Communication synchrone entre programmes par RPC et RMI

par **Michel RIVEILL**
Professeur à l'université de Nice - Sophia-Antipolis

Roland BALTER
Professeur à l'université Joseph-Fournier, Grenoble, Laboratoire SIRAC

et **Fabienne BOYER**
Maître de conférences à l'université Joseph-Fournier, Grenoble, Laboratoire SIRAC

1.	**Modèle client-serveur**	H 2 738 - 2
1.1	Description	— 2
1.2	Mise en œuvre : appel de procédure à distance	— 4
1.3	Exemples d'environnements à base de RPC	— 5
2.	**Environnement de développement pour RPC**	— 5
2.1	Langages de description d'interface	— 5
2.2	Exemple de description dans l'IDL de Corba	— 6
2.3	Outils de génération	— 6
3.	**Support système pour l'appel de procédure à distance**	— 6
3.1	Sémantique du RPC	— 7
3.2	Passage de paramètres	— 7
3.3	Désignation et liaison	— 8
	3.3.1 Principe	— 8
	3.3.2 Serveur de désignation	— 8
	3.3.3 Mise en œuvre	— 8
3.4	Appel statique et appel dynamique	— 9
3.5	Hétérogénéité	— 9
	3.5.1 Codage des données par les processeurs	— 9
	3.5.2 Codage des données par les applications	— 10
4.	**Mise en œuvre de l'appel de procédure à distance**	— 10
4.1	Envoi de messages	— 10
	4.1.1 Client-serveur en mode non connecté	— 10
	4.1.2 Client-serveur en mode connecté	— 12
	4.1.3 Éléments de choix	— 12
4.2	Appel de procédure avec langage de description d'interface	— 12
	4.2.1 Générateur RPCgen	— 12
	4.2.2 Exemple de réalisation	— 12
	4.2.3 Évaluation	— 14
4.3	Appel de méthode à distance : RPC à objet	— 14
	4.3.1 Java RMI	— 15
	4.3.2 Exemple	— 16
	4.3.3 Évaluation	— 16
5.	**Performances des RPC**	— 17
6.	**Conclusion**	— 18
	Références bibliographiques	— 19

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

Il existe plusieurs modèles d'organisation d'une application répartie. Citons, entre autres, les modèles à base d'échange de messages ou d'événements/réactions qui s'adaptent bien à des communications asynchrones ; le modèle client-serveur qui s'appuie sur une abstraction linguistique bien connue, l'appel de procédure synchrone ; les modèles utilisant la mobilité du code, par exemple des systèmes d'agents mobiles ; ou encore des modèles à objets répartis qui donnent au concepteur d'applications l'illusion d'une mémoire partagée d'objets distribués. Dans cet article, nous nous intéressons principalement au modèle client-serveur car il est aujourd'hui le plus répandu dans les produits industriels. Nous verrons cependant que la frontière entre les divers modèles d'organisation des applications réparties n'est pas étanche et que les applications réparties construites selon le modèle client-serveur empruntent assez souvent des propriétés et des mécanismes propres à d'autres modèles.

*L'article développe le modèle client-serveur selon deux axes : un axe « environnement de développement » qui présente les outils de construction d'applications, en particulier l'**appel de procédure à distance** ; un axe « système » qui présente les principes de mise en œuvre de l'appel de procédure à distance dans un environnement distribué hétérogène. Ces services systèmes sont généralement regroupés dans une couche de logiciel interposée entre l'application et le système d'exploitation, habituellement désignée par le terme générique de « middleware ».*

*Le modèle client-serveur de base met en jeu un processus **client**, qui demande l'exécution d'un service, et un processus **serveur**, qui réalise ce **service**. Client et serveur sont localisés sur deux machines reliées par un réseau de communication. Ce modèle a été introduit pour mettre en œuvre les premières applications réparties (transfert de fichiers, connexion à une machine distante, courrier électronique, etc.), réalisées chacune par un protocole applicatif spécifique. Dans une seconde étape, une construction commune, l'appel de procédure à distance, a été introduite pour fournir un outil général pour la programmation d'applications client-serveur.*

Nous avons volontairement regroupé dans cet article les deux modèles de communications synchrones existants, liant un processus client et un processus serveur : le modèle issu de la programmation procédurale permettant de réaliser des appels de procédure (ou de service) à distance (RPC : Remote Procedure Call) et son adaptation aux langages à objets qui permet de réaliser des appels de méthode à distance (RMI : Remote Method Invocation). Ces deux modèles utilisent des fondements communs et le second est une évolution naturelle du premier, imposée par le développement des langages à objets.

1. Modèle client-serveur

1.1 Description

Dans ce modèle, le dialogue entre le client et le serveur se fait par échange de messages plutôt que par mémoire partagée. Le modèle client-serveur garantit la protection mutuelle du client et du serveur par la séparation de leurs espaces d'adressage. Il permet également de localiser dans un serveur une fonction partagée par plusieurs clients.

Pour le client, un serveur se présente sous la forme d'une boîte noire sur la mise en œuvre de laquelle il ne possède pas d'information. Par contre, les services que rend le serveur sont connus du client par leur nom, les paramètres à fournir et les paramètres qui lui seront rendus après exécution du service. Le dialogue avec le serveur est à l'initiative du client. Il est réalisé par échange de deux messages (figure **1**) : le premier transmet une requête d'exécution du service chez le serveur en donnant le nom du service souhaité et les paramètres associés, le second est envoyé par le serveur et contient le résultat de l'exécution du service.

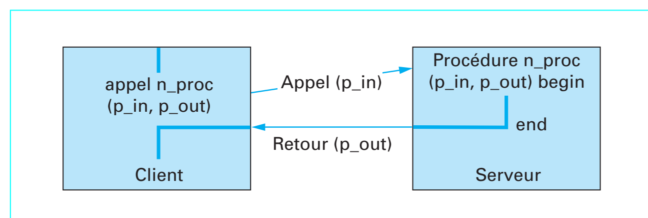


Figure 1 – Principe du modèle client-serveur

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

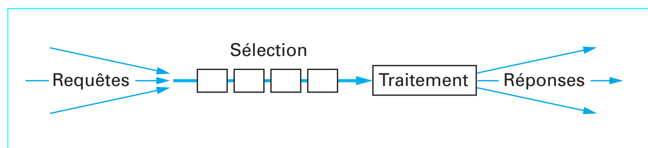


Figure 2 – Principe de réalisation d'un serveur

Il est important de noter que la terminologie client-serveur fait référence à un point de vue d'implémentation. D'un strict point de vue fonctionnel, il faudrait plutôt parler de modèle client-service puisque c'est le service qui constitue l'entité d'accès depuis un programme client. Le serveur représente l'environnement d'exécution qui permet de mettre en œuvre un ensemble de services.

La structure du serveur peut prendre plusieurs formes. Le schéma classique d'organisation du serveur est celui d'un processus cyclique qui a trois tâches essentielles (figure **2**) :

— recevoir, trier et conserver les requêtes avant leur exécution ;
— extraire une requête et exécuter le service demandé ;
— envoyer la réponse au client.

Nous avons supposé que le serveur peut exécuter plusieurs types de services, identifiés par un nom service-id différent (par exemple, pour un serveur de fichier : lire_fichier, écrire_fichier, imprimer_catalogue, etc.).

L'exécution proprement dite des services peut être réalisée de façon séquentielle. On parle alors de serveur « séquentiel » (figure ***3a***). Elle peut être sous-traitée à des flots d'exécution indépendants, processus ou processus léger (thread). On parle alors de serveur « multiple » ou encore de serveur « multiprogrammé » (figure ***3b***). Ce dernier schéma de réalisation accroît le parallélisme de traitement des requêtes issues des clients mais impose une synchronisation des différents flots d'exécution lors de l'accès aux données communes du serveur.

```
while true do
   begin
      Receive (client, message);
      Extract (message, service_id, <params>);
         Case service_id of
         service_1 : begin
            do_service [service_1]
                           (<params>, results);
            end
         ...
         autre_service : begin
            do_service [autre_service]
                          (<params>, results);
            end
         ...
      end
      Send (client, results);
   End
```

Lorsque le serveur peut servir plusieurs clients, il est intéressant de l'organiser comme une famille de processus coopérants pour permettre l'exécution concurrente de plusieurs requêtes et exploiter ainsi un multiprocessus ou des entrées-sorties simultanées. Le schéma classique comporte un processus cyclique, le **veilleur** (*daemon*), qui attend les demandes des clients. Lorsqu'une demande arrive, le veilleur active un processus exécutant qui réalise le travail demandé. Les exécutants peuvent être créés à l'avance et constituer un pool fixe ou être créés à la demande par le veilleur en utilisant des processus ou des processus légers (threads). Ce dernier schéma est illustré ci-après :

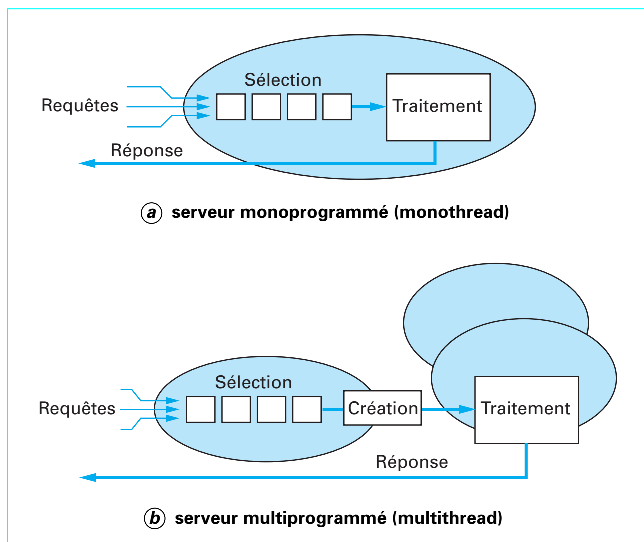


Figure 3 – Types de serveurs

```
while true do      -- processus veilleur
begin
   receive (client, message);
   extract (message, service-id, <params>);
   p := create_processus (client,
                     service_id, <params>);
end ;
```

```
Processus p:      -- processus exécutant
Begin
   do_service [service_id](<params>, results);
   send (client, results);
   exit;      -- autodestruction
end
```

Notons qu'il n'y a pas identité entre la notion de serveur et la notion de site. Plusieurs serveurs peuvent coexister sur un même site. Un serveur peut aussi être réparti sur plusieurs sites, pour augmenter sa disponibilité (par le principe de redondance) ou ses performances (en donnant ainsi la possibilité de réaliser du partage de charge entre plusieurs instances d'un serveur).

On distingue différents types de serveurs selon qu'ils gèrent des données persistantes ou non, ou selon qu'ils conservent l'état du dialogue avec un client ou non.

■ Un *serveur qui ne gère pas de données persistantes* est un serveur qui fournit un résultat calculé uniquement en fonction des paramètres d'appel du service demandé. Cette solution est très favorable en ce qui concerne la tolérance aux pannes ou le contrôle de la concurrence puisque le concepteur du serveur n'a pas à gérer ces problèmes. Un exemple type d'un tel service est le calcul d'une fonction scientifique. À l'opposé, le *serveur avec données persistantes* gère des données qui peuvent être modifiées par l'exécution des différents services. Il est alors nécessaire d'une part de contrôler les accès concurrents à ces données partagées pour un serveur « multiprogrammé », et d'autre part de mettre en place une politique de reprise après panne pour assurer la cohérence de ces données. Un exemple type d'un tel serveur est un objet serveur (au sens de la programmation objet) dont l'état est modifié par l'exécution de ses différentes méthodes.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

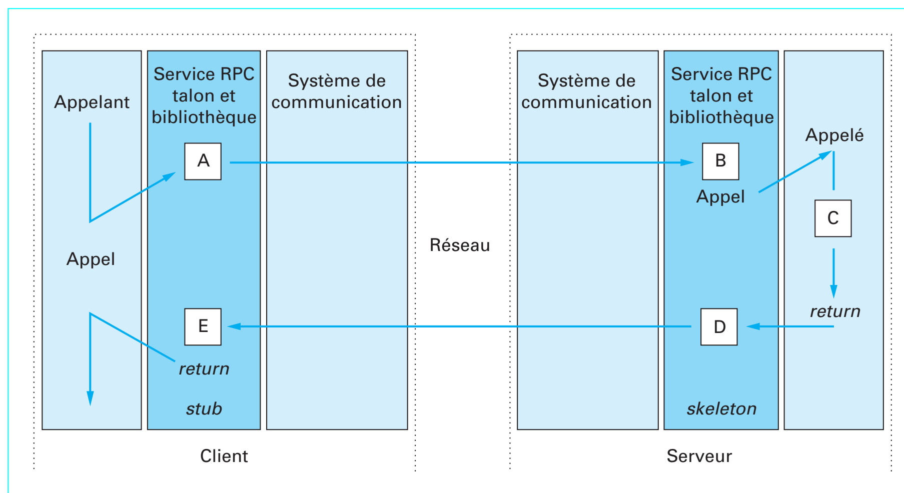


Figure 4 – Appel de procédure

Dans un *serveur sans état*, les requêtes émises par un même client s'exécutent sans lien. Il peut y avoir modification de données globales mais le serveur ne garde aucune trace des opérations réalisées par un client (en particulier, l'ordre d'exécution des requêtes n'a pas d'importance). C'est le cas par exemple dans un serveur de fichiers avec accès direct. Cependant, dans un certain nombre de cas, il est important que le serveur conserve des informations sur les requêtes antérieures, par exemple lorsque l'ordre des appels est un paramètre du service demandé. Un serveur de fichiers permettant l'accès séquentiel en est un exemple (dans ce cas, l'identificateur de l'enregistrement courant est un élément du contexte d'exécution du client sur le serveur).

