RPC-Talk : une librairie RPC pour Smalltalk

Francis Wolinski

Informatique CDC
Direction Technique
Service Recherche Développement Techniques Avancées
wolinski@icdc.fr

Dans le cadre de l'application de traitement automatique du langage Exoseme [Landau & al 93a] [Landau & al 93b] [Vichot & Wolinski 93], nous avions besoin d'informations lexicales depuis Smalltalk. Nous disposions de Kalipsos qui est un serveur développé par le Centre Scientifique d'IBM [Fargues & al 86]. Implémenté en Prolog et implanté sur un PS, il offre différents services : analyse morphologique, analyse syntaxique, etc. Ces services sont disponibles sous la forme d'un serveur RPC qui permet à différents clients écrits en C++ de les invoquer depuis n'importe quelle station de travail du réseau.

Une solution fruste aurait consister à développer en C++ un client spécialement pour l'application et à invoquer le serveur *via* ce client. Cette technique fonctionne, mais elle s'accompagne d'un certain nombre d'inconvénients : une efficacité médiocre car le client C++ est relancé à chaque appel, un coût de développement et de maintenance de ce client dédié, et une inélégance certaine à communiquer entre programmes par des chaînes de caractères pour lesquelles il faut écrire des parseurs.

Nous avons donc cherché la disponibilité d'un RPC en Smalltalk. A notre connaissance, seul HP Distributed Smalltalk dispose d'une telle couche de communication mais elle est dédiée aux fonctionnalités propres du produit. Nous nous sommes donc résolus à implémenter le protocole RPC sous la forme d'une librairie appelée RPC-Talk. Ensuite, pour accéder aux informations linguistiques de Kalipsos, il a suffi de spécifier le service RPC correspondant et d'instancier un client Smalltalk dédié à ce service.

Après cette première utilisation de RPC, et toujours dans le cadre du projet Exoseme, nous avons développé en Smalltalk un serveur de noms propres. Ce programme détecte, catégorise et "désambiguïse" les noms propres des textes qui lui sont soumis. A l'heure actuelle il est interrogé continuellement par un programme distant et traite environ 400 dépêches quotidiennes, soit à peu près 10000 requêtes.

L'objet de ce papier est double : promouvoir la technique RPC qui est souvent méconnue et montrer l'utilité et l'utilisation de la bibliothèque que nous avons développée.

1. Qu'est-ce que le RPC ?

This tools gives you the ability to put any number of machines at the service of your application [Bloomer 91].

RPC signifie *Remote Procedure Call* soit *appel de procédure distante* La présence du logiciel correspondant sur différentes machines permet, à partir d'une application, d'invoquer les services de toute autre application, quels que soient le langage d'implémentation, le système d'exploitation ou la machine d'implantation.

Disposer du RPC permet d'ouvrir assez facilement une application à d'autres applications. Soit une l'application a besoin d'un service disponible qu'on ne souhaite pas réimplémenter (ré-utilisabilité). Soit un service est suffisamment consommateur de CPU pour qu'il soit intéressant de l'implanter sur une autre machine (efficacité). Soit un service n'est disponible que sur une machine différente éventuellement spécialisée (portabilité). Dans tous ces cas, il faut un moyen simple pour permettre à deux applications de communiquer dynamiquement.

En proposant une description de haut niveau d'un service et en liant dynamiquement des applications au travers d'une architecture bus, RPC offre des facilités de communication indépendamment des langages d'implémentation, des systèmes d'exploitation et des machines d'implantation.

Plus précisément, cette technique fournit :

- une description à un niveau conceptuel du service rendu par l'application esclave (le serveur),
- une gestion simultanée par le serveur (*multithreading*) des contextes associés à chaque application maître connectée (les clients),
- l'encodage dans une description normalisée des paramètres et des réponses des requêtes en vue de leur transport par les couches spécialisées (TCP, UDP).

