une mode pseudo-admin, où l'utilité n'est que de pouvoir éteindre le serveur. Par défaut le mot de passe est… admin. Le mode peut être utilisé grâce à la commande :

```
./client –shutdown <password>.
```

La dernière, et la plus intéressante, est tout simplement de pouvoir utiliser la ligne de commande afin de spécifier la fonction à appeler sur le serveur. Cela permet, comme on le verra dans la partie « Exemple », de pouvoir utiliser le client de manière simple à partir de n'importe quel autre programme.

Elle s'écrit de la façon suivante :

```
./client –command -ret -TYP [-TYP1 ARG1 … -TYPN ARGN]
```

Où TYP est soit void, soit int, soit str.
Le type void ne peut apparaître dans les paramètres.

4. Debug

Lors de la compilation, un mode débug peut être activé pour voir l'état de création de tous les messages, ainsi que de leur envoi.
Il suffit tout simplement de compiler avec l'option :

```
make DEBUG=1
```

5. Exemples

Afin de montrer la construction et l'utilisation du client, deux exemples ont étés construits à l'aide de deux langages différents. Ces exemples peuvent être retrouvée dans le dossier *example*.

call_client.c (simplifié par rapport au code original)

```
int p[2], r, i;
char buf[64];

char *request[] = {
    "../bin/client", "-c", "concat", "-ret", "-str", "-str", argv[1],
    "-str", argv[2], NULL
};

pipe(p);

if (fork() == 0) { /* son */
   close(p[0]);
   dup2(p[1], STDOUT_FILENO);
   execvp(request[0], request);
}

close(p[1]);
while((r = read(p[0], buf, 64)) > 0) {
   write(STDOUT_FILENO, buf, r);
}
wait(NULL);
```

callClient.ml

```
open Unix

let read_and_show fd buf =
  let length = Bytes.length buf in
  let rec aux () =
    match read fd buf 0 length with
    | -1 -> failwith "read_and_show"
    | 0  -> ()
    | n  -> ignore (write stdout buf 0 n); aux ()
  in aux ()

let _ =
  match Array.length Sys.argv with
  | 3 ->
    ()
  | _ ->
    Printf.eprintf "%s ARG1 ARG2\n" Sys.argv.(0);
    exit 1

let _ =
  let request = [|
    "../bin/client"; "-c"; "concat"; "-ret"; "-str";
    "-str"; Sys.argv.(1); "-str"; Sys.argv.(2)
  |] in
  let infd, outfd = pipe () in
  match fork () with
  | -1 ->
    failwith "fork error"
  | 0  ->
    close infd;
    dup2 outfd stdout;
    execv request.(0) request
  | n  ->
    close outfd;
    let buffer = Bytes.create 64 in
    read_and_show infd buffer;
    ignore (Unix.wait ())
```

III. FONCTIONNALITÉS SUPPLÉMENTAIRES

1. Tests unitaires

Des tests unitaires, se trouvant dans le dossier test, sont là pour tester l'intégrité de nos fonctions de (dé)sérialisation, ainsi que tout le système de fonctions.

2. Documentation interne au serveur

Le serveur est capable de générer une documentation sur l'ensemble de ses fonctions disponibles. Ainsi, lors de l'enregistrement des fonctions, il est possible d'indiquer une petite documentation, qui pourra être fournie au client si celui-ci la désire, en utilisant la commande suivante :

```
./client –list
```

3. « Garbage collector »

Le processus principal du serveur ne fait pas, lorsqu'il reçoit une requête, de double fork afin de garder une trace de tous ses enfants directs.
La volonté de garder cette trace permet de vérifier, à tout instant, si leurs exécutions ont terminé de manière normale; ou si un signal interne à l'exécution d'une fonction a été reçu par le processus fils (par exemple une SIGFPE).
Cela permet donc de pouvoir informer le client de l'état de sa requête.

4. Timeout

Si la commande que le client a demandée dure plus de 5 secondes, alors sa commande est annulée, et le serveur s'occupe d'envoyer un message de timeout au client pour lui indiquer que sa commande est invalide.

5. Générateur de code

Afin de pouvoir prendre en compte le plus grand panel de fonctions possible, un générateur de code se charge d'écrire l'exécution de fonctions pouvant aller jusqu'à 255 paramètres !
L'outil utilisé porte le nom de « pump.py », source :
https://code.google.com/p/googletest/wiki/PumpManual

6. Shutdown

Le client a la possibilité, à l'aide d'une commande particulière, de pouvoir éteindre le serveur à distance, sans avoir d'accès à la machine lançant le serveur. Pour cela, il doit aussi saisir le mot de passe du serveur.
Le mot de passe par défaut est admin.