1.2 Mise en œuvre : appel de procédure à distance

L'appel de procédure à distance [1] est un mécanisme qui permet à un processus p, implanté sur un site C, d'exécuter une procédure P sur un site distant S, avec un effet globalement identique à celui qui serait obtenu par l'exécution locale de P sur C. Son intérêt est d'étendre à un système réparti une construction dont la sémantique est bien connue : l'appel de procédure. Comme nous le verrons au paragraphe 3, l'objectif d'identité entre appel de procédure à distance et appel de procédure local (que l'on présente souvent sous la propriété de « transparence de la distribution ») est difficile, voire impossible, à atteindre pour des raisons techniques liées précisément à la distribution du contrôle et des données. La réalisation repose sur l'utilisation d'un processus s (ou serveur), qui exécute la procédure P sur le site S. Le processus client p reste bloqué pendant l'exécution de P et il est réactivé au retour de P. On introduit deux procédures (figure **4**), appelées **talons**. Le talon client (*stub*), sur le site appelant, fournit au processus client une interface identique à celle de la procédure appelée ; de même, le talon serveur (*skeleton*), sur le site appelé, réalise pour le processus serveur une interface identique à celle d'un appel local. Chacun des talons, lié au programme du processus client ou serveur sur le site correspondant, fonctionne aussi comme un représentant de l'autre processus (client ou serveur) sur ce site.

En l'absence de défaillances, les fonctions du talon client, exécutées par le processus appelant, sont les suivantes :

— **au point A** : mettre les paramètres d'appel sous une forme permettant leur transport sur le réseau vers le site appelé ; c'est la fonction d'**emballage** (*marshalling*) ; générer un identificateur pour la requête en cours, armer un délai de garde et envoyer vers le site appelé un message contenant l'identité de la procédure appelée et les paramètres d'appel ; mettre en attente le processus client, en attendant la réponse du site appelé ;

— **au point E** : lorsque la réponse arrive, le processus client est réveillé ; il poursuit son exécution dans le talon client en extrayant du message de réponse les résultats, s'il y en a, pour les transformer dans une forme compréhensible par le site local : c'est la fonction de **déballage** (*unmarshalling*) ; désarmer le délai de garde et acquitter le message de réponse ; exécuter le retour de procédure, avec transmission des résultats ; tout se passe alors, pour le processus client, comme pour le retour d'un appel de procédure local.

Sur le site appelé, le processus associé à l'exécution de la requête appelle le talon serveur et exécute les fonctions suivantes :

— **au point B** : à partir du message reçu du client, enregistrer l'identificateur de la requête, déterminer la procédure appelée et déballer les paramètres d'appel ;

— **au point C** : exécuter la procédure appelée en lui passant les paramètres (il s'agit alors d'un appel de procédure local) ;

— **au point D** : au retour de cet appel, l'exécution se poursuit dans le talon serveur, pour préparer le message de retour vers le site appelant, en emballant les éventuels résultats ; armer un délai de garde et envoyer le message de retour au site appelant ; attendre l'accusé de réception du message de réponse, désarmer le délai de garde puis terminer l'exécution soit par autodestruction si le processus a été créé à la demande, soit par blocage s'il s'agit d'un processus pris dans un pool.

Le rôle des **talons** est de factoriser le code qui ne dépend que du service à exécuter, en particulier la réalisation de programmes d'emballage et de déballage qui nécessitent que soit défini un format standard pour la représentation des paramètres.

Le rôle de la **bibliothèque de RPC** est de factoriser le code qui est commun à tous les appels de procédure à distance et en particulier le code nécessaire au traitement des erreurs. Ce dernier est lié à la sémantique choisie pour le RPC (§ 3.1).

Les programmeurs d'applications réparties disposent d'outils pour réaliser l'appel de procédure à distance sans avoir à connaître le détail des mécanismes précédemment décrits. Ces outils reposent sur une spécification des interfaces dans un langage approprié, indépendant des langages de programmation, qui définit pour chaque procédure susceptible d'être appelée à distance, une interface qui comporte le nom de la procédure et, pour chaque paramètre, son type et son sens de transmission (argument ou résultat). Un langage de spécification d'interfaces commun à plusieurs lan-

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

gages de programmation permet la communication entre des programmes écrits dans ces différents langages. La description d'un service à l'aide du langage de spécification d'interface constitue un contrat entre le client et le serveur.

Il existe des **générateurs** (ou **compilateurs**) **de talons** qui, à partir de la spécification d'interfaces de procédures destinées à être appelées à distance, produisent les talons client et serveur nécessaires. Un compilateur de talons engendre les procédures d'emballage et de déballage en fonction de la description des paramètres et réalise la liaison entre le programme client et la procédure appelée.

1.3 Exemples d'environnements à base de RPC

Les environnements à base de RPC [2] sont aujourd'hui nombreux. Quelques environnements marquants sont cités ici.

- Le **RPC de Sun/ONC** a été développé initialement pour la mise en œuvre du système de fichiers répartis NFS [3]. Ce RPC, facile d'emploi, permet essentiellement de construire des appels de procédure entre stations Unix avec des programmes client et serveur écrits dans le langage C. Pendant très longtemps, ce RPC a servi de référence pour l'écriture d'applications réparties selon le modèle client-serveur.

- L'**OSF** (Open Software Foundation) a défini un RPC qui présente de nombreuses similitudes avec celui de Sun. La contribution de l'OSF porte plus précisément sur l'environnement DCE (Distributed Computing Environment) [4] [5] qui fournit un ensemble de services distribués de base nécessaires pour la mise en œuvre d'applications réelles à grande échelle : service de désignation, service de gestion du temps, services de sécurité, service de gestion de fichiers répartis, etc. Tous ces services sont construits eux-mêmes en utilisant le RPC.

- L'**OMG** (Object Management Group) [6], au travers de la spécification d'architecture Corba [7], a apporté deux contributions significatives au modèle client-serveur. La première concerne l'utilisation des propriétés bien connues de l'objet (modularité, encapsulation, typage, héritage, etc.) pour décrire les services accessibles à distance (§ 4.3). La seconde contribution est le rôle clé donné au langage de définition d'interface (IDL) comme format pivot de représentation du contrat entre le client et le serveur (§ 2.2). Ce langage permet l'écriture d'applications client-serveur multilangages.

- L'**environnement DCOM** [8] [9], disponible sur les diverses variations du système Windows de Microsoft, offre une fonction similaire pour écrire des applications à base d'objets/composants COM interconnectés selon un modèle client-serveur. Le RPC disponible dans Windows est une évolution du RPC de l'environnement DCE [4].

- **Java RMI** [10] a intégré au sein de la programmation objet la notion d'appel de procédure à distance. Il permet l'écriture d'applications client-serveur en Java avec la possibilité de charger dynamiquement le code du talon client. Cette facilité permet à un client de se lier au serveur uniquement lors de l'appel.

- **HTTP** [11] est un protocole client-serveur utilisé, entre autres, par les navigateurs des stations clientes pour dialoguer avec les serveurs Web à l'échelle de la planète. Dans sa version de base, HTTP permet l'accès distant à quatre services pour consulter ou modifier des données gérées par les serveurs. La consultation d'informations sur le Web est sans aucun doute l'application client-serveur la plus utilisée aujourd'hui.

2. Environnement de développement pour RPC

Ce paragraphe est consacré à la programmation d'une application répartie selon le modèle client-serveur. Cela inclut bien entendu la programmation du client, la programmation du serveur et la programmation des échanges entre le programme client et le programme serveur. L'idéal serait de programmer tous ces aspects à l'aide d'un seul langage de programmation intégrant la distribution. Il y a eu de multiples tentatives pour définir un tel langage, en particulier en s'appuyant sur des langages à objets existants (C++ et Eiffel entre autres), qui se sont traduites par des prototypes expérimentaux.

Cette approche présente de nombreux avantages car, du fait de l'intégration de la distribution dans les concepts du langage, le traitement de la distribution devient élémentaire (voire inexistant) pour le programmeur d'applications. Le compilateur du langage génère directement le code des talons sans qu'il soit besoin de faire appel à un IDL. Par ailleurs, les bibliothèques associées au langage fournissent les mécanismes systèmes nécessaires à la mise en œuvre des appels distants.

En réalité, cette approche n'a pas connu le succès escompté pour deux raisons : l'obligation d'utiliser un langage donné (dans une forme non normalisée en raison des extensions dues à la distribution) et la difficulté à prendre en compte des applications existantes, écrites dans un langage quelconque. L'émergence récente du langage Java, intégrant dès sa conception la distribution (fonction RMI, § 4.3) constitue une étape significative vers un mode de programmation plus uniforme. La cohabitation avec des éléments d'applications écrits dans d'autres langages reste néanmoins un problème d'actualité pour les utilisateurs du langage Java.

Aucune hypothèse n'est ici faite sur la nature du langage de programmation du programme client et du programme serveur. Le rôle du langage de description d'interfaces et les outils de génération correspondants sont plus particulièrement décrits.

2.1 Langages de description d'interface

Un langage de description d'interface (IDL : Interface Definition Language) [12] permet d'exprimer, sous la forme de contrats, la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets réalisant le service et en faisant abstraction des divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation du service. Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, c'est-à-dire les types du serveur (ou interfaces IDL) et les types de données échangées entre les clients et le serveur. Le contrat IDL isole ainsi les clients et les fournisseurs de l'infrastructure logicielle et matérielle, les reliant à travers le bus logiciel mettant en œuvre le RPC (figure **5**). Un langage de spécification d'interfaces peut être commun à plusieurs langages de programmation, il permet ainsi la communication entre des programmes écrits dans ces différents langages.

Le langage IDL permet de définir l'interface de programmation (API : Application Programming Interface) des services distants, c'est-à-dire les opérations, les paramètres, leur type, leur mode et les exceptions, indépendamment de tout langage de programmation. Les services sont implémentés dans un langage indépendant du langage IDL. Comme pour les implantations de serveurs, les clients sont développés de façon *ad hoc* dans un langage quelconque.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

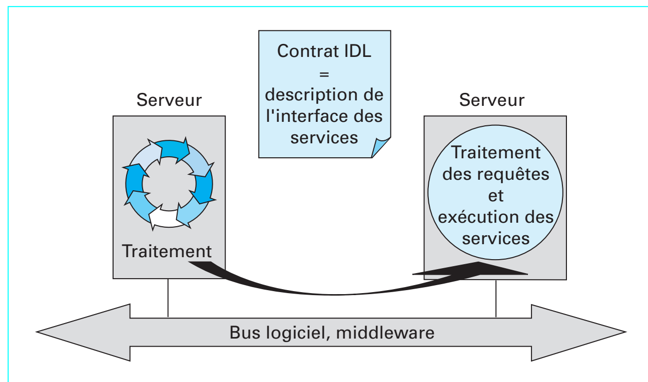


Figure 5 – Description IDL permettant de décrire l'interface d'accès à un serveur

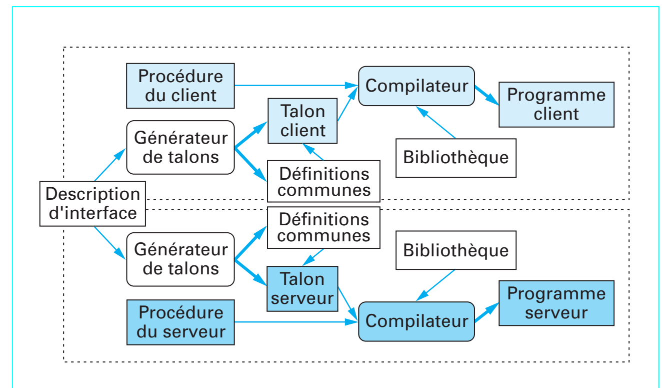


Figure 6 – Construction du client et du serveur

2.2 Exemple de description dans l'IDL de Corba

L'utilisation d'un langage IDL est illustrée par l'exemple d'un serveur « annuaire » dont la fonction est de gérer un annuaire de personnes partagé entre des utilisateurs. Une entrée de l'annuaire décrit un ensemble de propriétés associées à une personne (nom, statut, numéro de téléphone, etc.). Les services disponibles sur l'annuaire permettent, entre autres, d'ajouter/retirer une entrée ou de consulter les propriétés d'une personne donnée.

```
// description des données communes
module annuaire {
   typedef string Nom;
      // Nom d'une personne
   typedef sequence<Nom> DesNoms;
      // Ensemble de noms
   struct Personne {
      // Description d'une personne
      Nom nom;
      // - son nom
      string status;
      // - son status
      string telephone;
      // - numéro de téléphone
      string email;
      // - son adresse e-mail
      string url;
      // - son adresse WWW
};
// description des fonctions
// définies par le serveur
      interface Repertoire {
        readonly attribute string description;
        exception ExisteDeja { Nom nom; };
        exception Inconnu { Nom nom; };
        void ajouterPersonne (in Personne personne)
           raises(ExisteDeja);
        void retirerPersonne (in Nom nom)
           raises(Inconnu);
           void modifierPersonne (in Nom nom,
           in Personne p)
           raises(Inconnu);
        Personne obtenirPersonne (in Nom nom)
           raises(Inconnu);
        DesNoms listerNoms ();
      };
};
```

2.3 Outils de génération

À partir d'une description d'interface, un compilateur de talons construit un talon client et un talon serveur selon le principe décrit (figure **6**).