Un exemple classique est fourni par les SGBD. Un SGBD est un serveur et les applications qui lui sont connectées sont les clients. Le service rendu par le SGBD permet entre autres d'enregistrer, de modifier et de supprimer des tuples. Chaque client possède un contexte propre au sein du SGBD. Ce contexte contient l'ensemble des tables et des vues relatives à une application. En principe, tout SGBD devrait être accessible par RPC. En réalité, si les principaux SGBD du commerce s'appuient sur les couches basses TCP/IP, ils ne fournissent pas d'interface RPC standard et forcent leurs utilisateurs à invoquer le serveur au travers d'interfaces spécifiques.

1.1 Comment fonctionne le RPC ?

Le RPC comprend une méthode de spécification d'un service, un système pour gérer les connexions et les sessions ainsi qu'une technique d'encodage des requêtes.

1.1.1 Service, version et procédure

RPC fournit avant tout un système de description de haut niveau d'un service. Un service est la spécification d'une application écrite pour être installée sur une machine (le serveur) et repérée sur celle-ci par un numéro de programme. Un service est souvent disponible en différentes versions qui représentent les évolutions du service et qui restent accessibles à l'utilisateur. A la manière d'une librairie, une version comprend un jeu complet de procédures qui constituent un ensemble cohérent de points d'entrée du service. Chaque procédure est décrite par des paramètres : un numéro d'identification ainsi que la spécification de ses arguments et de ses résultats tous fortement typés.

1.1.2 Connexion des clients au serveur

En pratique, la mise en œuvre s'effectue par l'intermédiaire du *portmapper*. Ce dernier est un service RPC, résident sur les machines, et gérant l'attribution des adresses logiques :

- Le client, qui souhaite se connecter à un serveur, interroge le *portmapper* de la machine serveur en lui donnant le numéro du programme souhaité ; celui-ci lui répond en renvoyant l'adresse logique à laquelle il doit se connecter.
- Une fois l'adresse logique connue, le client établit une connexion avec le serveur et, *via* ce canal de communication, il peut solliciter les différentes procédures du service.

1.1.3 Communications entre clients et serveur

La communication entre un client et un serveur est synchrone. A un instant donné, le programme client invoque une procédure RPC d'un service auquel il s'est préalablement connecté et se met en attente. La couche RPC du client code cette requête en une suite d'octets suivant une conversion normalisée. La couche TCP transporte le flot de données de la machine cliente jusqu'à la machine serveur. La couche RPC du serveur décode la requête et la transmet à la procédure idoine. Une fois la requête exécutée, son résultat est codé puis transmis à la machine cliente par la couche TCP. Une fois décodé le résultat est transmis au client en attente qui peut alors continuer son exécution.

Chaque requête et chaque réponse RPC traversent les sept couches ISO : physique, données, réseau, transport (TCP ou UDP), session (RPC), présentation (XDR) et application.

1.1.4 Installation d'un serveur sur une machine

En pratique, un serveur s'installe en se connectant à une adresse logique et en communiquant au *portmapper* son numéro de programme pour l'enregistrer au niveau de la machine. Un serveur peut gérer simultanément plusieurs clients. En général, il gère pour chacun d'eux un contexte particulier qui est utilisé à chaque appel. En cas de panne, un serveur peut sauvegarder et restaurer les différents contextes qu'il gère.

1.2 Pourquoi faire du RPC en Smalltalk ?

La technique RPC permet d'abord de faire communiquer les applications entre elles, quels que soient leurs langages d'implémentation et leurs machines d'implantation. Si Smalltalk permet de définir des primitives écrites en langage C [Goldberg & Robson 83], qui peuvent servir de liaison avec d'autres programmes, la facilité d'utilisation de RPC est d'au moins un ordre de grandeur supérieur car il n'y a aucune phase d'écriture de programme ni de compilation.

Elle permet ensuite de réaliser des applications Smalltalk distribuées et de paralléliser les différentes composantes d'une application à des fins d'efficacité. Même si Smalltalk simule une machine parallèle grâce aux processus, disposer de deux machines fonctionnant réellement en parallèle peut s'avérer utile. Il est par exemple possible de scinder une application, comprenant un système expert d'une part et un module calculatoire d'autre part, tous deux très consommateurs de CPU.

Enfin, l'utilisation de cette technique force à concevoir des applications entières comme des classes d'objets avec des interfaces bien définies L'expérience montre que tant qu'une application n'a pas été effectivement "distribuée" sa modularité peut être remise en cause. Avec RPC, les interfaces d'une application sont définies par la spécification du service rendu par celle-ci.