Pour un programmeur d'applications, la situation idéale est celle où le générateur de talons produit le code du talon client dans le langage utilisé pour programmer le client et le code du talon serveur dans celui utilisé pour le serveur. Pour pouvoir programmer le client et le serveur sans avoir à analyser le code produit, il est nécessaire de connaître la projection des différents types du langage IDL vers les différents types des langages cibles du générateur de talons. Certains générateurs de talons n'ont qu'un seul langage cible (C pour RPCgen, Java pour rmic), d'autres sont multilangages (Ada, C, C++, Java, COBOL et Smalltalk pour Corba).

Les *définitions communes* contiennent la projection des types utilisateurs définis en IDL dans la description d'interface dans le langage cible du générateur. La *bibliothèque* contient l'ensemble du code de l'appel de procédure à distance qui ne dépend pas du service appelé, par exemple le protocole de reprise en cas de panne.

Pour obtenir le programme client, il suffit de compiler simultanément le talon client (*stub*) généré à partir de la description de l'interface du service et le programme du client préalablement écrit. Pour obtenir le programme serveur, il suffit, de la même manière, de compiler simultanément le talon serveur (*skeleton*) et le programme du serveur préalablement écrit afin d'être connu du client. Le client et le serveur sont généralement déployés sur des sites distincts et les langages d'écriture du programme client et du programme serveur peuvent être différents.

3. Support système pour l'appel de procédure à distance

Comme cela a été souligné dans l'introduction, l'assimilation pure et simple de l'appel de procédure à distance à un appel de procédure local se heurte à de nombreuses difficultés techniques qui sont présentées ici. Ce paragraphe s'intéresse plus particulièrement au modèle de traitement des pannes (qui fixe la sémantique du RPC), à la prise en compte de l'hétérogénéité des différentes plates-formes ou des langages de programmation du client ou du serveur, et aux problèmes de désignation et de liaison.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

3.1 Sémantique du RPC

Définir une sémantique pour le RPC consiste à décrire de manière précise le comportement du client et du serveur à l'occurrence de pannes. Dans les faits, la sémantique est définie par le nombre d'exécutions de la procédure appelée après une défaillance du serveur ou du système de communication, suivie d'une tentative de reprise. Les différentes sémantiques mises en œuvre sont décrites :

— la procédure est exécutée *au plus une fois*. Si le client obtient une réponse, il est assuré qu'il y a eu exécution du service. S'il n'obtient aucune réponse à l'expiration du délai de garde (voir point A de la figure **4**), il considère que l'appel a échoué. Cette sémantique est la plus simple à mettre en œuvre ;

— la procédure est exécutée *au moins une fois*. Le client répète sa requête tant qu'il n'a pas eu de réponse correcte. Cette sémantique est acceptable si la procédure est idempotente (si l'effet de plusieurs appels successifs est identique à celui d'un seul appel) ;

— la procédure est exécutée *exactement une fois*. Il s'agit de la sémantique des appels locaux. Elle est très difficile à réaliser dans le cadre d'un RPC et nécessite des mécanismes évolués pour la détection des erreurs, l'élimination des requêtes multiples et la redondance du serveur nécessaire à sa disponibilité.

La sensibilité des systèmes répartis aux **défaillances** est un problème bien connu compte tenu de la multiplicité des matériels et des logiciels concernés. Les principales causes de défaillance qui affectent la mise en œuvre du RPC sont : l'impossibilité pour un client de localiser le serveur, la perte du message de requête ou du message de réponse, la disparition du processus serveur ou du processus client suite à une panne logicielle ou matérielle du site correspondant.

La détection d'une panne repose sur un mécanisme simple : le délai de garde qui est armé soit au point A (côté client), soit au point D (côté serveur) (voir figure **4**). L'expiration du délai de garde du point A permet de détecter une panne liée soit à la perte du message d'appel, soit à la panne du serveur, soit à la perte du message de réponse sans qu'il soit possible pour le processus client de distinguer entre les trois possibilités. L'expiration du délai de garde du point D permet au processus serveur de détecter une défaillance qui peut être la panne du client ou la perte du message de réponse ou de son acquittement. Là aussi, il n'est pas possible pour le processus serveur de faire la distinction entre ces différentes causes.

La difficulté consiste, en fonction d'un diagnostic peu précis — « il y a eu une panne » —, à effectuer une réparation. La nature de cette réparation est liée à la sémantique du RPC souhaitée.

■ **Perte des messages**

Généralement, le premier diagnostic que fait le client ou le serveur en l'absence d'information complémentaire est d'assimiler l'expiration du délai de garde à la perte du message. La perte du message (requête détectée par le client, réponse détectée par le client ou le serveur) est réparée par une simple retransmission du message à l'identique. Pour cela, le message de requête (respectivement de résultat) est mémorisé sur le site du client (respectivement du serveur). La réémission du message de requête peut entraîner une nouvelle exécution du service. Dans le cas où la sémantique du RPC précise que l'appel doit être réalisé au plus une fois (ou exactement une fois), l'identificateur de requête permet de déceler une nouvelle demande et de l'éliminer si besoin.

■ **Panne du serveur**

La panne du serveur peut avoir lieu avant ou après l'exécution du service. Si la procédure est idempotente, on est alors dans une situation analogue à celle des pertes de messages. Dans le cas contraire, il est nécessaire de mettre en œuvre une procédure de reprise fondée sur des mécanismes transactionnels : l'état initial du serveur (enregistré avant l'exécution du service) est restauré par le serveur lors de son redémarrage, puis le client réémet l'appel. Dans certains cas, il est aussi possible de réexécuter directement le service sans attendre une nouvelle requête du client.

■ **Panne du client**

Détectée par le serveur, la panne du client peut laisser chez un serveur des processus orphelins (processus affectés à l'exécution d'un service pour le compte d'un client qui n'existe plus). Plusieurs solutions existent pour identifier ces processus inutiles : éliminer tous les orphelins d'un client lors du redémarrage ; associer un temps limite au temps de calcul pour chaque service et si ce temps est dépassé, le serveur doit s'assurer que le client existe toujours.

3.2 Passage de paramètres

Pour un appel de procédure à distance, qui établit une communication entre deux espaces virtuels distincts, le programmeur souhaite recréer les modes de passage de paramètres usuels :

— **valeur** : ce mode est bien adapté au RPC car, au lieu de recopier sur la pile la valeur du paramètre comme cela est fait dans un appel local, il suffit de transmettre celle-ci dans le corps du message d'appel. Après déballage des paramètres, le talon serveur recopie la valeur reçue dans la pile du processus associé au service appelé ;

— **copie/restauration** : dans ce mode, la valeur des paramètres est copiée sur la pile de l'appelé comme dans le cas précédent, puis restaurée dans la pile de l'appelant lors du retour ;

— **référence** : ce mode est totalement inutilisable en réparti car il consiste à recopier sur la pile de l'appelé l'adresse mémoire du paramètre transmis par l'appelant. Dans le cas d'un RPC, il est donc nécessaire de simuler ce mode d'appel. Une solution peut consister en l'interdiction, ce qui introduit une distinction notable entre les procédures locales et les procédures distantes. Une autre solution consiste à simuler ce mode d'appel par copie/restauration. Cette solution n'est pas vraiment équivalente au passage par référence, comme le montre le contre-exemple suivant.

Procédure	Appel local	Appel réparti

double_incr (x, y)
x := x + 1;
y := y + 1;
	a := 0 ;
	double_incr (a, a)
	
	résultat : a = 2
		a := 0 ;
		double_incr (a, a)
		
		résultat : a = 1

D'autres solutions, qui ne sont pas disponibles dans les environnements d'utilisation commune, consistent soit à reconstruire l'état de la mémoire du client sur le site serveur (solution très coûteuse), soit à utiliser une mémoire virtuelle répartie (solution difficile à mettre en œuvre et nécessitant des processeurs et des systèmes homogènes).

D'une manière générale, le passage de paramètres lors d'un RPC se résume donc à un passage par valeur. Celui-ci ne présente pas de difficultés majeures mais les concepteurs de RPC ont redéfini les solutions définies pour les réseaux à des fins d'optimisation. Les solutions aujourd'hui normalisées reposent sur l'utilisation d'un IDL qui permet de décrire les types simples des langages et d'associer pour chacun d'eux une procédure pour le passage des paramètres par valeur. Malheureusement, chaque IDL a son propre principe de transmission de paramètre (Sun/ONC, RPC/DCE, Corba, Java RMI) généralement proche du principe suivant :

— IN : passage par valeur (aller) ;

— OUT : passage par valeur (retour) ;

— IN-OUT : passage par copie-restauration, mode qui n'est pas équivalent au passage par référence.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

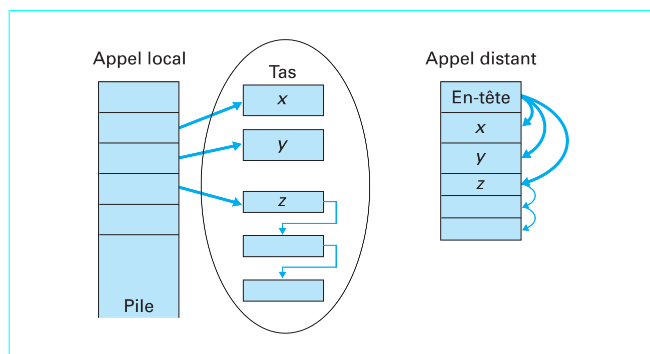


Figure 7 – Passage de paramètres complexes

En ce qui concerne le passage des paramètres de types complexes (par exemple, une liste chaînée), la plupart des RPC adoptent une solution simple qui consiste à demander à l'utilisateur de construire les procédures d'emballage et de déballage des paramètres (§ 3.5 et 4.2.2) La solution la plus sophistiquée est proposée par Java RMI dont la solution repose sur une procédure de « sérialisation/désérialisation » des paramètres qui permet de transformer n'importe quelle structure de paramètres en un flot d'octets et réciproquement.

L'exemple de la figure **7** illustre le travail à faire lors de l'emballage puis du déballage des paramètres de l'appel. L'opération d'emballage doit mettre dans une zone de mémoire unique (le message) l'ensemble des paramètres de l'appel alors que ceux-ci sont généralement dispersés dans la pile en ayant éventuellement des liens entre eux (liste chaînée par exemple). L'opération de déballage doit être capable de reconstruire l'état de la pile et du tas comme si l'appel avait été local.

3.3 Désignation et liaison

3.3.1 Principe

Dans un système informatique, le **nom** d'un objet est une information qui remplit deux fonctions : désigner l'objet, c'est-à-dire le distinguer des autres objets du système, et servir de chemin d'accès à l'objet pour permettre son utilisation dans le système. Dans le cas présent, le nom doit contenir au minimum l'identité du service à exécuter et éventuellement le nom et la localisation du serveur qui exécute ce service.

En pratique, on distingue généralement deux types de noms : le *nom symbolique* (appelé aussi nom externe) qui permet la désignation au niveau du programmeur et le *nom interne* (ou nom système) qui est utilisé par le système sous-jacent pour désigner la mise en œuvre du service. La **fonction de désignation** permet d'associer un nom symbolique (en général, une chaîne de caractères) et un nom interne interprétable par les couches basses du système ou directement par le matériel. Ce nom interne est souvent une adresse ou encore un identificateur unique à partir duquel le système peut retrouver l'adresse par association statique ou dynamique à un nom de niveau inférieur.

Les notions ci-dessus s'appliquent à tout système informatique. Dans un système réparti apparaissent les aspects spécifiques suivants :

— l'espace des noms a une grande taille ;
— l'espace des noms évolue dynamiquement et rapidement, non seulement par création de nouvelles entités, mais aussi par adjonction permanente de nouveaux composants ou système, ou par réorganisation de sa structure interne ;
— la recherche de l'indépendance entre entités logiques réalisant le service et la localisation physique (**transparence**) conduit à utiliser fréquemment la liaison dynamique entre noms et objets. Cette propriété permet de modifier la localisation des services sans modification de nom ;
— des sous-ensembles importants du réseau peuvent ne pas être accessibles à un instant donné, par suite de pannes ou de déconnexions. La panne d'une partie du service de localisation ne doit pas compromettre la réalisation de la désignation dans le reste du système.