On peut se demander pourquoi une telle couche, apparemment si utile, n'existe pas aujourd'hui sur le marché. A notre connaissance, seul HP Distributed Smalltalk est fondé sur une couche interne RPC mais qui n'est pas accessible en dehors du cadre fourni par HP. Outre la date encore récente de la mode du client-serveur, on peut trouver plusieurs raisons à cette absence :

- Smalltalk est un environnement intégré particulièrement endocentrique ! L'environnement est conçu principalement pour

charger dans l'image tous les outils nécessaires. Le travers de ce type de langage, à la Lisp, est qu'on (re)fait tout dans le langage et qu'en général tout fonctionne sur la même machine, dans la même mémoire virtuelle, sans réel soucis de modularité ou d'efficacité...

• Smalltalk est éminemment portable : il n'y a donc peu lieu de se préoccuper de l'accès à des machines différentes ; le portage de l'application sur une machine cible s'effectue avec un coût *quasi* nul.

• Certains systèmes d'exploitation offrent des facilités qui permettent de se passer de RPC dans certains cas :

- *Dynamic Data Exchange* sur Windows et OS/2 ou bien un *pipe* sous Unix : échange dynamique de données vers une autre application tournant sur la même machine. Cette technique est limitée à des échanges unidirectionnels ; il est par exemple possible depuis Smalltalk d'envoyer des informations à un traitement de textes pour qu'il imprime des textes paramétrés.
- *Dynamic Linkage Library* sur Windows et OS/2 ou Unix : lien dynamique d'une sous-application dont il devient alors possible d'invoquer des fonctions.
- *Object Linkage Embedding* sur Windows et Mac : invocation automatique et transparente d'une application lorsque l'on manipule l'un de ses produits dans une autre application. Par exemple, un graphique intégré à un texte est directement éditable sous le traitement de textes sans repasser explicitement par le logiciel graphique.

Si ces techniques rendent bien des services, elles n'offrent pas une gestion de contexte multi-clients et multi-plates-formes comme RPC.

2. Mise en œuvre d'un client en RPC-Talk

Dans cette partie, nous expliquons comment activer puis spécifier un client en RPC-Talk. Un client est une instance de la classe `RPCClient` ou de l'une de ses sous-classes. Nous illustrons le mode client au moyen de trois exemples. Pour chaque exemple, nous montrons : l'instanciation et la connexion du client, la requête du client avec la réponse du serveur et la déconnexion du client.

Il faut bien se rendre compte que pour que cette activation soit possible il faut que le client ait une spécification préalable du service auquel il s'adresse. L'art de RPC est de fournir aux clients, qui souhaitent invoquer un service, la spécification strictement nécessaire à cette invocation. Dans notre implémentation, cette spécification réside en un objet complexe créé par une méthode d'initialisation.

En référence au §1.2 à propos des primitives en langage C, cette spécification de service se réalise en Smalltalk au moyen d'une méthode qu'il faut effectivement écrire. Il faut voir cette méthode, non pas comme un élément de programme mais comme l'initialisation d'un objet complexe.

2.1 La calculette : un exemple trivial

Dans cet exemple, on suppose l'existence d'un serveur permettant d'effectuer les opérations arithmétiques élémentaires.

2.1.1 Utilisation

On suppose le serveur mis en place sur la machine *anjou* ; création d'un client sur une machine quelconque :

```
C := RPCClient service: #Calculette host: 'anjou'
C openSession
      -> Client connected on port 2001
```

Activation du client :

```
C exec: #plus with: 1 with: 2
      -> 3
C exec: #moins with: 1 with: 2
      -> -1
```

Déconnexion du client :

```
C closeSession
```

2.1.2 Spécification

La méthode ci-dessous est une description conceptuelle d'un service de calculette réalisant additions et soustractions d'entiers non-signés. Pour des commodités propres à Smalltalk, ce service est dénommé `#Calculette` ; par ailleurs, il possède un numéro de programme donné. Ce service définit deux procédures appelées `#plus` et `#moins` qui ont respectivement les index `1` et `2`. La procédure `#plus` prend en arguments deux entiers non-signés et retourne un entier non-signé. La procédure `#moins` prend en arguments deux entiers non-signés et retourne un entier (signé).

```
!RPC class methodsFor: 'rpc initialization'!

initializeCalculette
      | service |
      service := self name: #Calculette program: 536870912.
      service
            name: #plus
            index: 1
            argsSpecs: #(#xdrUnsignedInteger #xdrUnsignedInteger )
```

```
      resultsSpecs: #xdrUnsignedInteger.
service
      name: #moins
      index: 2
      argsSpecs: #(#xdrUnsignedInteger #xdrUnsignedInteger )
      resultsSpecs: #xdrInteger
```

2.2 Le compteur : la gestion d'un contexte

Le célèbre compteur Smalltalk qui illustre toujours le modèle MVC est un bon exemple de service RPC nécessitant la gestion d'un contexte (bel exemple de ré-utilisabilité !). Chaque client, une fois connecté, dispose d'un compteur privé qu'il peut utiliser à sa guise. Le serveur a la capacité de gérer simultanément plusieurs compteurs.

2.2.1 Utilisation

On suppose le serveur mis en place sur la machine *anjou* ; création des clients sur des machines quelconques :

```
C1 := RPCClient service: #Counter host: 'anjou'
C1 openSession
      -> Client connected on port 3210
C2 := RPCClient service: #Counter host: 'anjou'
C2 openSession
      -> Client connected on port 3210
```

Activation des clients :

```
C1 value
      -> 0
C2 value
      -> 0
7 timesRepeat: [C1 exec: #incr]
C2 exec: #decr
C1 exec: #value
      -> 7
C2 exec: #value
      -> -1
```

Déconnexion des clients :

```
C1 closeSession
C2 closeSession
```

2.2.2 Spécification

Le service correspondant au compteur est spécifié par quatre procédures : incrémentation et décrémentation du compteur, mise à jour directe de sa valeur et obtention de sa valeur.

```
!RPC class methodsFor: 'rpc initialization'!

initializeCalculator
      | service |
      service := self name: #Calculator program: 20020.
      service
            name: #incr
            index: 1
            argsSpecs: nil
            resultsSpecs: nil.
      service
            name: #decr
            index: 2
            argsSpecs: nil
            resultsSpecs: nil.
      service
            name: #set
            index: 3
            argsSpecs: #xdrInteger
            resultsSpecs: #xdrInteger.
      service
            name: #value
            index: 4
            argsSpecs: nil
            resultsSpecs: #xdrInteger
```

2.3 Kalipsos : utilisation dans le cadre d'Exoseme

Pour l'application Exoseme, nous avons besoin d'informations linguistiques depuis Smalltalk et nous disposons du serveur Kalipsos [Fargues & al 86] qui est utilisé par une autre application. Une fois la technique RPC disponible en Smalltalk, il est possible d'invoquer le serveur et donc de disposer depuis Smalltalk de techniques propres au traitement automatique du langage.

Pour des raisons techniques d'encapsulation des requêtes Prolog envoyées au serveur Kalipsos, ce client est une instance de la classe `Kalipsos` sous-classe de `RPCClient`.

2.3.1 Utilisation

Notre application effectue une reconnaissance semi-automatique des noms propres d'un texte écrit en français. Un certain nombre de formes de surface de noms propres sont répertoriées ; elles s'appuient sur du vocabulaire spécifique (prénoms, formes juridiques de sociétés, ...).

L'intérêt de Kalipsos réside dans l'utilisation de ses lexiques qui donne les caractéristiques grammaticales de chaque mot sous la forme d'un tableau de propriétés. Par exemple pour le mot *table*, Kalipsos rend *substantif* ou *verbe*. Différentes méthodes de la classe `Phrase` font appel à ce service ; en particulier une méthode détermine si le mot courant est français et une autre vérifie que c'est un nom commun :

```
!Phrase methodsFor: 'rpc'!

close
     kalipsos closeSession

open
      kalipsos := Kalipsos service: #Kalipsos host:'ps'.
      kalipsos openSession

!Phrase methodsFor: 'request'!

motCourantFrancais
      ^(kalipsos morpheur: (mots at: index)) isEmpty not

motCourantNomCommun
      ^(kalipsos morpheur: (mots at: index)) includes: 'substantif'

!Phrase class methodsFor: 'creation'!
new
      ^super new open
```