3.3.2 Serveur de désignation

La correspondance entre noms symboliques et noms internes est souvent réalisée, dans les systèmes répartis, par un service spécialisé appelé **service de désignation**. Ce service est mis en œuvre par un ensemble de **serveurs de noms** (*name servers*). Un serveur de noms est un composant important d'un système réparti. Le service qu'il fournit doit répondre aux exigences suivantes :

— disponibilité car le bon fonctionnement du serveur de noms conditionne l'accès des utilisateurs aux autres services du système ;
— efficacité car le serveur de noms est consulté souvent ;
— capacité d'évolution pour permettre l'adjonction, la suppression ou le déplacement de composants du système.

Notons qu'au moins un serveur de noms doit être connu directement par un nom interne ou une adresse, défini statiquement ou mis à jour par l'administration du système, et communiqué à tout client lors de sa connexion au système. La disponibilité du service de désignation est obtenue par redondance. Deux méthodes sont utilisées :

— plusieurs serveurs indépendants peuvent assurer chacun la totalité du service, leur liste est connue *a priori* de tout utilisateur du système. En cas de défaillance d'un serveur, les clients s'adressent au serveur suivant de la liste ;
— plusieurs serveurs coopèrent à la réalisation du service. L'organisation doit permettre de continuer à assurer le service en cas de défaillance d'un ou plusieurs serveurs. Par exemple, on peut distribuer les noms entre les serveurs de manière que les objets les plus importants soient accessibles *via* au moins deux serveurs distincts.

3.3.3 Mise en œuvre

Pour qu'un client puisse localiser un service et y accéder, ce service doit être enregistré dans un service de désignation lui-même connu du client. Pour cela, lors du démarrage, un serveur enregistre dans un service de désignation généralement préexistant et connu de tous, différentes informations comme : son nom externe, son nom interne ou sa localisation (son adresse), et éventuellement une description des services qu'il offre. Les actions d'enregistrement du service dans un annuaire ou de consultation de l'annuaire sont pris en charge par les talons serveur et client. Des exemples de tels services sont le DNS d'Internet [13] et le service de désignation de Corba [14].

Pour pouvoir effectuer un appel, le client doit transformer le nom externe (symbolique) qu'il connaît en une chaîne d'accès qui repose généralement sur le nom interne (adresse). Cette transformation, aussi appelée liaison, peut être effectuée :

— statiquement : la correspondance entre nom du service et adresse est fixée une fois pour toutes, soit lors de l'écriture du programme, soit dans une phase de chargement et d'édition de lien, préalable à l'exécution. Le nom utilisé est alors immuable ;
— dynamiquement : l'adresse du serveur est recherchée par le client lors du premier appel au serveur ou lors de chaque appel, ce qui apporte plus de souplesse dans la mise en œuvre du service et permet une modification de la localisation du serveur.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

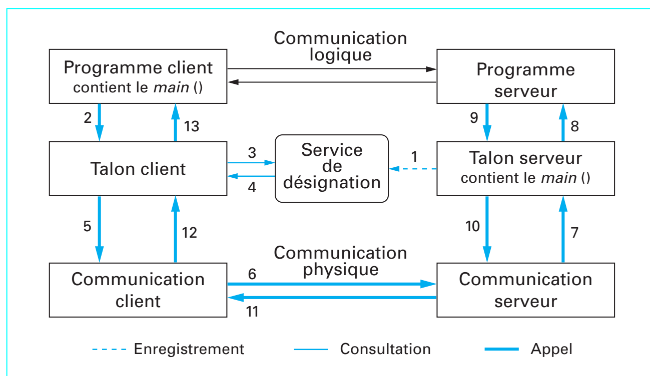


Figure 8 – Désignation et liaison

La liaison statique a pour seul avantage son efficacité. Par contre, la liaison dynamique permet de séparer la connaissance du nom du service de la sélection de la procédure qui va l'exécuter. Elle permet l'implantation retardée du service.

La figure **8** décrit les différentes étapes.

■ **Étape 1** (1) : le serveur enregistre son nom externe et son nom interne (« adresse ») dans le service de désignation. Le nom interne (« adresse ») du serveur est par exemple constitué du numéro IP de la station sur laquelle il s'exécute et du numéro de port TCP sur lequel ce serveur attend les requêtes.

■ **Étape 2** (2, 3, 4) : le client qui souhaite effectuer un appel de procédure consulte le service de désignation. Pour cela, il émet un appel de procédure vers le service de désignation (dont le nom interne est connu de tous) pour demander la traduction du nom externe du serveur que le client connaît, en un nom interne qui permettra la localisation du service et l'acheminement des requêtes.

■ **Étape 3** (5, 6, 7, 8, 9, 10, 11, 12, 13) : le client connaît le nom interne du serveur et peut donc construire la chaîne d'accès lui permettant de se connecter au serveur.

3.4 Appel statique et appel dynamique

La plupart des RPC offrent uniquement un mode de construction statique des talons clients et serveurs. Pour offrir plus de souplesse, certains RPC, comme Corba, permettent la construction dynamique de requêtes (appels dynamiques DII : Dynamic Invocation Interface) ou la construction dynamique de talons serveurs (DSI : Dynamic Skeleton Interface). Une telle approche permet, par exemple, de construire un pont générique capable d'interconnecter deux plates-formes RPC. Si un client de la plate-forme A souhaite appeler un service localisé sur un serveur B, la partie du pont présente sur A représente tous les serveurs de B et la partie du pont présente sur B représente tous les clients de A. Il est donc impossible que ces ponts contiennent toutes les interfaces disponibles sur A ou B, la solution adoptée repose sur la construction dynamique soit d'un talon client, soit d'un talon serveur.

Offrir l'appel dynamique impose de pouvoir récupérer, lors de l'exécution, l'interface du service que l'utilisateur souhaite accéder. Pour mettre en place cette nouvelle fonction, la description d'interface est alors enregistrée dans un répertoire d'interfaces, utilisé soit par les générateurs de talons, soit par les outils d'invocation dynamique (figure **9**).

L'appel dynamique s'applique aux cas où l'interface de l'objet appelé n'est pas connue au moment de la compilation du programme client, mais trouvée dynamiquement, en cours d'exécution, en consultant le répertoire d'interfaces. En l'absence de talon client, les paramètres de l'appel doivent être accompagnés de l'indication de leur type, pour vérifier que ce type est conforme à celui spécifié dans l'interface.

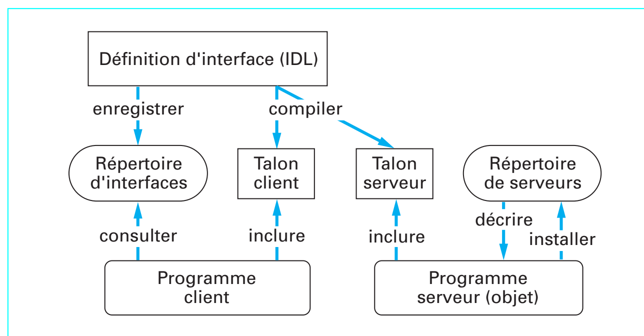
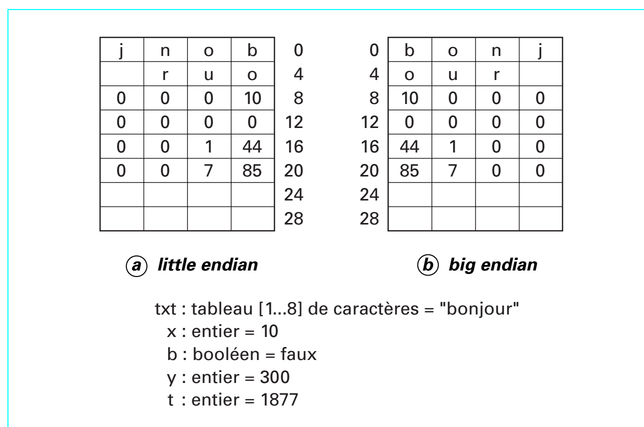


Figure 9 – Clients et serveurs dans le modèle Corba



j	n	o	b	
	r	u	o	4
0	0	0	10	8
0	0	0	0	12
0	0	1	44	16
0	0	7	85	20
				24
				28

(a) *little endian* (première ligne d'adresse : 0)

0	b	o	n	j
4	o	u	r	
8	10	0	0	0
12	0	0	0	0
16	44	1	0	0
20	85	7	0	0
24				
28				

(b) *big endian*

txt : tableau [1...8] de caractères = "bonjour"
x : entier = 10
b : booléen = faux
y : entier = 300
t : entier = 1877

Figure 10 – Différentes représentations des données

3.5 Hétérogénéité

L'hétérogénéité des données se trouve à différents niveaux :

— le codage des données en mémoire (§ 3.5.1). Deux représentations coexistent : *little endian* et *big endian* ;

— le codage des données par les compilateurs (§ 3.5.1). Deux compilateurs C, par exemple, n'ont pas nécessairement la même manière de coder des structures et en particulier d'aligner ou non les données sur des mots mémoires. Ces spécificités dépendent en même temps du compilateur (et de sa volonté d'optimiser soit la place mémoire, soit les temps d'accès) et des contraintes des processeurs ;

— le codage des données par les applications (§ 3.5.2). Aujourd'hui, les applications manipulent des données structurées qu'il est nécessaire de faire migrer entre stations (sons, images, documents, etc.).

3.5.1 Codage des données par les processeurs

Chacune des stations possède son propre système de représentation des données : *little endian* ou *big endian* (figure **10**). De même, chaque langage de programmation, voire chaque compilateur pour un langage donné, possède sa propre représentation

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

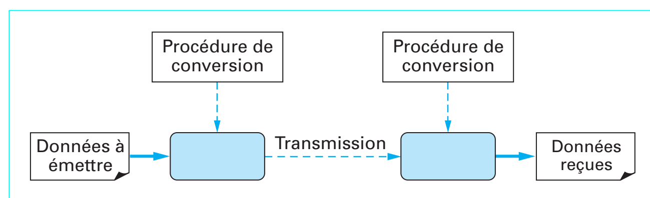


Figure 11 – Codage et décodage XDR

des données. Il est donc nécessaire, dans le cas du modèle client-serveur, où chacune des deux entités peut s'exécuter sur une machine différente et être programmée par un langage différent, de trouver une solution permettant de transformer les données du format du client dans le format du serveur lors de l'appel ou du format du serveur dans le format du client lors du retour.

Pour être indépendant d'un système de représentation des données (imposé par exemple par une machine, un langage), on utilise généralement lors du transport une représentation universelle. L'OSI a normalisé une représentation ASN.1 (Abstract Syntax Notation 1) qui transporte, en même temps que les données, la description du type codée sous une forme abstraite (métadescription). Cette norme est relativement peu utilisée en raison de la complexité inhérente à ce système de métareprésentation pour construire des appels de procédure entre processus. On lui préfère le protocole XDR [15] (eXternal Data Representation) développé par Sun pour son RPC. Ce protocole est plus simple car il ne transporte pas la représentation des données. Il permet néanmoins de représenter celles-ci sous une forme universelle.

Nota : ASN.1 est par contre très utilisé dans le domaine de la gestion de système et c'est le langage de description des MIB (Management Information Base).

Lors de la phase de codage (figure **11**), XDR transforme les données à émettre en une suite d'octets qui peut être envoyée dans un fichier ou sauvegardée en mémoire. XDR sait transformer les types simples du langage C ; il faut par contre lui donner les procédures permettant de transformer les types complexes d'une application donnée. Lors de la phase de décodage, le travail inverse est réalisé et il faut à nouveau fournir les procédures complémentaires permettant de reconstruire les données de types complexes.

Ces solutions normalisées, de bas niveau, imposent l'utilisation d'une double transformation (côté client et côté serveur), qui peut être inutile dans le cas où le serveur et le client utilisent déjà le même codage. D'autres solutions peuvent être utilisées : représentation locale pour le client et conversion par le serveur qui doit connaître les codes des différents clients ou négociation entre le client et le serveur pour déterminer le codage à utiliser. Un exemple d'utilisation de XDR est donné au paragraphe 4.1.1.

3.5.2 Codage des données par les applications

La plupart des applications de l'Internet manipulent des types de données structurés. Ces nouveaux formats, dépendant des applications, possèdent leur propre système de transcodage et ils sont aussi utilisables dans le cadre du RPC. Par exemple, ont été définis des formats spécifiques pour assurer le transport des données entre des stations hétérogènes, en particulier dans le domaine du multimédia : format d'image (gif, jpeg), de film (mpeg), de son (wav), etc.

Le développement du courrier électronique (service construit à l'aide du RPC) a imposé la définition d'un format [16] permettant de regrouper à l'intérieur d'un même message des données de différents types. Chaque élément d'un message MIME (Multipurpose Internet Mail Extensions) porte diverses indications : indicateur de type/sous-type (text/plain, text/enriched, image/gif, image/jpeg, audio/x-wav, audio/basic, audio/x-mpeg, application/postscript, application/rtf, application/pdf, etc.), indicateur d'encodage (Content-Transfer-Encoding: quoted-printable), indicateur de jeu de caractères (charset=iso-8859-1). Ces indications sont automatiquement ajoutées par le logiciel de courrier électronique s'il se conforme au standard MIME et s'il a été correctement configuré. Ce type de format est de plus en plus utilisé pour échanger des données sur l'Internet.