2.3.2 Spécification

Le service correspondant au serveur Kalipsos définit entre autres une procédure appelée `#goal`. Cette procédure prend en entrée un tableau composé d'un entier et d'une chaîne de caractères. Elle renvoie un tableau composé d'un entier non-signé et d'une chaîne de caractères. Ces deux tableaux ont été définis sous la forme de structures de données dénommées respectivement `'KLP_GOAL'` et `'KLP_RETURN_GOAL'` (Cf. §4 étape 3).

On remarque également la définition d'un *timeout* de 120 secondes : au-delà de ce temps imparti, le client connecté au serveur Kalipsos générera une erreur signalant probablement un problème au niveau du serveur (désactivation, panne, ...).

```
!RPC class methodsFor: 'rpc initialization'!

initializeCalculator
      | service |
      service := self name: #KLP program: 536871065.
      service withTimeout: 120.
      service struct: 'KLP_GOAL'
            desc: #(#xdrInteger #xdrString).
      service struct: 'KLP_RETURN_GOAL'
            desc: #(#xdrUnsignedInteger #xdrString).
      (...)
      service
            name: #goal
            index: 3
            argsSpecs: 'KLP_GOAL'
            resultsSpecs: 'KLP_RETURN_GOAL'
```

3. Mise en œuvre d'un serveur en RPC-Talk

Nous avons vu dans la partie précédente comment s'activait un client en RPC-Talk et quelle était pour un client la spécification nécessaire pour accéder à un service distant. Il est également possible d'utiliser RPC-Talk pour implémenter un serveur ; c'est-à-dire un programme résident sur une machine, à l'écoute des invocations de clients et prêt à leur répondre. Un serveur est une instance de la classe `RPCServer`.

A titre d'illustration, nous reprenons ci-dessous les trois exemples précédents du point de vue du serveur. Pour ces exemples, nous montrons la mise en place du serveur, sa désinstallation et la spécification du service correspondant.

D'une manière générale cette spécification vient compléter celle établie pour le mode client car des paramètres propres au serveur sont effectivement introduits (bloc ou sélecteur d'exécution, contexte, ...). En pratique, pour les applications RPC faisant communiquer deux images Smalltalk, le serveur et les clients peuvent partager une spécification unique : celle du serveur. On ne s'étonnera donc pas de voir réutilisés les sélecteurs des méthodes d'initialisation.

3.1 La calculette

Une calculette réalisant des opérations arithmétiques est l'exemple le plus simple d'un service qui peut être implémenté en RPC. Ce service élémentaire ne gère pas de contexte ; chaque invocation n'a donc aucun effet de bord.

3.1.1 Mise en place

Mise en place du serveur sur la machine *anjou* :

```
S := RPCServer service: #Calculator
S installServer
      -> Server installed on port 2001
```

Désinstallation du serveur :

```
S desinstallServer
      -> Server desinstalled
```

3.1.2 Spécification

Dans le cas d'un serveur, la définition du §2.1.2 est simplement complétée par la donnée du mécanisme d'exécution qui est dans ce cas un bloc Smalltalk. Ce bloc possède autant d'arguments que ceux spécifiés pour chaque procédure.

```
!RPC class methodsFor: 'rpc initialization'!

initializeCalculator
      | service |
      service := self name: #Calculator program: 536870912.
      service
            name: #plus
            index: 1
            argsSpecs: #(#xdrUnsignedInteger #xdrUnsignedInteger )
            resultsSpecs: #xdrUnsignedInteger
            exec: [:x :y | x + y].
      service
            name: #moins
            index: 2
            argsSpecs: #(#xdrUnsignedInteger #xdrUnsignedInteger )
            resultsSpecs: #xdrInteger
            exec: [:x :y | x - y]
```

3.2 Le compteur

La gestion d'un serveur de compteurs en Smalltalk présuppose l'existence d'une classe de compteur ayant défini les méthodes exposées au §2.2.2.