Le développement des documents électroniques a fait émerger la nécessité de posséder des formats de représentation évolutifs. XML est un format qui permet de mettre des données structurées dans un fichier texte (ce qui ne signifie pas nécessairement que ce fichier texte soit lisible directement). Des exemples de données structurées sont : feuilles de calcul, carnet d'adresses, transactions financières, dessins en CAO, etc. L'intérêt de ces formats est lié au souhait de manipuler ces données à travers d'autres logiciels que ceux qui les ont produites, indépendamment d'une plate-forme donnée.

4. Mise en œuvre de l'appel de procédure à distance

Les approches pour la réalisation effective d'un appel de procédure à distance présentées ici utilisent des interfaces de programmation de différents niveaux : utilisation directe des sockets, utilisation d'un IDL et d'un générateur de talons (RPCgen de Sun), utilisation de Java RMI.

4.1 Envoi de messages

La mise en œuvre d'un appel de procédure à distance de manière directe en utilisant les sockets (ports de communication associés à des processus) présente uniquement un objet pédagogique afin de bien mettre en évidence les différents éléments de l'appel qui sont cachés lors d'une réalisation avec des générateurs de talons.

Selon le mode de réalisation recherché (mode non connecté utilisant le protocole UDP ou mode connecté utilisant le protocole TCP), les programmes du serveur et du client prennent l'une des formes présentées ci-après.

4.1.1 Client-serveur en mode non connecté

La figure **12*a*** montre l'enchaînement des appels Unix permettant de construire :

— un processus serveur monoprogrammé ou un serveur multiprogrammé créant un processus « fils » pour traiter chaque nouvelle requête ;
— un processus client.

■ **Structure de données communes**

```
/*Descripteurs associés aux sockets */
int                 MySocket;   /*port de la socket
                                              client*/
struct sockaddr_in MyAddress; /*descripteur de ce
                                              socket*/
struct sockaddr_in ServerAddress; /*descr. du
                                       socket serveur*/
/*Informations associées aux sites*/
struct hostent  *ServerHost; *site du serveur*/
/*Informations associées aux messages*/
long  SendMessage;         /*message à envoyer*/
long  MessAnswer;          /*réponse à un message*/
struct sockaddr SenderAddress; /*adresse réseau
                                          expéditeur*/
int   SenderLength;    /*longueur adresse réseau*/
```


Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

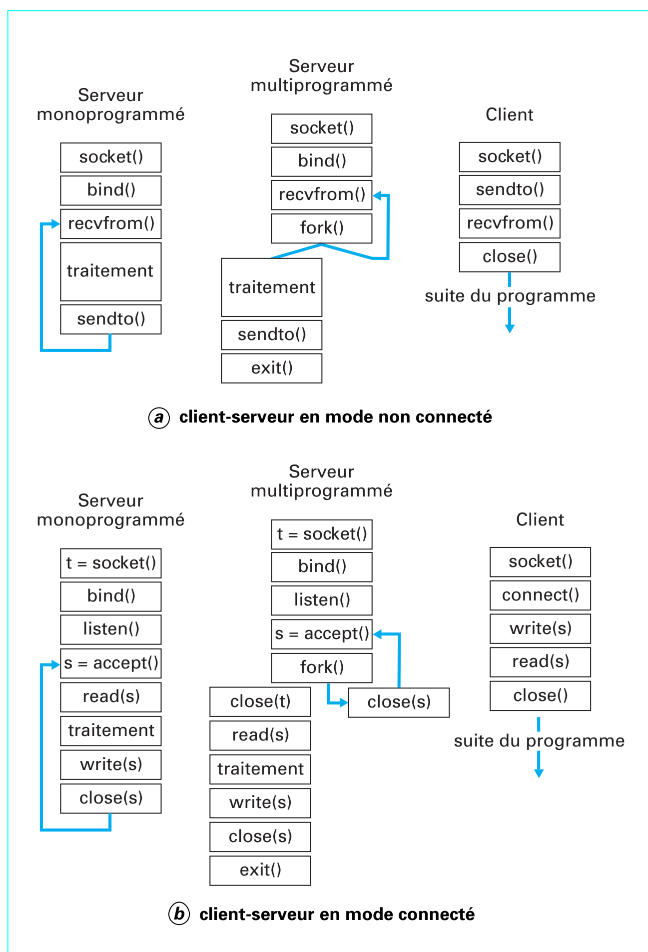


Figure 12 – Enchaînement des appels

Programme du client

```
int rv;     /*valeur retournée par les différentes
                                       fonctions*/
/*Récuperation du nom du serveur et calcul de son
                                  adresse réseau*/
ServerHost = gethostbyname ("rangiroa");
if (ServerHost == NULL) {
   printf("Nom du site serveur inexistant:
                                  %s\n",argv[2]);
   exit(1);
}
ServerAddress.sin_addr.s_addr = ((struct in_addr*)
                      ServerHost->h_addr)->s_addr;
ServerAddress.sin_family = ServerHost->h_addrtype;
ServerAddress.sin_port = htons(PORT_DAEMON);
/*Création du socket client*/
if ((MySocket = socket(AF_INET,
                             SOCK_DGRAM, 0))== -1){
   printf("Création impossible du socket\n");
   exit(1);
}
/*Liaison du socket*/
MyAddress.sin_family = AF_INET;
MyAddress.sin_addr.s_addr = INADDR_ANY;
MyAddress.sin_port = 0;
if (bind(MySocket, &MyAddress, sizeof(MyAddress))
                                          == -1){
   printf("Liaison impossible du socket\n");
   exit(1);
}
SendMessage =...; /*construction du message
                                        d'appel*/
/*message d'appel*/
if ((rv = sendto(MySocket, (char *) &SendMessage,
            sizeof(SendMessage), 0, ServerAddress,
            sizeof(ServerAddress))) == -1){
   printf("Erreur dans envoi de message\n");
   exit(1);
}
/*attente du message de réponse*/
if ((rv = recvfrom(MySocket, &MessAnswer,
             sizeof(MessAnswer), 0, &SenderAddress,
             &SenderLength) == -1){
   printf("Erreur réponse du serveur\n");
   exit(1);
}
/*Fermer le socket utilisé*/
close(MySocket);
```

Programme du serveur

```
/*création du socket du serveur*/
if((ServerSocket = socket(AF_INET, SOCK_DGRAM,0))
                                          == -1){
   printf("Création impossible du socket du
                                     serveur\n");
   exit(1);
}
/*Liaison du socket*/
ServerAddress.sin_family = AF_INET;
ServerAddress.sin_addr.s_addr = INADDR_ANY;
ServerAddress.sin_port = htons(PORT_DAEMON);
if (bind(ServerSocket, &ServerAddress, sizeof
                         (ServerAddress)) == -1) {
   printf("Erreur de liaison pour le socket du
                                     serveur\n");
   exit(1);
}
/*Initialisations nécessaires à l'envoi de la
                                        réponse*/
SenderAddress.sa_family = AF_INET;
SenderLength = sizeof(SenderAddress);
while (TRUE) {
   /*Attente d'une demande*/
   if ((rv = recvfrom(ServerSocket, (char *)
        &CommandMessage, sizeof(CommandMessage),
        0, &SenderAddress, &SenderLength)) == -1){
      printf("Erreur en réception serveur\n");
      exit(1);
   }
```

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

```
    /*Récupération du service à exécuter et
                                  de l'adresse client*/
    command = Extract(CommandMessage);
    State.address = SenderAddress;
    State.length = SenderLength;
    /*exécution du service*/
    ...
    /* Construction et envoi du message de réponse */
    Answer = ...;
    if   ((rv = sendto(ServerSocket,(char *)&Answer,
                sizeof(Answer),  0,  &State.address,
                State.length)) == -1) {
       printf("Erreur dans message d'arrêt\n");
       exit(1);
    }
}
/*Rendre les ressources avant de terminer*/
close(ServerSocket);
```

4.1.2 Client-serveur en mode connecté

Nous ne détaillons pas ici les programmes du client et du serveur pour le mode connecté. Ils peuvent être aisément déduits à partir du mode non connecté présenté précédemment et à partir de l'enchaînement des appels décrits dans la figure **12*b*** permettant de construire, sous Unix :

— un processus serveur monoprogrammé ou un serveur multiprogrammé créant un processus « fils » pour traiter chaque nouvelle requête ;

— un processus client.

4.1.3 Éléments de choix

Les principales caractéristiques du **protocole UDP** utilisé pour construire un client-serveur en **mode non connecté** sont :

— pas d'établissement préalable d'une connexion. Il est donc adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (un message) ;

— mode d'échange par messages. Le récepteur reçoit les données suivant le même découpage que celui effectué par l'émetteur. La taille des messages est bornée : ce protocole n'est pas adapté aux échanges nécessitant d'importants transferts de données ;

— pour répondre à chaque client, le serveur doit en récupérer l'adresse. Il faut pour cela utiliser les primitives sendto et recvfrom.

Les principales caractéristiques du **protocole TCP** utilisé pour construire un client-serveur **en mode connecté** sont :

— établissement préalable d'une connexion (circuit virtuel). Le client demande au serveur s'il accepte la connexion. La fiabilité est assurée par le protocole de transport utilisé (TCP) et il est possible d'émettre et de recevoir des caractères urgents (OOB : Out Of Band) ;

— mode d'échange par flots d'octets. Le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur ;

— après initialisation, le serveur est « passif ». Il est activé lors de l'arrivée d'une demande de connexion d'un client. Un serveur peut répondre aux demandes de services de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente ;

— la construction de serveurs multiplexés est relativement aisée, même dans le cas d'échanges nécessitant plusieurs demandes de service.

Les protocoles connectés sont généralement préférables pour des transmissions sur une longue durée, sur une longue distance ou avec un important volume de données à transférer. Pour les réseaux locaux où le temps de réponse est crucial, on choisit souvent des protocoles non connectés.

4.2 Appel de procédure avec langage de description d'interface

Le RPC de Sun, aujourd'hui disponible sur la plupart des systèmes Unix, est devenu un standard de facto. Il est disponible au programmeur par l'intermédiaire du générateur de talon RPCgen et de la bibliothèque rpclib. Le passage des paramètres se fait uniquement par copie-restauration et le traitement de l'hétérogénéïté est pris en charge par le protocole XDR (§ 3.5.1).

À l'aide des outils disponibles, il est possible de programmer des applications client-serveur qui répondent à divers besoins [17] :

— le premier niveau permet la construction de la partie cliente de serveurs déjà construits et enregistrés, accessibles par le RPC de Sun, par exemple un client SMTP (messagerie électronique) ou un client NFS (fichiers répartis) ;

— le deuxième niveau permet la mise en œuvre d'un RPC dont la sémantique est *au plus une fois*. Cette mise en œuvre utilise les sockets en mode UDP. Du fait de sa simplicité d'utilisation, c'est le niveau le plus utilisé, il permet de construire la partie cliente et la partie serveur ;

— le troisième niveau permet d'implémenter des RPC ayant la sémantique *au plus une fois*. Il est mis en œuvre à l'aide de TCP. Ce RPC gère un temporisateur qui permet la réémission des requêtes avant de retourner une erreur. La durée des temporisations et le nombre de réémissions sont paramétrables.

4.2.1 Générateur RPCgen

RPCgen est le générateur de talons de Sun, il permet de construire un programme client et un programme serveur à partir de la spécification des services du serveur décrits dans un langage de description d'interface proche du langage C. Le mode opératoire est conforme à ce qui est indiqué au paragraphe 2 : description de l'interface dans un fichier suffixé « interface_name . x » puis compilation de ce fichier par RPCgen qui produit : un fichier décrivant le format des données échangées entre le client et le serveur (prog . h), deux fichiers permettant de construire le client (prog_clnt . c et prog_xdr . c) et deux fichiers permettant de construire le serveur (prog_svc . c et prog_xdr . c).

Les programmes client . c et serveur . c restent à la charge du programmeur mais RPCgen peut apporter une aide en fournissant un squelette du fichier client . c contenant déjà une fonction *main* et la manière d'appeler les différents services du serveur, et un squelette du fichier serveur . c contenant les prototypes des différents services.

4.2.2 Exemple de réalisation

La manière la plus simple de comprendre le fonctionnement de RPCgen est de construire un exemple simple, comme par exemple un annuaire permettant d'associer à une personne un numéro de téléphone, puis d'effectuer une recherche.

Description de l'interface

Le fichier de données est suffixé « . x ».

1. On définit les données en entrée et en sortie. Une fonction a au plus un paramètre d'entrée et un de sortie. On doit donc avoir recours à des structures pour des appels avec plusieurs paramètres.

2. On déclare ensuite le programme et l'ensemble des procédures appelées à distance.

```
const MAX_NAME = 255;
typedef char Name <MAX_NAME>;
typedef long PhoneNumber;
typedef long status;
struct entry {Name name; PhoneNumber phoneNumber;}
typedef struct entry Entry;
```


Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

```
program DISTR_AGENDA{
   version VERSION_NUMBER{
      PhoneNumber Lookup (Name)=1;
                             //numéro de la procédure
      status Insert (Entry)=2;
                             //numéro de la procédure
   } = 1;                    //numéro de version
} = 76;                      //numéro de programme
```

La compilation de ce fichier par la commande RPCgen interface.x produit les fichiers : interface.h, interface_xdr.c, interface_svc.c et interface_clnt.c.

Le fichier interface.h contient la déclaration des données décrites dans le fichier interface.x :

```
#define MAX_NAME 255
typedef struct{
      u_int Name_len;
      char *Name_val;
}Name;
typedef long Agenda;
typedef long status;
struct entry{
      Name name;
      Agenda agenda;
};
typedef struct entry entry;
typedef entry Entry;
```

mais aussi la compilation de l'interface de l'objet serveur, elle aussi décrite dans le fichier interface.x :

```
#define DISTR_AGENDA ((unsigned long)(76))
#define VERSION_NUMBER ((unsigned long)(1))
#define Lookup ((unsigned long)(1))
extern Agenda * lookup_1();
#define Insert ((unsigned long)(2))
extern status * insert_1();
extern int distr_agenda_1_freeresult();
```

et la signature des fonctions xdr qui seront utilisées pour convertir les données :

```
/* the xdr functions */
extern bool_t xdr_Name();
extern bool_t xdr_Agenda();
extern bool_t xdr_status();
extern bool_t xdr_entry();
extern bool_t xdr_Entry();
```

Procédures de conversions de paramètres (XDR)

Pour convertir les paramètres, RPCgen utilise le protocole XDR [15] qui reconnaît des types de base (divers formats d'entier ou de réel, booléen, énumération, suite d'octets de longueur fixe ou variable, chaîne de caractères), des types composés (tableau de longueur fixe ou variable, structure, union) et possède un mécanisme permettant la construction de fonctions de traduction pour des types privés liés à l'application.

La compilation du fichier d'interface par RPCgen produit un fichier interface_xdr.c qui contient l'appel aux procédures de conversion des données au format réseau pour le talon client et les procédures de conversion inverse qui seront utilisées sur le site du serveur. Les procédures de conversion pour les types de base sont contenues dans la bibliothèque XDR. Il reste éventuellement à compléter ces procédures si le type à manipuler n'est pas assimilable à un assemblage de type de base. Par exemple, si le paramètre est une liste chaînée, le programmeur doit donner la fonction permettant d'emballer cette liste dans un message, puis la fonction permettant de déballer la liste du message.

Dans le cadre de l'exemple, RPCgen génère automatiquement les procédures de conversion. Le fichier généré, de nom interface_xdr.c, a la forme suivante :

```
/*
* Please do not edit this file.
* It was generated using rpcgen.
*/
#include "interface.h"

bool_t xdr_Name(xdrs, objp) // mise au format
                               // réseau du type Name
   register XDR *xdrs;
   Name *objp;
{
   register long *buf;
   if(!xdr_array(xdrs, (char **)&objp->
               Name_val, (u_int *) &objp->Name_len,
               MAX_NAME,
      sizeof(char),(xdrproc_t)xdr_char))
      return (FALSE);
   return (TRUE);
}
...
bool_t xdr_Entry(xdrs, objp)
      register XDR *xdrs;
      entry*objp;
{
      register long*buf;
   if(!xdr_Name(xdrs, &objp->name))
         return (FALSE);
      if (!xdr_Agenda(xdrs, &objp->agenda))
         return (FALSE);
         return (TRUE);
}
```

Les autres procédures de conversion ont été omises afin de ne pas alourdir la rédaction.

Développement du programme client

La compilation du fichier d'interface par la commande **rpcgen** – Sc interface.x > client.c produit un squelette pour le programme client. Celui-ci a la forme suivante (les parties de code en gras ont été ajoutées ultérieurement) :

```
/* 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 "interface.h"

void distr_agenda_1(host)
   char *host;
{
   CLIENT      *clnt;
   PhoneNumber *result_1;
   Name        lookup_1_arg;
   Status      *result_2;
   Entry       insert_1_arg;
#ifndef DEBUG           // Code pour la Mise au
                                  //point du RPC
   clnt = clnt_create(host, DISTR_AGENDA,
                     VERSION_NUMBER, "netpath");
   if (clnt == (CLIENT *) NULL) {
      clnt_pcreateerror(host);
      exit(1);
   }
#endif /* DEBUG */
```

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

```
    /* Modèle pour l'appel du service : lookup */
    lookup_1_arg.Name_len=strlen("Michel")+1;
    lookup_1_arg.Name_val="Michel";
    result_1 = lookup_1(&lookup_1_arg, clnt);
    if (result_1 == (PhoneNumber *) NULL) {
       clnt_perror(clnt, "call failed");
    }
    printf("retour lookup : numéro de téléphone
                                = %d\n",*result_1);
    /* Modèle pour l'appel du service : insert */
    insert_1_arg.name.Name_len=strlen ("Michel")+1;
    insert_1_arg.name.Name_val="Michel";
    insert_1_arg.agenda=12;
    result_2 = insert_1(&insert_1_arg, clnt);
    if (result_2 == (status *) NULL) {
       clnt_perror(clnt, "call failed");
    }
    printf("retour insert : status = %d\n",
                                       *result_2);
}
main(argc, argv)
    int argc;
    char *argv[];
{
    char *host;
    if (argc < 2) {
       printf("usage: %s server_host\n", argv[0]);
       exit(1);
    }
    host = argv[1];
    distr_agenda_1(host);
}
```

Le programme client a le fonctionnement suivant :

— *création d'un descripteur RPC* (handle) ;

— *appel*. Seule la construction du paquet d'appel et l'extraction des résultats ont été effectivement programmées ;

— *libération* du descripteur.

Développement du programme serveur

La compilation du fichier d'interface par la commande **rpcgen** – Ss interface.x > serveur.c produit un squelette pour le programme serveur. Celui-ci a la forme suivante (les parties de code en gras ont été ajoutées ultérieurement) :

```
/* 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 "interface.h"
Agenda *lookup_1(argp, rqstp)
    Name *argp; struct svc_req *rqstp;
{
    static PhoneNumber result;
    /* insert server code here */
    printf ( "serveur : lookup %s\n",
                              argp->Name_val);
    return (&result);
}
status *insert_1(argp, rqstp)
    Entry *argp; struct svc_req *rqstp;
{
    static status result;
    /* insert server code here */
    printf ( "serveur : insert %s-%d\n",
           argp->name.Name_val, argp->phoneNumber);
    return (&result);
}
```

Lors de son activation, le programme serveur a le fonctionnement suivant :

— *création d'un descripteur RPC* (handle) ;

— *enregistrement* auprès du processus PortMap afin que le service soit connu sur le réseau ;

— *traitement des appels*. Cette partie, similaire à celle d'une procédure locale, est la seule à avoir été effectivement programmée.

Production de code

L'outil RPCgen produit aussi un squelette de Makefile capable de générer le code du serveur et du client (rpcgen -Sm interface.x > Makefile) qu'il faut compléter en modifiant les lignes :

```
SOURCES_CLNT.c = client.c interface_xdr.c
                                interface_clt.c
SOURCES_SVC.c = serveur.c interface_xdr.c
                                interface_svc.c
CLIENT = client
SERVER = serveur
```

4.2.3 Évaluation

Les principales limitations de RPCgen sont les suivantes :

— la taille du message d'appel et du message de réponse doit être inférieure à 8 Ko (liée à l'utilisation d'UDP) ;

— il n'y a qu'un seul paramètre d'appel et un seul de retour. S'il y en a plusieurs, il faut construire une structure ;

— l'utilisation de XDR peut être omise dans le cas où le site client et le site serveur utilisent le même format de donnée ;

— il n'existe aucune facilité pour écrire un serveur multiplexé.

Malgré ces limitations, RPCgen reste très utilisé dans l'environnement Unix.

4.3 Appel de méthode à distance : RPC à objet

L'appel de méthode à distance (RMI : Remote Method Invocation) est le nom donné à l'appel de procédure à distance lorsque le serveur est représenté par un objet. Dans ce cas, les services (disponibles sur le serveur sont définis par les méthodes (ou opérations) de l'objet. L'appel de méthode à distance est le mécanisme de base pour les applications réparties écrites à l'aide du langage Java ou développées dans l'environnement Corba de l'OMG. Cet « objet serveur » peut être un objet du langage de programmation, — c'est le cas dans Java RMI — ou un objet géré par le système et n'ayant pas nécessairement une représentation langage — c'est le cas dans Corba.

Il n'y a pas de différence fondamentale d'un point de vue pratique entre un appel de procédure et un appel de méthode (figure **13**). Néanmoins, le principe de désignation est un peu différent. Dans le cas de l'appel de procédure, on identifie le serveur par un couple : adresse IP du site et port TCP ou UDP lié au processus. Dans le cas de l'appel de méthode, c'est l'objet lui-même qui est directement désigné par un identificateur d'objet (OID : Object Identifier). Un OID est un identificateur unique pour l'ensemble du système réparti. Rappelons également que, quelle que soit la nature des noms internes, il est toujours possible d'y associer un nom symbolique.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

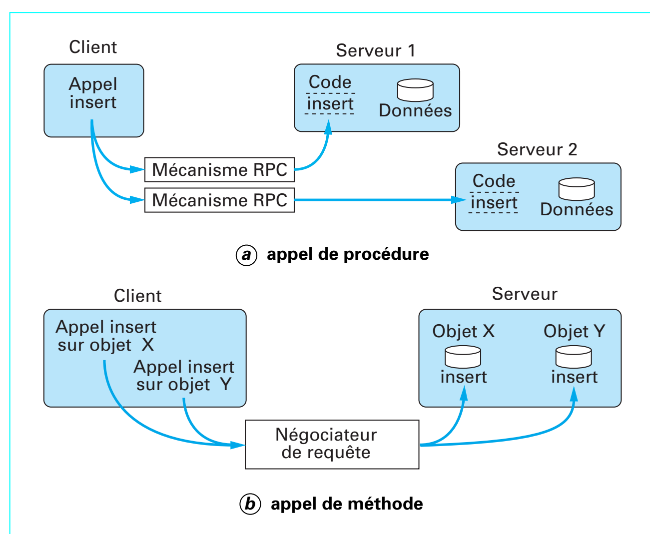


Figure 13 – Appel de procédure et appel de méthode

Nota : la suite de cet article présente l'appel de méthode à distance intégré à Java : RMI. Le lecteur intéressé par l'utilisation de l'infrastructure Corba se reportera à l'article [H 2 758] qui lui est consacré.

4.3.1 Java RMI

RMI permet de rendre accessible à distance des objets Java tout en cachant, d'un point de vue syntaxique, la répartition de l'application. La mise en œuvre de ces appels repose sur les principes évoqués précédemment avec quelques aspects spécifiques liés au langage Java.

■ **Description de l'interface** de l'objet que l'on souhaite pouvoir accéder à distance.

Les spécifications de l'objet distant sont définies par une interface Java qui étend l'interface java.rmi.Remote et qui doit être capable de lever l'exception java.rmi.RemoteException. Les paramètres doivent être soit des types simples du langage Java, soit des objets Java sérialisables, soit des références vers des objets distants (des interfaces). Cette description est relativement semblable à celle des fichiers *.x de RPCgen.

```
import java.rmi.*
public interface ItfObjetDistant extends
                                    java.rmi.Remote{
   // Name et PhoneNumber sont des classes
                                       sérialisables
   public PhoneNumber Lookup(Name)
                      throwsjava.rmi.RemoteException;
   public status Insert(Name,PhoneNumber)
                      throwsjava.rmi.RemoteException;
}
```

La *sérialisation* des objets consiste à les transformer en une chaîne d'octets. Elle est offerte naturellement pour tout objet Java qui dérive d'un objet sérialisable ; elle est récursive, c'est-à-dire qu'il est possible de transformer en une suite d'octets n'importe quel graphe d'objet sérialisable et de restaurer à l'identique ce graphe par l'opération inverse. D'une manière générale, tous les objets Java sont sérialisables, sauf les flots d'exécution (threads) et les objets qui en dérivent. En pratique, la sérialisation est utilisée pour envoyer une copie de l'objet à travers une connexion réseau ou pour le conserver dans un fichier. Dans la plupart des langages de programmation, le programmeur qui désire conserver une structure de données dans un fichier doit écrire un algorithme spécifique, ce qui est souvent pénible et ennuyeux. Java simplifie grandement cette opération en la réalisant automatiquement et récursivement. Par exemple, pour envoyer un arbre en mémoire à un autre programme par le réseau, il suffit d'envoyer l'objet racine et Java suivra les références de la structure pour copier l'arbre entier. Pour des utilisations particulières (limitation du nombre d'objets à sérialiser, sécurité), il est possible de spécifier la façon dont un objet doit être transformé en une chaîne d'octets. La *désérialisation* est l'opération inverse qui permet de reconstruire un objet (ou un graphe d'objets) à partir d'une chaîne d'octets.