3.2.1 Mise en place

Mise en place du serveur sur la machine *anjou* :

```
S := RPCServer service: #Compteur
S installServer
      -> Server installed on port 3210
```

Désinstallation du serveur :

```
S desinstallServer
      -> Server desinstalled
```

3.2.2 Spécification

Par rapport à la calculette, la définition du service diffère sur deux points. Tout d'abord, le contexte est "passé" au serveur à sa mise en place par l'intermédiaire d'un bloc (voir l'envoi de message `service withSession: [Counter new]`). Ensuite, le mécanisme d'exécution est indiqué non pas par un bloc comme dans l'exemple de la calculette mais par un sélecteur. Ce dernier correspond à une méthode implémentée dans la classe du contexte,

en l'occurrence le compteur, et possède autant de mots-clés que d'arguments spécifiés pour la procédure. Dans notre exemple, la classe `Counter` implémente les quatre méthodes : `#incr`, `#decr`, `#set:` et `#value`.

```
!RPC class methodsFor: 'rpc initialization'!

initializeCompteur
      | service |
      service := self name: #Compteur program: 20020.
      service withSession: [Counter new].
      service
            name: #incr
            index: 1
            argsSpecs: nil
            resultsSpecs: nil
            exec: #incr.
      service
            name: #decr
            index: 2
            argsSpecs: nil
            resultsSpecs: nil
            exec: #decr.
      service
            name: #set
            index: 3
            argsSpecs: #xdrInteger
            resultsSpecs: #xdrInteger
            exec: #set:.
      service
            name: #value
            index: 4
            argsSpecs: nil
            resultsSpecs: #xdrInteger
            exec: #value
```

3.3 Kalipsos

Kalipsos est un serveur complexe écrit en Prolog ; il n'est donc pas question de donner des éléments concernant Smalltalk dans cette partie. Toutefois, à titre de comparaison, nous donnons un extrait du fichier de spécification original (d'extension `x`). L'utilitaire *rpcgen* est un générateur de code qui, à partir d'un tel fichier et d'options Unix savamment dosées, permet de générer un code en langage C correspondant à un client ou à un serveur [Bloomer 91]. Ce code peut ensuite être compilé et lié au programme de l'application.

Fourni avec le serveur, ce fichier nous a servi de base pour spécifier ce service. On reconnaîtra l'essentiel de la description du §2.3 sous une forme évidemment proche du langage C. Les éléments supplémentaires sont nécessaires au serveur tandis que les éléments manquants sont en fait les paramètres introduits lors de l'exécution de *rpcgen*.

```
struct KLP_GOAL (
      int session_id;
      char *goal; );
struct KLP_RETURN_GOAL (
      unsigned short rc;
      char *answer; );
(...)
program KPROGNUM
{
      version KPROGVERS
      {
      (...)
      struct KLP_RETURN_GOAL klp_goal_eval(KLP_GOAL) = 3;
      } = 1;
} = 0x20000099;
```

Là où C++ nécessite un précompilateur générant du code en langage C devant à son tour être effectivement compilé, Smalltalk n'a besoin que d'une méthode générant une instance qui gère la spécificité de chaque service.

4. Spécification d'un service en RPC-Talk

Nous avons vu que les clients (resp. serveurs) nécessitaient une spécification du service qu'ils invoquent (resp. rendent). Cette spécification est représentée par un objet dont la structure permet au client comme au serveur de fonctionner.

Ce type d'objet constitue la spécification d'un service invoqué ou rendu. Il offre une description conceptuelle d'un service (assez semblable à la spécification d'une classe en C++). Une spécification identique doit naturellement être utilisée par le serveur et ses clients.

RPC est *a priori* défini pour des langages statiquement typés. Il assure le transport des types supportant une représentation XDR qui convertit les objets typés en une séquence d'octets bien définie. Nous avons implémenté les conversions des types XDR de base, par convention ceux-ci sont désignés par un symbole Smalltalk (Cf. §4.2).

Un service est utilisé dynamiquement aussi bien par le client que par le serveur. Le client l'utilise pour coder les arguments des procédures et pour en décoder les résultats. Inversement, le serveur l'utilise pour décoder les arguments des procédures et pour en coder les résultats.