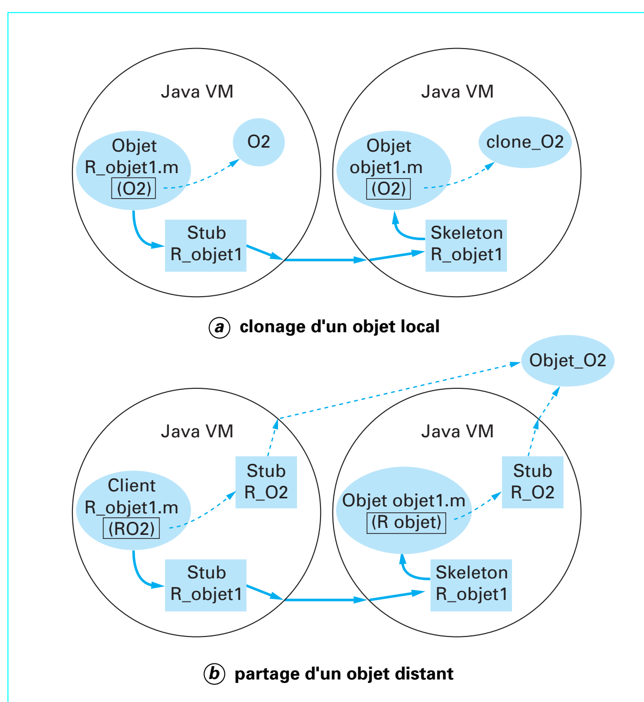


Figure 14 – Passage de paramètre

Lors du passage de paramètre, Java RMI maintient les invariants de désignation et les différents cas suivants peuvent se produire :

— si le paramètre est une **valeur** (entier, booléen, etc.), celle-ci est transmise directement au serveur lors de l'appel ;

— si le paramètre est un **objet local** (figure **14*a***), il doit être local pendant le traitement du service par le serveur. L'objet paramètre est alors cloné, ainsi que tous les objets qu'il désigne par utilisation implicite de la sérialisation Java. Le mode de passage du paramètre est par **valeur** ;

— si le paramètre est un **objet distant** (figure **14*b***), il reste distant pendant le traitement du service. Le mode de passage du paramètre est *par référence* ce qui permet le partage de l'objet entre plusieurs serveurs. Celui-ci est mis en œuvre par copie-restauration de l'objet « référence » (talon client) qui permet l'accès à l'objet distant, ce qui revient à dire que c'est la référence (*stub*) qui est dupliquée et non pas l'objet référencé. La duplication du *stub* utilise aussi la sérialisation Java.

■ **Implémentation de l'objet serveur** mettant en œuvre au moins cette interface. L'objet serveur est un objet Java « standard » qui peut implémenter plusieurs interfaces distantes et d'autres interfaces locales. Il doit donner le code des méthodes pouvant être appelées à distance et celui des méthodes locales.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

L'implantation de l'objet distant hérite de java.rmi.UnicastRemoteObject et implante l'interface précédente.

```
public class CodeObjetDistant implements
                                        ItfObjetDistant
   extends java.rmi.UnicastRemoteObject {
   // Constructeur
   CodeObjetDistant()
                    throwsjava.rmi.RemoteException
      {...} ;       //code du constructeur
   public PhoneNumber Lookup(Name)
                    throwsjava.rmi.RemoteException
      {...} ;       //code de la méthode Lookup
   public Insert(Name, PhoneNumber)
                    throwsjava.rmi.RemoteException
      {...} ;       //code de la méthode Insert
   // Le code des méthodes distantes est du code Java
   // Eventuellement, cet objet contient d'autres
   // méthodes destinées au fonctionnement de la
                                                classe
}
```

■ **Compilation de l'interface** par l'outil rmic afin de produire le *stub* et le *skeleton*. Comme Java utilise la liaison dynamique, la production des différents talons peut être différée et réalisée après la compilation du client et du serveur.

■ Pour qu'un client puisse accéder à un objet distant, il est nécessaire que celui-ci soit enregistré dans un **service de désignation**. Le principe de désignation d'un serveur reprend la structure d'une URI (Uniform Resource Identifier), à savoir : rmi ://nom_de_machine : port/nom_de _serveur. Le nom de la machine et le port sont ceux associés au service de désignation de rmi (rmiregistry). L'enregistrement des services est réalisé par l'intermédiaire de la classe Naming (par appel de la méthode rebind).

■ Java permettant le **chargement dynamique de code**, il n'est pas nécessaire que le code du *stub* soit présent sur le site du client. Par contre, le service de désignation doit connaître un répertoire accessible par le protocole http qui contient le code.

4.3.2 Exemple

Nous poursuivons la description de l'exemple de l'annuaire commencé au paragraphe 4.2.2 en nous focalisant sur le code du client et l'installation du serveur. Le code proprement dit de l'objet serveur n'est pas décrit ici car il ne présente pas d'intérêt particulier pour l'exposé (il s'agit d'un ensemble de méthodes écrites en Java).

■ **Installation et déploiement du serveur**

La compilation du serveur est réalisée en deux étapes de la façon suivante :

```
javac ItfObjetDistant.java CodeObjetDistant.java
rmic ItfObjetDistant
```

Ainsi, en Java RMI, la production des talons client et serveur s'effectue uniquement sur le site du serveur. La propriété de liaison dynamique de code permet en outre de faire cette compilation après la compilation du serveur pour laquelle seule l'interface des services distants est nécessaire. Pour que le client puisse dynamiquement récupérer le talon client, il faut le déposer sur un site Web et préciser cette URI lors du lancement du serveur.

```
java -Djava.rmi.server.codebase
                       = http://site_web/... Server
```

Pour qu'un objet serveur puisse être accessible, il faut créer au moins une instance du serveur et l'enregistrer dans le service de nom de Java (rmiregistry). Faire l'hypothèse que ce service est déjà activé, permet de procéder de la manière suivante :

```
public static void main(String args[]) {
   String URI;
   try {
      // Crée une instance de l'objet serveur
      ItfObjetDistant obj = new
               CodeObjetDistant("HelloServeur");
      // Enregistre l'objet créé auprès
                                du serveur de noms.
      URI =
      "//"+InetAddress.getLocalHost().getHostName()+":"
                              +port+"/mon_serveur";
      Naming.rebind(URI, obj);
      System.out.println("HelloServer"+"
                               bound in registry");
   } catch (Exception exc) {...}
}
```

■ **Écriture du client**

L'écriture du client ne présente pas de difficulté particulière puisque Java RMI cache le fait que l'objet appelé est distant. La seule opération complémentaire, engendrée par la distribution, consiste à construire la chaîne d'accès à l'objet distant, c'est-à-dire à installer les talons pour les services distants. Cette opération est réalisée automatiquement lors de la résolution du nom.

```
import java.rmi.*;
public class Client {
   public static void main(String args[]) {
      try {
         // Construction de la chaîne d'accès
                                 à l'objet serveur
         ItfObjetDistant obj = (ItfObjetDistant)
                  Naming.lookup("//site_execution_
                               serveur/mon_serveur");
         // Appel d'une méthode sur l'objet distant.
         PhoneNumber phone = obj.Lookup("Michel");
      } catch (Exception exc) {...}
      ...
```

La compilation du serveur se fait à l'aide des étapes suivantes :

```
javac ItfObjetDistant.java Client.java
```

On peut remarquer qu'en Java RMI, la production des talons client et serveur s'est effectuée uniquement sur le site du serveur. Seule l'interface des services distants est nécessaire sur le site du client.

La propriété de liaison dynamique de code permet de récupérer le code du talon client uniquement lors de la résolution de la chaîne d'accès. Pour cela, il faut, lors du lancement du serveur, préciser l'URI d'un site Web contenant le code du talon client et avoir auparavant déposé à cette adresse le code correspondant. Le lancement du client se fait comme pour un programme Java standard : java Client.

4.3.3 Évaluation

Java RMI est un bon exemple de l'intégration d'un appel de procédure dans un langage de programmation. Les **points forts** de cette approche sont les suivants :

— RMI est très simple à mettre en œuvre et les concepts utilisés sont faciles à comprendre. RMI fait partie intégrante de Java et ne nécessite aucun autre outil que le JDK (Java Development Kit). Un client RMI n'a besoin de connaître que l'interface de l'objet. Les classes contenant le talon client sont chargées, éventuellement depuis un site distant, à l'exécution ;

— RMI permet une gestion fine de la sécurité, pas toujours simple à comprendre. Il bénéficie de la sécurité de Java et d'un

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

RMISecurityManager qui doit être paramétré pour gérer par exemple des listes d'accès ;

— Java RMI est un mécanisme de base qui permet de construire d'autres modèles de programmation de plus haut niveau, par exemple JavaSpace [18] pour la construction d'un espace de données partagées ou les aglets [19] pour la programmation d'agents mobiles.

Dans ce tableau quelque peu idyllique, des **points faibles** subsistent. Java RMI ne permet que l'appel de méthodes sur des objets Java. Des outils sont en cours de définition pour permettre la coopération avec des serveurs Corba : JavaIDL permet la traduction d'une interface Java vers une interface IDL Corba ; le protocole utilisé par RMI a été porté sur IIOP afin de permettre l'interconnexion avec des objets Corba. Java RMI est lent et la sérialisation Java est un mécanisme redoutable car il permet de transporter de très gros volumes de données sans s'en rendre compte. Les techniques de compilation à la volée (*just-in-time*, JIT) devraient permettre d'atteindre des vitesses proches de celle d'un code compilé.

Attention, comme dans la plupart des RPC, les outils de développement et de mise au point sont rudimentaires.

5. Performances des RPC

Il est difficile de citer des chiffres précis concernant les appels de procédures tant les performances des processeurs et des réseaux évoluent rapidement.

Dans un premier temps, nous citons des résultats un peu anciens (1990) [20], mais dont les éléments d'analyse sont toujours d'actualité. Sur cette architecture spécifique — le Firefly —, le coût d'exécution du client et serveur (hors communication) était de 606 ms, le coût d'un RPC nulle était de 2 514 ms et celui d'un RPC avec un paquet d'appel utile de 1 Ko de 6 524 ms (figure **15**).

Indépendamment de ces coûts directs, les auteurs ont aussi évalué les facteurs d'amélioration possibles (tableau **1**).

Tableau 1 – Facteurs d'amélioration d'un appel de procédure

Appel de procédure	0 octet	1 440 octets
Vitesse des processeurs × 3	52 %	36 %
Attente active	17 %	7 %
Contrôleur amélioré	11 %	28 %
Protocole de transport en assembleur	10 %	4 %
Protocole amélioré	8 %	3 %
Pas de contrôle d'erreur	7 %	16 %
Vitesse du réseau × 10	4 %	18 %
Suppression des couches IP/UDP	4 %	1 %

Les principales causes d'amélioration d'un appel de procédure ne viennent pas d'une optimisation à outrance des piles de protocoles ou des mécanismes systèmes utilisés mais de l'évolution « naturelle » de la vitesse des processeurs (si la taille des paramètres est faible) ou du débit du réseau si les paramètres sont de taille importante.

La figure **16** donne un ordre de grandeur de l'évolution du coût d'un appel RMI en fonction de la taille des paramètres. À ces différentes mesures, il faudrait ajouter si nécessaire le coût de la sérialisation Java des objets à passer en paramètres.

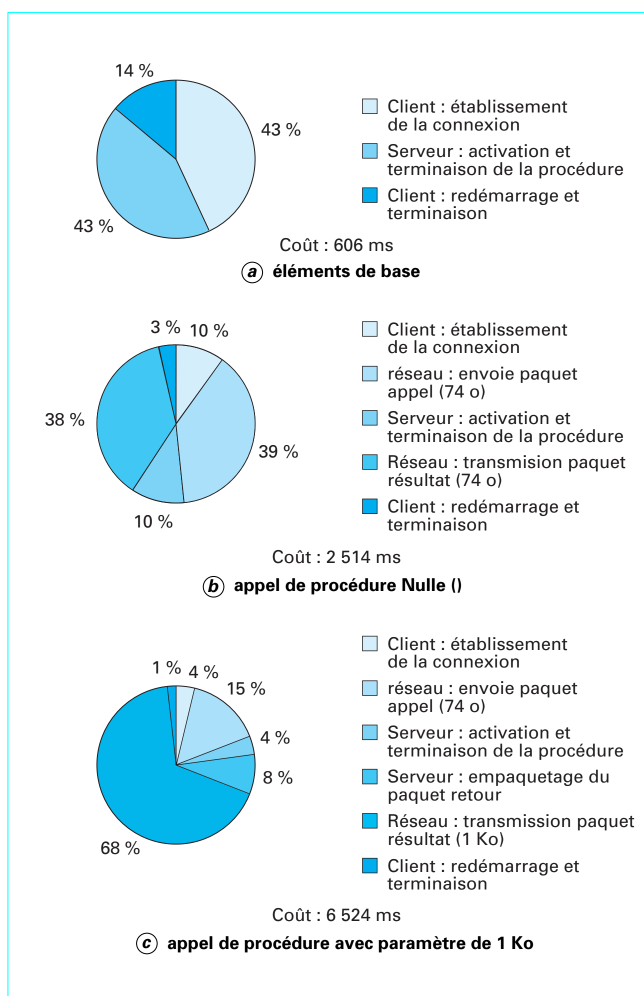


Figure 15 – Mesure des différents éléments d'appels de procédure

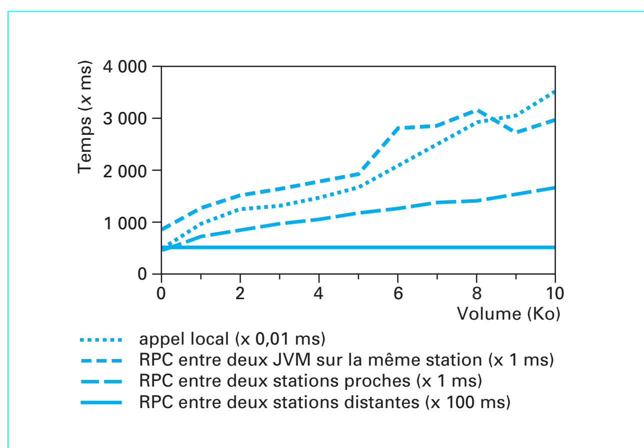


Figure 16 – Évolution des coûts d'un appel Java RMI en fonction de la taille des paramètres

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

6. Conclusion

L'appel de procédure à distance (ou son équivalent dans le monde objet : l'appel de méthode à distance) est un modèle de structuration qui permet de décrire l'interaction entre deux composants logiciels qui communiquent par échange de messages. Un des deux composants, le client, a l'initiative du dialogue et reste bloqué pendant que l'autre composant, le serveur, traite la requête.