4.1 Spécification d'un service

Nous avons vu que la définition d'un service se faisait par une méthode d'initialisation dédiée qui instancie le service et l'initialise. Cette écriture s'effectue en quatre étapes :

1. Définition du nouveau service répertorié par son nom et par un numéro de programme. Le nom du service est un symbole Smalltalk qui permet de le désigner aisément au sein de l'image ; le numéro de programme est utilisé pour l'identifier au niveau de la machine.
2. Donnée éventuelle de paramètres spécifiques au client ou au serveur :
   - Le contexte géré par le serveur par la donnée d'un bloc de création. A la connexion d'un nouveau client l'exécution de ce bloc rend le contexte du client.
   - La durée maximale d'attente du client (*timeout* exprimé en secondes). Cette durée limite l'attente de réponse de la part du client en cas de panne du serveur.
3. Chaque version est numérotée implicitement à partir de `1`. Le numéro de version est une convention RPC qui permet à chaque application d'offrir simultanément des niveaux de version différents. Pour chaque version, définitions éventuelles de structures de données qui sont utilisées pour passer des paramètres complexes. Une structure se définit par :
   - Un nom de structure (une chaîne de caractères).
   - Une définition de structure (un tableau de types prédéfinis ou de sous-structures).
4. Pour chaque version, définition des différentes procédures par la donnée de :
   - Son nom : il s'agit d'un symbole Smalltalk qui permet de désigner aisément la procédure au sein du service.
   - Son numéro : `1` et plus, `0` étant réservé à la procédure `#ping` présente dans chaque service,*i.e.* pas d'argument et `nil` en retour. Le numéro des procédures est un identifiant RPC qui permet de désigner chaque procédure au sein d'un service.
   - La spécification de ses arguments :
     . `nil` : procédure sans argument,
     . un symbole Smalltalk : procédure à un seul argument,
     . un tableau de types prédéfinis : procédure à arguments multiples,
     . une chaîne de caractères : procédure dont l'argument est défini par une structure de données.
   - La spécification des résultats : *idem*.
   - Uniquement pour un serveur, la spécification de l'exécution : un bloc Smalltalk ou un sélecteur de méthode s'il y a un contexte. Le bloc d'exécution (seul) ou le sélecteur (avec le

contexte) permet au serveur de réaliser l'exécution requise par le client (avec passage de paramètres).

4.2 Types XDR disponibles

La technique RPC est fondée sur une norme de codage de certains types. Cette norme appelée XDR (pour eXternal Data Representation) permet de transposer de nombreux types en une série d'octets[1]. Les conversions courantes se font sur quatre octets mais certaines se font sur un nombre d'octets variable (toujours multiple de quatre). Les types supportés par XDR et implémentés dans RPC-Talk sont désignés par des symboles Smalltalk.

Les types XDR

Les types XDR de base sont implémentés dans RPC-Talk :

- `#xdrInteger` : entiers signés (en complémentation à deux)
- `#xdrUnsignedInteger` : entiers non-signés
- `#xdrFloat` : flottants
- `#xdrDouble` : doubles
- `#xdrBool` : booléens
- `#xdrEnum` : énumérations (un entier non-signé)
- `#xdrOpaque` : opaques (tableau d'octets quelconques)
- `#xdrString` : chaînes de caractères

Certains objets Smalltalk courants

Pour les transfert d'image Smalltalk à image Smalltalk uniquement, certains types ont été ajoutés. Ils correspondent à des classes courantes :

- `#xdrDate`
- `#xdrTime`
- `#xdrCharacter`
- `#xdrSymbol`
- `#xdrIntegerPoint`[2]

Enfin toujours d'image Smalltalk à image Smalltalk, pour le transfert d'objets plus complexes qui supportent le Binary Object Storage System (BOSS), il est possible d'utiliser le type général `#xdrBOSS`.

---

[1] En fait, il existe différentes normes de RPC. Nous nous sommes appuyé sur la norme ONC (Open Network Communication) qui utilise l'encodage XDR. D'autres normes utilisent d'autres types d'encodages...