Ce modèle est sans doute le plus répandu aujourd'hui pour la mise en œuvre d'applications réparties. La raison principale de ce succès tient au fait que l'appel de procédure à distance s'efforce de reproduire, pour un environnement réparti, les conditions d'un appel de procédure local — à la fois sur le plan du modèle de programmation et sur le plan du modèle d'exécution. Tout au long de cet article, nous avons montré que cet objectif de compatibilité sémantique entre l'appel de procédure à distance et l'appel de procédure local était difficile, voire impossible, à réaliser. L'émergence des langages de description d'interface (IDL) pour spécifier le contrat entre la partie cliente et la partie serveur d'une application est un élément clé du modèle client-serveur. D'une part, l'IDL permet de s'abstraire d'un langage de programmation particulier pour la programmation du client ou du serveur. D'autre part, les outils de génération de codes associés à l'IDL simplifient de façon significative la programmation des interactions entre le client et le serveur.

Ce modèle de programmation est aujourd'hui largement utilisé pour accéder à des services distants comme par exemple les moniteurs transactionnels. Dans ce cadre, deux « normes » ou méthodes d'accès coexistent. Pour illustrer la première approche, nous pouvons citer la proposition X/Open pour laquelle l'accès aux services a été défini par un ensemble de procédures regroupées dans une API (Application Programming Interface). Dans la seconde, les fonctions d'accès au moniteur transactionnel sont décrites par une description IDL et c'est ce qui a été fait pour décrire l'OTS (Object Transaction Services) de Corba.

Le modèle client-serveur et l'appel de procédure à distance présentent néanmoins un certain nombre de limitations qu'il convient de rappeler ici. La première d'entre elles concerne le schéma de synchronisation entre le client et le serveur (appel synchrone avec blocage du client jusqu'au retour du résultat). Dans certains cas, le client n'attend pas un résultat de l'exécution du service (par exemple, dans le cas d'un service d'impression). Dans d'autres situations, il faudrait que le flot d'exécution associé au client puisse continuer après envoi de la requête au serveur, avec la possibilité de s'enquérir d'un éventuel résultat plus tard, à un instant décidé par le programme du client. Ces extensions du modèle client-serveur ont été introduites sous la forme d'un **appel de procédure asynchrone** (notons au passage le côté paradoxal de l'expression). Dans ce type d'appel, la requête d'appel spécifie en outre un objet « futur » où doit être déposé le résultat de l'exécution du service. L'objet « futur » est consulté par le processus client pour prendre connaissance du résultat. D'autres variantes ou évolutions du modèle client-serveur ont été proposées dans la littérature [21]. Nous pensons que l'utilisation d'un RPC asynchrone ne doit pas être encouragée dans la mesure où il existe des modèles de programmation asynchrones, fondés sur l'échange de messages ou sur le paradigme événement/réaction, qui sont plus appropriés à la mise en œuvre d'applications réparties dans lesquelles le découplage temporel entre client et serveur est une nécessité.

Un autre problème, souvent évoqué pour le modèle client-serveur, concerne le coût du mécanisme. Cette observation a amené les concepteurs à étudier diverses formes d'optimisation pour réduire ce coût chaque fois que possible. Ainsi, lorsque l'appel de procédure s'effectue entre deux processus qui sont sur la même station de travail (mais dans des espaces d'adressage différents), il est possible d'envisager un modèle de RPC plus léger permettant de ne pas traverser toutes les piles de protocoles et d'utiliser une zone mémoire du noyau présente dans les deux processus pour effectuer le passage de paramètres. Cette forme d'optimisation, connue sous le terme « RPC léger » (*lightweight RPC*), est étudiée en détail dans [22].

L'appel de procédure à distance (RPC) est un mécanisme de « bas niveau » qu'il est nécessaire de compléter par différents services additionnels pour la construction d'applications réparties opérationnelles. Citons un service de désignation, des services de sécurité, un service de gestion du temps ou encore un service de gestion de fichiers répartis. Ces services constituent eux-mêmes des applications réparties de complexité significative et présentent un caractère suffisamment générique pour être utilisés par de nombreuses classes d'applications. C'est pourquoi les principaux environnements client-serveurs (§ 1.3) sont constitués d'une offre *middleware* complète qui intègre un ensemble de services bâtis autour d'un bus logiciel de type RPC. C'est ainsi le cas pour DCE (Distributed Computing Environment), pour l'environnement Corba de l'OMG ou encore pour l'environnement COM/DCOM de Microsoft.

Les outils de développement associés au modèle client-serveur sont généralement limités à la génération automatique des talons et le programmeur a très peu d'aide pour le déploiement et la mise au point des applications construites selon ce modèle. Bien que d'un apprentissage très facile, l'utilisation du modèle client-serveur (et des objets répartis) est délicate car la plupart des constructions syntaxiques masquent la différence entre appel de procédure local et appel de procédure à distance. Dans les faits, nous avons vu que la sémantique est différente à cause des restrictions sur le passage de paramètres d'une part et en raison des incertitudes sur l'exécution de l'appel provoquées par les défaillances d'autre part. La sémantique précise d'un RPC est définie par la nature de son implémentation et le concepteur d'applications doit en tenir compte. Le concepteur d'applications doit prendre en charge explicitement un certain nombre d'aspects systèmes liés au fonctionnement de l'application répartie dans un environnement distribué hétérogène, sensible aux pannes et aux intrusions externes. Cela concerne, entre autres, le schéma de désignation, la mise en œuvre de la protection et le contrôle des accès concurrents, les mécanismes de tolérance aux pannes. Les paramètres de performance et de disponibilité du serveur imposent également des choix d'architecture (par exemple, la réplication du serveur pour accroître la disponibilité et mettre en œuvre une fonction de répartition de charge). La cohabitation des aspects fonctionnels du serveur (c'est-à-dire la description des services) et des aspects non fonctionnels (persistance, sécurité, performance) est un frein considérable à l'utilisation du modèle client-serveur. Cette observation milite en faveur d'une séparation claire de ces aspects. C'est une évolution qui se traduit aujourd'hui dans l'émergence des *modèles à composants*. Un composant définit un objet (ou un groupe d'objets) auquel des propriétés non fonctionnelles sont associées, qui sont prises en charge par des fonctions systèmes *ad hoc*. La programmation des services (les objets) et celle des fonctions systèmes est généralement confiée à des classes différentes d'utilisateurs (voire à des sociétés différentes). Un exemple d'environnement à composants pour Java est donné par les EJB (Enterprise Java Beans). Il y a également tout un champ d'activité à l'OMG pour faire évoluer le modèle d'objets Corba vers des composants.

Nota : à ce sujet, le lecteur pourra se reporter à l'article *Programmation par composants* [H 2 759].

Enfin, notons que le modèle client-serveur ne permet de décrire qu'une interaction élémentaire entre deux composants logiciels. Ce n'est pas suffisant pour décrire de façon globale tous les composants d'une application répartie. L'émergence des modèles à composants, présentée brièvement ci-dessus au sujet des propriétés non fonctionnelles, vise également à apporter des solutions à

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique

cette limitation actuelle du modèle client-serveur. Au contraire du modèle client-serveur pour lequel seule est décrite l'interface d'accès à des services, le modèle à composants introduit aussi la notion d'interface de sortie d'un composant. Cette interface décrit les autres composants qui sont nécessaires à l'exécution d'un composant donné, c'est-à-dire les composants qui sont susceptibles d'être appelés à leur tour par l'exécution des services d'un composant. On désigne ces composants sous le terme de « dépendances » d'un composant donné. Cela permet de définir, de proche en proche, tous les composants entrant dans la réalisation d'une application donnée. Cette description globale définit l'*architecture* de l'application. Elle est fournie soit par des règles de programmation imposées par le modèle de composants, soit en utilisant un langage déclaratif *ad hoc* entrant dans la famille des langages de description d'architecture (ADL : Architecture Description Language). Des outils de développement sont associés à ces nouvelles classes de langage pour aider le concepteur d'applications à définir les composants, à générer le code des talons, à déployer le code sur les sites d'exécution et à mettre au point l'application. Cette forme de programmation est désignée globalement sous le terme de « programmation à gros grain » (*programming in the large*) ; elle vise à compléter les formes de programmation traditionnelles, limitées aux codes du client et du serveur.

Références bibliographiques

[1] BIRREL (A.D.) et NELSON (B.J.). – *Implementing Remote Procedure Call.* ACM Transactions on Computer Systems, *2*(1), 39-59, fév. 1984.

[2] *RPC : Remote Procedure Call specification.* RFC 1050, avr. 1988.

[3] OSADZINSKI (A.). – *The Network File System (NFS).* Vol. 8. Computer Standards & Interfaces, Pays-Bas (1988).

[4] ROSENBERG (W.), KENNEY (D.) et FISHER (G.). – *Comprendre DCE.* Addison-Wesley (1993). http://www.osf.org/dce

[5] *OSF DCE : Introduction to OSF DCE.* Révision 1.1. Open Software Foundation (1995).

[6] *The Common Object request Broker Architecture.* Révision 2.0. Object Management Group (1995). http://www.omg.org

[7] GEIB (J.-M.), GRANSART (C.) et MERLE (P.). – *Corba : des concepts à la pratique.* InterÉditions (1997). http://corbaweb.lifl.fr

[8] GRIMES (R.). – *Professional DCOM Programming.* Wrox Press Ltd., Birmingham (1997).

[9] ROGERSON (D.). – *Inside Com.* Microsoft Press (1997).

[10] DOWNING (T.B.). – *RMI : Developing Distributed Java Applications with Remote Method Invocation and Object Serialization.* IDG Books, San Mateo, CA, États-Unis (1998).

[11] BERNERS-LEE (T.), FIELDING (R.T.), FRYSTYK NIELSEN (H.), GETTYS (J.) et MOGUL (J.). – *Hypertext Transfer Protocol HTTP/1.1.* RFC 2068, janv. 1997.

[12] SNODGRASS (R.). – The Interface Description Language : Definition and Use, Computer Science Press, Rockville, MD, États-Unis (1989).

[13] COMER (D.E.) et STEVENS (D.L.). – *Internetworking with TCP/IP.* Vol. 3 : *Client-Server Programming wand Applications, BSD Socket Version.* Prentice Hall (1993).

[14] *Corba Services : Common Object Services Specification.* Édition révisée. Object Management Group. 31 mars 1995.

[15] *XDR : External data representation standard.* Sun Microsystems, RFC 1014, juin 1987.

[16] *MIME. Part one : Format of Internet Message Bodies.* RFC 2045 (1990). *Part two : Media Types.* RFC 2046 (1990). *Part three : Message Header Extensions for Non – ASCII Text.* RFC 2047 (1990). *Part four : Registration Procedures.* RFC 2048 (1990). *Part five : Conformamance Criteria and Examples.* RFC 2049 (1990).

[17] *Network Programming Guide.* Sun Microsystems (1990).

[18] ARNOLD (K.), FREEMAN (E.) et HUPFER (S.). – *JavaSpace Technology : A Tutorial and Reference Guide.* Addison-Wesley, Reading, MA, États-Unis (1998).

[19] LANGE (D.B.) et OSHIMA (M.). – *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley Longman (1998).

[20] SCHROEDER (M.D.) et BURROWS (M.). – *Performance of Firefly RPC.* ACM Transaction on Computer System *8*(1), 1-17, janv. 1990.

[21] LISKOV (B.) et SHRIRA (L.). – *Promises : Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems.* Proc. of SIGPLAN, 260-267 (1988).

[22] BERSHAD (B.N.), ANDERSON (T.E.), LAZOWSKA (E.D.) et LEVY (H.M.). – *Lightweight remote procedure call.* ACM Transaction on Computer System, *8*(1), 37-55, janv. 1990.

Toute reproduction sans autorisation du Centre français d'exploitation du droit de copie est strictement interdite.
© Techniques de l'Ingénieur, traité Informatique