[2] Un point Smalltalk est composé de deux coordonnées qui peuvent être des entiers, des flottants, des doubles, etc. Or la classe des coordonnées d'un point détermine le type de conversion XDR pour l'encodage à utiliser. C'est pourquoi, XDR nécessite de typer les points Smalltalk, ou tout au moins, de décider *a priori* (ce qui est assez rare dans ce langage) quelle sera la classe des coordonnées des points transférés.

5. Conclusion

Depuis, RPC-Talk a été utilisé au LAFORIA dans le cadre du projet SAGESSE [Caetano 94] pour le système SECIV de formation assistée par ordinateur pour apprendre gérer des scénarios de crise civiles. RPC-Talk a permis de séparer le poste du formateur du poste de l'élève en installant un serveur et un client sur chacun des postes. Le formateur, depuis son poste, définit des scénarios, les transmet à l'élève sous forme d'événements. L'élève, depuis son poste, réagit au scénario et retourne ses actions au formateur. Dans ce projet, RPC est uniquement utilisé pour transférer des informations d'un poste à un autre. Les serveurs installés sont réceptifs aux événements envoyés, ils les répercutent simplement sur les applications locales sans retourner de réponse particulière.

RPC-Talk a également été connecté à Epitalk [Pachet 94] [Pachet & al 95] à divers titres. D'abord, pour implémenter des *proxies* comme des espions particuliers. Un *proxy* est un objet spécial présent dans une image et représentant un objet actif dans une autre image. Ensuite, pour faire du conseil à distance : les conseillers épiphytes [Pachet & al 94] sont lancés sur une machine différente de l'application espionnée et ne consomment donc pas la CPU de l'application. Et enfin, pour définir des applications maître-esclave dans lesquelles une action effectuée dans une image est immédiatement répercutée sur l'autre (en fait, ce qui a été fait pour le système SECIV). Ces connexions ont nécessité la définition d'un service muet dans lequel l'*espion-proxy* transmet les interactions au conseiller sans attendre de réponse.

Ces premières réalisations montrent l'indéniable intérêt que présentent, d'une part, cette technique pour réaliser facilement des systèmes distribués, et d'autre part, la maîtrise de cette technique pour l'adapter aux problèmes particuliers qui se présentent.

Bibliographie

[Bloomer 91] Bloomer J., *Power Programming with RPC*, O'Reilly & Associates, Inc (1991)

[Caetano 94] Caetano H., *Représentation des connaissances dans le système informatisé de formation: Intempéries*, Colloque "Patrimoine culturel et formation" Ravello, Italie (1994)

[Fargues & al 86] Fargues J., Landau M.-C., Dugourd A., Catach L., *Conceptual Graphs for Semantics and Knowledge Processing*, IBM Journal of Research & Development (1986)

[Goldberg & Robson 83] Goldberg A., Robson D. *Smalltalk-80 : the language and its implementation*, Addison Wesley (1984)

[Landau & al 93a] Landau M.-C., Sillion F., Vichot F., *Exoseme : A Thematic Document Filtering System*, Intelligence Artificielle, Avignon (1993)

[Landau & al 93b] Landau M.-C., Sillion F., Vichot F., *Exoseme : A Document Filtering System Based on Conceptual Graphs*, Theory and Applications, International Conference on Conceptual Structures (ICCS), pp.171-187, Québec (1993)

[Pachet 94] Pachet F., *A spying mechanism in Smalltalk-80*. Laforia internal report, Given as a tutorial at the second European Smalltalk User Group (ESUG), summer school, Cork, Ireland (1994)

[Pachet & al 94] Pachet F., Giroux S., Paquette G. *Pluggable Advisors as Epiphyte Systems*, Computer Aided Learning in Science and Engineering (CALISCE), pp. 167-174, Paris (1994)

[Pachet & al 95] Pachet, F., Wolinski F., Giroux, S. *Spying as a Programming Paradigm*, Technology of Object-Oriented Languages and Systems (TOOLS), à paraître, Versailles (1995)

[Vichot & Wolinski 93] Vichot F., Wolinski F. *Exoseme : une application du traitement automatique du langage*, Informatique CDC, Rapport Interne (1993)

ERROR: undefined
OFFENDING COMMAND:

STACK: