Objets répartis avec Ada 95

Laurent Pautet — Thomas Quinot — Samuel Tardieu

École Nationale Supérieure des Télécommunications
46, rue Barrault
75 634 Paris Cedex 13

{pautet,quinot,tardieu}@enst.fr

RÉSUMÉ. *Cet article propose un état de l'art sur la mise en œuvre de systèmes à objets répartis utilisant le langage Ada 95. Il présente l'annexe optionnelle pour les systèmes répartis de la norme Ada 95, et une implémentation de cette annexe,* GLADE*. Il décrit ensuite l'utilisation de la plate-forme* CORBA *en Ada 95. Enfin, il présente des solutions actuelles et des recherches en cours pour l'interopérabilité entre ces deux systèmes.*

ABSTRACT. *This article presents the state of the art in the implementation of distributed objet systems using the Ada 95 language. The Distributed Systems optional annex of the Ada 95 standard is first descsribed, as well as an implementation of this annex:* GLADE*. Usage of the* CORBA *platform in Ada 95 is then discussed. Present solutions and current research work on interoperation of both systems is finally presented.*

MOTS-CLÉS : *Ada 95, annexe des systèmes répartis, CORBA, interopérabilité*

KEYWORDS: *Ada 95, distributed systems annex, CORBA, interoperability*

1. Introduction

Le langage Ada est né d'un constat du ministère de la Défense américain : en 1975, il est apparu que ses services utilisaient plus de 450 langages de programmation différents, certains n'ayant même pas de nom. Un appel d'offres a été lancé, recherchant un langage unique et généraliste, adapté aussi bien aux besoins de gestion qu'aux applications scientifiques, ou encore aux programmes temps-réel et embarqués. Une norme préliminaire a été publiée en juin 1979, une version provisoire en juillet 1980, et la norme définitive a été adoptée en 1983 [RM 83]. Elle est devenue une norme internationale grâce à son adoption par l'ISO[1] en 1987 [RM 87]. La conception d'Ada est guidée par un souci d'intégrer dans le langage des pratiques de programmation respectueuses des principes de génie logiciel, et de faciliter le développement de larges projets, qui font intervenir de nombreuses équipes indépendantes, et qui doivent être maintenus longtemps après leur livraison.

En 1995, après un long processus de révision commencé en 1988, une nouvelle norme ISO correspondant au langage Ada a été adoptée [RM 95], et introduit un grand nombre de nouveaux concepts, notamment des fonctionnalités orientées objets, un modèle pour le développement d'applications réparties et de nouveaux mécanismes pour le temps-réel, tout en préservant une compatibilité ascendante avec la première version [INT 95].

Dans la partie 2, nous présentons l'une des annexes optionnelles au langage introduite par la révision de 1995 : l'annexe des systèmes répartis. La partie 3 décrit notre implémentation de cette annexe, qui comporte des optimisations destinées à en améliorer les performances, ainsi que des extensions permises par la norme, que nous proposerons pour une prochaine révision. La partie 4, enfin, après avoir introduit l'utilisation des systèmes à objets répartis CORBA en Ada 95, présente nos travaux en cours visant à rendre interopérables l'annexe des systèmes répartis et CORBA.

2. Ada 95 et l'annexe des systèmes répartis

2.1. *Principales caractéristiques*

Le langage Ada, depuis ses origines, et en application de son cahier des charges, a comme propriétés principales :

– Typage fort : notamment, plusieurs types de même représentation interne peuvent être incompatibles (par exemple un type entier et un autre allant de 1 à 5).

– Abstraction et encapsulation : un programme Ada est organisé en paquetages et en sous-programmes. Chaque paquetage contient une partie visible représentant l'interface à utiliser pour accéder à ses services, une partie privée contenant les détails des types utilisés dans la partie publique et un corps abritant la mise en œuvre des services du paquetage.

1. International Standards Organization.

– Concurrence : il est possible d'écrire en Ada des programmes comportant plusieurs tâches, une tâche étant un flot de contrôle s'exécutant indépendamment des autres, sauf en cas de synchronisation explicite. Chaque tâche possède sa propre pile et ses variables locales, mais peut accéder à toute variable se trouvant dans son champ de visibilité.

– Synchronisation : des interactions entre les tâches peuvent être créées par le biais de rendez-vous. Ces rendez-vous permettent l'échange d'information entre tâches de manière synchronisée.

– Facilité de lecture : contrairement à d'autres langages de programmation, l'utilisation d'Ada facilite les lectures multiples d'un code existant plutôt que celle d'écriture du code initial.

– Bibliothèques prédéfinies : des bibliothèques normalisées dans le manuel de référence permettent de disposer de certaines fonctionnalités courantes.

– Contrôle de l'élaboration : les différents paquetages formant une application Ada s'élaborent selon des règles précises ; leurs déclarations sont évaluées avant de lancer le programme principal afin notamment de positionner les variables à leur valeur initiale.

– Validation : tout programme prétendant être un compilateur Ada doit, au préalable, être validé par un institut indépendant. Cette validation est organisée autour d'une vaste suite de tests vérifiant le respect de la norme.

Parmi les nouveaux concepts introduits par la révision de 1995, on trouve :

– Programmation orientée objets : Ada supporte l'héritage simple (l'héritage multiple s'obtenant par combinaison de plusieurs mécanismes du langage), le polymorphisme et l'aiguillage dynamique (ou liaison dynamique) d'appels de méthodes. Ada 95 est d'ailleurs le premier langage objet normalisé par l'ISO.

– Unités de bibliothèque hiérarchiques : une **unité de bibliothèque** est un paquetage ou un sous-programme qui n'est pas imbriqué dans une partie déclarative. Autrement dit, il s'agit d'une unité d'abstraction du niveau le plus haut. Il est possible d'étendre un paquetage par la création d'un paquetage enfant, qui obtient la visibilité sur la partie privée de son père.

– Nouveaux types de pointeurs : à la différence des pointeurs d'Ada 83 qui ne pouvaient désigner que des entités allouées dynamiquement, Ada 95 autorise certains types de pointeurs à désigner des objets situés sur la pile.

– Objets protégés : la synchronisation a été enrichie grâce à l'introduction des objets protégés. Par certains aspects, ces objets ressemblent aux moniteurs [HOA 74] et aux régions critiques [BRI 72].

– Annexes optionnelles : ces annexes, qui ne sont pas obligatoirement mises en œuvre dans tout compilateur Ada validé, doivent, si elles sont fournies, respecter la norme. Des annexes existent entre autres pour le temps-réel et la répartition.

Ces caractéristiques font d'Ada un langage puissant adapté à un grand nombre de situations. Il satisfait aux exigences du cahier des charges initial, qui imposent la possibilité d'utiliser le langage dans tous les domaines de l'industrie.

2.2. *Annexe des systèmes répartis*

L'annexe optionnelle des systèmes répartis définit un moyen de développer des applications réparties sans sortir du cadre du langage ; notamment, toutes les règles concernant le typage fort et les exceptions restent applicables dans l'intégralité du programme réparti, sans considération de localisation des différents nœuds logiques.

Avant de présenter les idées principales ayant guidé la définition de l'annexe des systèmes répartis, nous définissons quelques termes qui seront utilisés par la suite.

2.2.1. *Définitions*

La suite de ce document utilisera couramment les termes de « directive de compilation », « type abstrait » et « méthode ou sous-programme abstraits ».

Une **directive de compilation** (ou « *pragma* » en anglais) est une construction syntaxique indiquant certaines propriétés au compilateur sans effet sémantique. Par exemple, la présence de la directive de compilation `Inline` appliquée à un sous-programme demande expressément au compilateur de ne pas générer d'appel à ce sous-programme, mais d'insérer le code du sous-programme à l'endroit de l'appel lorsque c'est possible.

Un **type abstrait** est un type que l'on ne peut pas instancier (ou « virtuel » dans d'autres langages). Son rôle est de définir un gabarit : les **types concrets** dérivés d'un type abstrait devront obligatoirement redéfinir toutes les **méthodes abstraites** du type de base. Un type ou une méthode abstraits sont identifiés par la présence du mot-clé `abstract`.

2.2.2. *Philosophie de l'annexe*

Certains systèmes, tels que CORBA, sont entièrement organisés autour de la notion de répartition. Inversement, l'annexe des systèmes répartis d'Ada a pour but de supprimer la barrière existant entre les programmes centralisés et les programmes répartis. Ainsi est-il trivial de passer de la version monolithique d'une application à sa version répartie par simple changement d'outil : les mêmes sources conviendront aux deux versions. Cependant, ce modèle ne permet pas toujours d'obtenir la version monolithique à partir de la version répartie d'une application.

Dès lors, la norme Ada n'introduit aucun nouveau mot-clé spécifique à la répartition ; seules quelques nouvelles **directives de compilation** permettent de décrire les propriétés supplémentaires de certaines unités de bibliothèque vis-à-vis de la répartition. La présence de ces directives garantit qu'une application monolithique pourra être transformée en application répartie, en vérifiant par exemple que les types utilisés entre les différents nœuds logiques sont transportables sans perte de signification. Par exemple, une valeur entière gardera la même signification sémantique sur toutes les partitions. À l'opposé, une tâche utilise des données et des contextes locaux qu'il est difficile de transporter sur un autre nœud logique.

L'annexe de systèmes répartis introduit la notion de **partition** comme étant un nœud logique, agrégat d'unités de bibliothèque (voir partie 2.1). Il distingue deux types de partitions :

1. Les partitions actives, qui possèdent un ou plusieurs flots de contrôle, émettent et reçoivent des requêtes dans le but de réaliser l'objectif de l'application répartie.

2. Les partitions passives, qui servent d'espace de stockage utilisé par une ou plusieurs tâches situées dans des partitions actives.

Tous les appels entre partitions actives se font, en Ada, par le biais d'appels de sous-programmes ou de méthodes d'objets répartis. Le langage n'offre pas la possibilité d'utiliser, pour la synchronisation, des rendez-vous répartis dont l'un des participants se trouve sur une première partition et l'autre sur une deuxième. Cette possibilité a été écartée afin de préserver de tout changement les exécutifs Ada existants (notamment temps-réel).

2.3. *Catégorisation des paquetages*

L'unité de base de répartition en Ada étant l'unité de bibliothèque (voir partie 2.1), il n'est pas possible d'avoir un paquetage présent en partie sur une partition et en partie sur une autre. Les paquetages, dans le langage Ada, sont catégorisés à l'aide de directives de compilation, dont la hiérarchie respecte la règle suivante : `Remote_Call_Interface` > `Remote_Types` > `Shared_Passive` > `Pure`, où $C_1 > C_2$ signifie qu'un paquetage de catégorie C_1 peut avoir une visibilité sur un paquetage de catégorie C_2.

Un paquetage catégorisé `Shared_Passive` ne pourra donc avoir de visibilité que sur d'autres paquetages de même catégorisation ou de catégorisation plus restrictive (`Pure`). En ce qui concerne les catégorisations `Remote_Call_Interface` et `Remote_Types`, les limitations ne s'appliquent qu'à la déclaration de ces paquetages ; aucune limite de dépendance sémantique ne s'applique à leur corps.

Les différentes catégories de paquetages signifient respectivement :

– `Remote_Types` : le paquetage ne contient, dans la partie visible de sa déclaration, que des types pouvant être transportés entre les différentes partitions. On pourra notamment définir des références sur sous-programmes ou sur objets distants. Les types transportables peuvent être utilisés par plusieurs partitions, et être transmis entre partitions en conservant leur sémantique. Un tel paquetage sera dupliqué sur toutes les partitions sur lesquelles il est référencé.

– `Remote_Call_Interface` : le paquetage contient dans sa partie visible la déclaration des sous-programmes pouvant être appelés à distance (ou « *Remote Procedure Call* » en anglais) et de types transportables. Un tel paquetage définit l'interface d'un service localisé sur un nœud particulier d'une application répartie : il ne peut être dupliqué.

– `Shared_Passive` : le paquetage ne contient aucun flot de contrôle par lui-même. Les variables définies dans la partie visible d'un tel paquetage seront accessibles depuis plusieurs partitions du système réparti à travers un support partagé éventuellement réparti comme une mémoire, un système de fichiers ou une base de données. De plus, dans ces deux derniers cas, la persistance est obtenue automatiquement. Un tel paquetage définit un corpus de données passif partagé par l'ensemble des partitions qui prennent part à une application répartie : il ne peut être dupliqué.

– `Pure` : le paquetage est garanti sans effet de bord et ne conserve aucun état interne. Il contient typiquement des définitions de types simples et les opérations primitives applicables sur ces types. Un tel paquetage sera dupliqué sur toutes les partitions sur lesquelles il est référencé.

Toute unité de bibliothèque sans catégorisation est qualifiée de normale et sera dupliquée sur toutes les partitions sur lesquelles elle est référencée. Les unités définissant des entités statiques comme les paquetages catégorisés `Remote_Call_Interface` et `Shared_Passive` ne peuvent être dupliquées à la différence des paquetages sans catégorisation ou catégorisés `Remote_Types` et `Pure`.

La directive de compilation `Asynchronous` peut également être appliquée à un sous-programme sans paramètre de sortie d'une unité de bibliothèque ayant la catégorie `Remote_Call_Interface`, ce qui a pour effet de rendre tout appel à ce sous-programme unidirectionnel. Toute levée d'exception est alors ignorée. L'exécutif doit garantir que l'appel est exécuté au plus une fois.

La directive de compilation `All_Calls_Remote` peut être appliquée à une unité de bibliothèque catégorisée `Remote_Call_Interface` de sorte que tout appel à un sous-programme distant de cette unité devra transiter par le sous-système de communication même si l'appel peut être résolu en local. Cette fonctionnalité peut se révéler fort utile lors de la mise au point de l'application alors qu'elle pas encore été répartie puisque les latences induites par la communication sont introduites.

2.4. *Références sur entités distantes*

Il existe deux types d'entités sur lesquels il est possible d'obtenir une référence ou pointeur distant en Ada : les types étiquetés limités privés, utilisés pour la programmation orientée objets, et les sous-programmes. Le premier cas est proche des objets répartis trouvés dans CORBA et RMI, alors que le second est plus spécifique à Ada.

2.4.1. *Références sur objets distants*

Rappelons qu'un **type étiqueté** (ou « *tagged type* » en anglais) est une classe dans la terminologie objet. Un **type limité** (ou « *limited type* » en anglais) est un type dont il n'est pas possible de copier les instances, l'opération d'affectation n'étant pas définie. Un **type privé** (ou « *private type* » en anglais) est un type dont la définition complète n'est pas accessible en dehors du paquetage le définissant et de ses paquetages enfants ;

notamment dans le cas d'un type composite, structure ou tableau, il n'est pas possible d'accéder aux différents champs ou enregistrements d'une instance.

Un type pointeur défini dans la partie visible d'un paquetage portant une des catégorisations `Remote_Types` ou `Remote_Call_Interface` est un pointeur sur objets distants si les deux conditions suivantes sont vérifiées :

1. il désigne une hiérarchie de types dont la racine est un type étiqueté limité privé (c'est-à-dire un type possédant les trois caractéristiques énoncées ci-dessus) ;
2. c'est un pointeur général, c'est-à-dire qu'il possède le qualificatif `all` (les pointeurs généraux en Ada peuvent désigner aussi bien des objets locaux ou globaux que des objets alloués dynamiquement ; voir section 2.1).

Une telle référence peut pointer aussi bien sur un objet local que sur un objet distant. Toute utilisation de celui-ci dans le cadre d'un appel de méthode dynamique (c'est-à-dire quand le code réellement exécuté dépend du type de l'argument polymorphe au moment de l'exécution) donne lieu à un double aiguillage : l'appel est d'abord transféré sur la partition sur laquelle a été élaboré l'objet. Ensuite, sur cette partition, l'appel est aiguillé vers la méthode correspondant au type local de l'objet.

Ces pointeurs, équivalents d'un point de vue sémantique à des pointeurs locaux, comportent toutefois deux limitations :

– il est interdit de les déréférencer en dehors d'un appel de méthode ;

– il est interdit de les convertir en un type pointeur local.

En effet, permettre ces deux opérations reviendraient à autoriser l'accès au contenu d'un objet potentiellement distant. Le caractère privé obligatoire du type pointé empêche toute modification des champs, excepté lors de l'appel de méthodes ; le caractère limité, obligatoire lui aussi, interdit les copies de l'objet.

La directive de compilation `Asynchronous` s'applique aux pointeurs sur objets distants et rend automatiquement unidirectionnel tout appel à une méthode d'objet réparti n'ayant aucun paramètre de sortie.

2.4.2. *Références sur sous-programmes distants*

Des pointeurs sur sous-programmes peuvent être déclarés dans les paquetages `Remote_Call_Interface` ou `Remote_Types`. Ces pointeurs deviennent automatiquement des pointeurs sur sous-programmes distants.

Comme les pointeurs sur objets répartis, les pointeurs sur sous-programmes distants connaissent quelques limitations :

– ils ne peuvent pointer que sur des sous-programmes appelables à distance, c'est-à-dire définis dans un paquetage catégorisé à l'aide de la directive `Remote_Call_Interface`;

– ils ne peuvent être convertis qu'en un autre type pointeur sur sous-programme distant.

2.5. *Traitement des exceptions*

Mis à part les changements introduits par l'utilisation de la directive de compilation `Asynchronous`, le traitement des exceptions dans une application répartie écrite en Ada 95 ne diffère aucunement de celui effectué dans une application monolithique. Dans un programme monolithique, une exception inconnue pour des raisons de visibilité peut être rattrapée par un traite-exceptions grâce à une alternative `when others`. Dans un programme réparti, il en est de même : une exception `E` levée sur une partition P_1 peut être inconnue sur une partition P_2 mais rattrapée à l'aide de cette même alternative.

Cependant, si cette exception est implicitement ou explicitement renvoyée sur P_1 ou sur toute autre partition sur laquelle `E` est connue, alors elle pourra être rattrapée par un traite-exception spécifique de type `when E` : le type de l'exception n'est pas perdu en cours de route, même si celui-ci est inconnu sur certaines partitions intermédiaires.

2.6. *Interface entre compilateur et exécutif*

Les effets sémantiques de l'annexe des systèmes répartis ayant été décrits, nous présentons rapidement la manière dont la répartition se concrétise pour un développeur de compilateur.

Le manuel de référence définit l'interface à utiliser entre le compilateur et le sous-système de communication de l'annexe des systèmes répartis. Un paquetage unique, `System.RPC`, définit certains types et sous-programmes qui devront être utilisés comme points d'entrée du sous-système de communication. Cette règle vise à permettre à l'utilisateur de choisir de manière indépendante le compilateur et le sous-système de communication.

Le sous-système de communication propose des services identiques à ceux d'un **ORB** (*Object Request Broker*) puisqu'il se charge :

– d'envoyer les requêtes d'appels de sous-programmes ou de méthodes à distance auprès d'un serveur,

– d'analyser ces requêtes chez le serveur pour en effectuer le traitement en déléguant éventuellement auprès de **tâches Ada anonymes**,

– d'envoyer éventuellement le résultat de ce traitement auprès du client qui débloquera la tâche responsable de l'appel à distance.

Cependant, la normalisation des différentes couches de l'annexe de systèmes répartis d'Ada 95 et de CORBA diffèrent comme le montrent les deux parties de la figure 1. D'autre part, le sous-système de communication de l'annexe propose certains services qui n'apparaissent pas dans la norme CORBA, comme l'avortement d'appels distants ou le calcul de terminaison globale répartie.

Enfin, le processus consistant à rassembler les différents paquetages en partitions est un processus post-compilatoire non normalisé. La norme laisse également une certaine place pour d'éventuelles extensions, l'équipe de normalisation étant consciente de ne pas explorer toutes les possibilités en la matière.

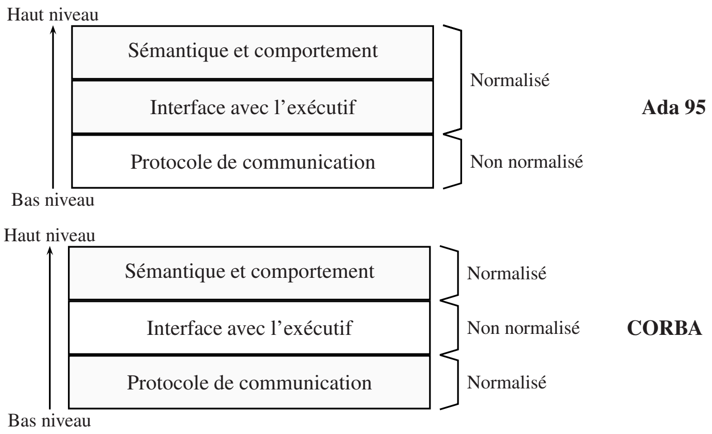

Figure 1. *Couches d'une application répartie en Ada 95 et en CORBA*

2.7. *Exemple d'utilisation de l'annexe*

```
package Correspondants is

   pragma Remote_Types;
   --  Les types définis dans ce paquetage sont transportables

   type Correspondant is abstract tagged limited private;

   procedure Envoyer (C : in Correspondant; M : in String) is abstract;
   --  Demande l'affichage du message M sur l'écran du correspondant C.

   type Ref_Correspondant is access all Correspondant'Class;
   pragma Asynchronous (Ref_Correspondant);
   --  Référence sur correspondant réparti échangée entre
   --  partitions. La procédure Envoyer sera automatiquement rendue
   --  unidirectionnelle grâce à Asynchronous.

private
   type Correspondant is abstract tagged limited null record;
   --  Le type concret reste à définir et à surcharger.
end Correspondants;
```

Figure 2. *Définition d'un gabarit d'objet réparti*

Dans cette section, nous développons un exemple de service d'échange de messages entre correspondants, en utilisant les fonctionnalités de l'annexe.

Sur la figure 2, nous définissons un paquetage `Remote_Types` décrivant les méthodes abstraites d'un correspondant abstrait. Il définit également un type de référence

```
with Correspondants; use Correspondants;

package Mes_Correspondants is

   pragma Remote_Types;

   type Mon_Correspondant is new Correspondant with private;

   procedure Envoyer (C : in Mon_Correspondant; M : in String);
   --  Cette procédure sera exécutée de manière unidirectionnelle en
   --  raison du pragma Asynchronous appliqué à Ref_Correspondant.

private
   type Mon_Correspondant is new Correspondant with null record;
end Mes_Correspondants;
```

Figure 3. *Définition d'objets répartis concrets*

```
with Correspondants; use Correspondants;

package Centre_De_Communication is

   pragma Remote_Call_Interface;
   --  Les sous-programmes définis dans ce paquetage pourront être
   --  invoqués depuis une autre partition de manière transparente.

   Déjà_Enregistré : exception;

   procedure Enregistrer (C : in Ref_Correspondant; N : in String);
   --  Enregistre un nouveau correspondant C de nom N auprès du
   --  serveur. Cette procédure n'est pas rendue unidirectionnelle car
   --  on veut rattrapper l'exception Déjà_Enregistré en cas de
   --  duplication de nom.

   function Rechercher (N : String) return Ref_Correspondant;
   --  Recherche un correspondant de nom N et retourne sa référence
   --  s'il existe. Retourne 'null' sinon.

   procedure Déconnecter (C : in Ref_Correspondant);
   pragma Asynchronous (Déconnecter);
   --  Déconnecte de manière unidirectionnelle un correspondant.

end Centre_De_Communication;
```

Figure 4. *Serveur de références de correspondants*

Ref_Correspondant sur tout objet réparti dérivé de ce type abstrait. Un utilisateur pourra développer un correspondant concret en dérivant ce type abstrait comme indiqué sur la figure 3. Afin de faire connaître la référence de son objet correspondant auprès du centre de communication, l'utilisateur devra utiliser la procédure Enregistrer et fournir un nom à son correspondant. Notons que si le paquetage Mes_Correspondants se trouve dupliqué sur plusieurs partitions, chaque instance sera considérée comme différente des autres et donc le type Mon_Correspondant ne pourra être confondu.

Sur la figure 4, un centre de communication se charge de gérer les différents correspondants. Ces correspondants s'inscrivent auprès du centre de communication en indiquant leur nom et une référence sur l'objet réparti qui va permettre d'établir un dialogue avec eux. Le centre de communication se comporte comme un serveur de noms semblable à ce que l'on aurait pu obtenir en utilisant un service de COS Naming. Le centre de communication ne peut être dupliqué. Il s'agit d'un paquetage `Remote_Call_Interface` qui constitue donc une entité statique.

```
with Ada.Command_Line;
with Correspondants;          use Correspondants;
with Centre_De_Communication; use Centre_De_Communication;

procedure Envoyer_Message is
   C : Ref_Correspondant;
begin
   --  Ce programme envoie le message donné en deuxième argument de la
   --  ligne de commande au correspondant nommé dans le premier
   --  argument, sous réserve que celui-ci soit enregistré.

   C := Rechercher (Ada.Command_Line.Argument (1));
   --  Recherche le correspondant auprès du centre de
   --  communication. Cet appel sera automatiquement transféré sur la
   --  partition sur laquelle le paquetage Centre_De_Communication a
   --  été élaboré.

   if C /= null then
      Envoyer (C.all, Ada.Command_Line.Argument (2));
      --  Cet appel sera automatiquement transféré sur la partition où
      --  le correspondant a été créé par l'intermédiaire de deux
      --  aiguillages l'un distant et l'autre local.
   end if;
end Envoyer_Message;
```

Figure 5. *Appel de méthode sur objet distant*

Sur la figure 5, nous voulons qu'un client communique avec un correspondant donné. Pour cela, le client doit retrouver la référence de l'objet réparti en utilisant son nom. Pour exécuter une méthode d'un objet à partir de sa référence, il lui suffit de déréférencer cette référence (par l'instruction **C.all**) dans le cadre d'un appel sur une de ses méthodes, par exemple, Envoyer[2].

La deuxième partie de l'exemple présente un aspect tout-à-fait original de l'annexe des systèmes répartis : les objets partagés (voir section 2.3).

Nous cherchons désormais à définir un tableau sur lequel nous écrirons 10 messages. Une écriture effacera le message le plus vieux pour le remplacer par le nouveau. Nous aurons également la possibilité de gérer un numéro de version afin de surveiller régulièrement si de nouveaux messages ont été écrits.

Sur la figure 6, nous définissons un objet tableau à cohérence forte qui sera localisé sur un support partagé. La cohérence forte est assuré par le type protégé (voir section 2.1). Nous définissons également un objet version à cohérence faible, cet objet

2. L'appel d'une méthode *m* sur un objet *o* s'obtient par *m (o)* en Ada 95 et par *o.m()* en C++ ou JAVA.

n'étant pas protégé. Ces deux objets sont localisés sur un support partagé puisqu'ils sont définis dans un paquetage `Shared_Passive`.

Pour garantir la cohérence forte, le compilateur se charge de verrouiller l'objet en faisant appel à un mécanisme propre au support partagé. Dans le cas d'un système partagé de fichiers de type NFS [COR 93], l'exclusion mutuelle sera obtenue en déposant un verrou sur le fichier. Les manipulations sur le tableau seront éventuellement soumises à des méthodes d'emballage et de déballage de sorte que le partage de ces données pourra s'effectuer entre des machines hétérogènes.

Pour enregistrer un nouveau message dans le tableau, nous faisons appel à la méthode Écrire de l'objet protégé sur la figure 7. Au préalable, le numéro de version sera incrémenté. Ces deux opérations s'effectueront directement sur le support partagé.

3. GLADE

3.1. *Présentation*

Ada Core Technologies[3] propose à la communauté Ada un compilateur Ada 95 nommé GNAT qui fournit toutes les fonctionnalités définies par la norme. Ce compilateur libre appartient à la famille des compilateurs GCC. Une conséquence majeure est qu'il est relativement aisé de faire de GNAT un compilateur croisé pour n'importe quelle plateforme où GCC est disponible.

GLADE[4] est l'un des outils disponible dans la chaîne de production de GNAT et permet le développement d'applications réparties à partir de l'annexe des systèmes répartis d'Ada 95. Il se compose d'un outil de configuration appelé GNATDIST et d'un sous-système de communication entre partitions appelé GARLIC pour Generic Ada Reusable Library for Inter-partition Communication. Cet environnement permet de construire aisément une application répartie pour un ensemble de machines hétérogènes ou non.

3.2. *Gnatdist*

Le processus consistant à rassembler les différentes unités de bibliothèque en partitions est un processus post-compilatoire non normalisé. L'outil GNATDIST et son langage de configuration ont donc été conçus pour permettre à l'utilisateur de partitionner facilement son application répartie [GAR 96].

Le langage de configuration de GNATDIST est un langage purement déclaratif syntaxiquement proche d'Ada 95 [KER 96]. Il permet d'affecter les unités de bibliothèque

3. `http://www.gnat.com/`
4. La distribution actuelle de GLADE a été développée par l'ENST ; elle est maintenue par ACT Europe. Voir `http://www.act-europe.fr/`.

```
package Centre_De_Messagerie is

   pragma Shared_Passive;
   --  Ces données sont partagées par toutes les partitions qui les
   --  référencent. Pour assurer une cohérence forte, Tableau est un
   --  type protégé. Version se contente d'une cohérence faible.

   type Ligne is record
      Contenu  : String (1 .. 80);
      Longueur : Positive := 0;
   end record;
   type Texte is array (1 .. 10) of Ligne;

   protected Tableau is
      procedure Ecrire (L : String);
      --  Ajoute de manière atomique une ligne sur le tableau.

      function Lire return Texte;
      --  Retourne de manière atomique le texte écrit sur le tableau.
   private
      T : Texte;
   end Tableau;

   Version : Natural := 0;

end Centre_De_Messagerie;
```

Figure 6. *Définition d'objets partagés*

```
with Ada.Command_Line, Ada.Text_IO;
with Centre_De_Messagerie; use Centre_De_Messagerie;

procedure Diffuser_Message is
   T : Texte;
begin
   --  Utilisé avec un argument sur la ligne de commande, ce programme
   --  va écrire dans un tableau blanc. En tous les cas, il va lire et
   --  afficher le contenu du tableau blanc.

   if Ada.Command_Line.Argument_Count /= 0 then
      Version := Version + 1;

      Tableau.Ecrire (Ada.Command_Line.Argument (1));
      --  Ajoute un message dans le tableau partagé par plusieurs
      --  partitions sans utiliser d'appels de méthodes à distance.
   end if;

   --  Lit le tableau directement à partir de la zone de donnée
   --  partagée par les partitions et l'affiche.

   T := Tableau.Lire;
   for L in T'Range loop
      Ada.Text_IO.Put_Line (T.Contenu (1 .. T.Longueur));
   end loop;
end Diffuser_Message;
```

Figure 7. *Utilisation d'objets partagés*

à des partitions et assure la cohérence de la configuration. L'utilisateur échappe ainsi à une connaissance trop détaillée du processus de production des partitions.

Grâce au langage de configuration, l'utilisateur peut également configurer des services ou des extensions propres à GARLIC comme le filtrage de certains canaux de communication entre partitions ou encore les nombres minimum et maximum de tâches anonymes disponibles sur une partition (voir définition en 2.6).

L'outil va donc se charger de :

– produire les différents souches et squelettes si nécessaire,

– puis en fonction de la configuration, pour chaque partition,

  - choisir d'inclure la souche ou le squelette des unités de bibliothèque,
  - configurer les services ou les extensions demandés,
  - effectuer l'édition de liens en incluant le sous-système de communication.

3.3. *Garlic*

Une bibliothèque logicielle, GARLIC, constitue le sous-système de communication de haut niveau de GLADE [KER 95]. Il permet la mise en œuvre de l'interface décrite par le manuel de référence (voir section 2.6) et se charge de la gestion des appels de sous-programmes ou de méthodes à distance, de l'emballage et du déballage des paramètres en utilisant **XDR** (*eXternal Data Representation* [SUN 90]), de l'envoi et la réception des requêtes et de l'interaction avec les couches de communication orientées objets comme le montre la figure 8.

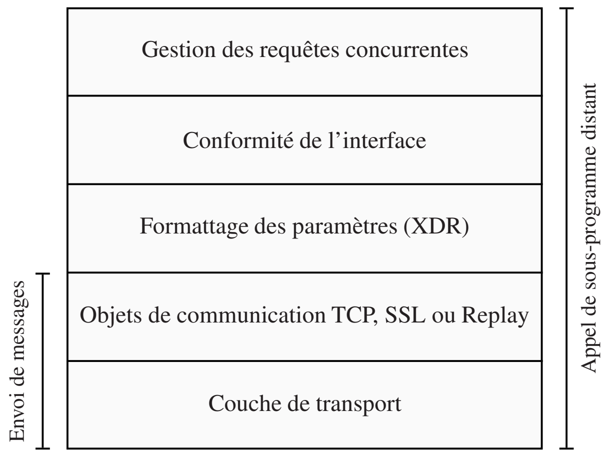

Figure 8. *Architecture de Garlic*

GARLIC a été conçu pour permettre à l'utilisateur de reconfigurer le sous-système de communication aussi bien statiquement que dynamiquement. Cela concerne, par exemple, le nombre de tâches pouvant supporter l'exécution d'appels de sous-programmes ou de méthodes à distance, le calcul de la terminaison de l'application répartie ou le contrôle de cohérence des spécifications entre client et serveur. Il peut également configurer les extensions que nous proposons, comme par exemple, les protocoles ou les filtres à utiliser pour communiquer entre partitions ou la réplication des serveurs internes.

3.4. *Compilation*

Les fonctionnalités pour la répartition étant parties intégrantes du langage, le compilateur GNAT est chargé de convertir toutes les références aux entités distantes (sous-programmes ou objets) en des appels appropriés au sous-système de communication [PAU 98]. Pour chaque invocation de méthode à un objet distant, le compilateur produit un code permettant de localiser l'objet et d'effectuer l'appel à la méthode distante. Pour des appels normaux à des sous-programmes distants, le compilateur localise le serveur distant en utilisant un serveur de noms interne.

Afin d'être conforme à la norme, le compilateur doit produire pour tout paquetage catégorisé `Remote_Call_Interface` des souches et des squelettes utilisant l'interface normalisée. Comme un tel paquetage contient des sous-programmes appelables à distance, on retrouve ici un schéma de production similaire à celui des modèles classiques d'appel de procédure à distance.

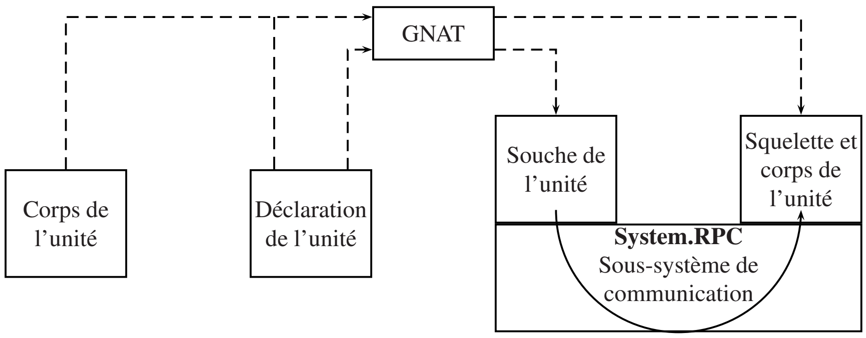

Figure 9. *Code nécessaire à l'appel de sous-programme à distance*

Les objets répartis sont accessibles de manière transparente par l'intermédiaire d'objets relais produits par le compilateur. Ces objets relais possèdent les mêmes méthodes que les objets issus de la racine de l'arbre d'objets répartis qui les concernent (voir figure 10). Chaque appel à une méthode virtuelle d'un objet réparti provoque un appel local à une méthode de l'objet relais, laquelle se charge de produire un appel distant à la méthode du véritable objet. Ceci aboutit à un double aiguillage : le premier aiguillage se charge, grâce à l'appel auprès de l'objet relais, de contacter la partition ayant créé l'objet réparti, le second consiste en un aiguillage classique de méthode sur objet [PAU 98].

L'un des avantages majeurs de cette technique est l'absence de surcoût à l'exécution lorsque l'appel s'effectue sur un objet local, puisque dans ce cas, l'objet relais et l'objet réel ne font qu'un, sauf dans le cas où le pragma `All_Calls_Remote` s'applique à l'unité courante. Aussi la présence de fonctionnalités de répartition ne ralentit-elle en rien une application Ada tant qu'elle n'est pas véritablement répartie. Ceci évite également d'avoir à produire des souches et des squelettes pour les unités de bibliothèque catégorisées `Remote_Types`.

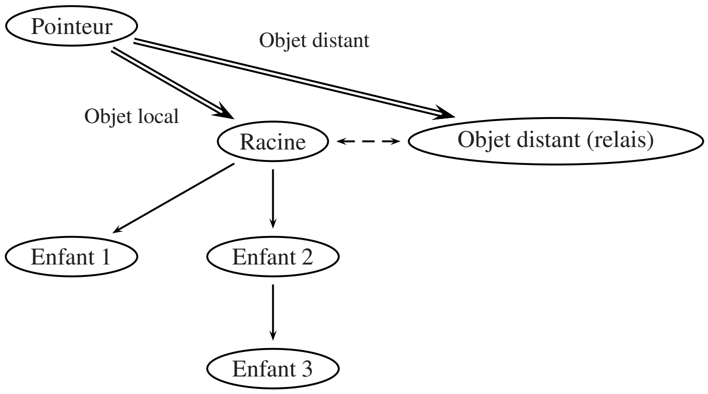

Figure 10. *Mise en œuvre d'un type d'objet distant*

3.5. *Communication*

GARLIC constitue en quelque sorte un ORB pour l'annexe des systèmes répartis d'Ada 95 (voir figure 9). Il inclut plusieurs services internes comme :

– un service de localisation de partition,

– un service de gestion d'unité de bibliothèque

– un service de détection de terminaison.

Le service de localisation de partition se charge d'attribuer un numéro unique à chaque partition, comme l'exige l'interface de la norme présentée en 2.6. Il se charge de sauvegarder la localisation de cette partition en termes de protocole et de données spécifiques à ce protocole utilisé pour contacter la partition (par exemple, protocole TCP et <numéro IP> :<port>). Il pourra renseigner toute partition cherchant à communiquer avec une partition pour laquelle elle n'a aucun renseignement. Il enregistrera également pour chaque partition des informations comme les filtres entre partitions ou les modes de terminaison des partitions.

Le service d'unité de bibliothèque se charge d'enregistrer le numéro de partition des unités de bibliothèque ainsi que leurs numéros de version ou d'autres informations internes. Il effectue le travail d'un service de noms comme ferait un service de noms tel que le COS[5] Naming. Le compilateur produit pour chaque unité de bibliothèque catégorisée `Remote_Call_Interface` ou `Shared_Passive` une portion de code visant à enregistrer chaque unité `Remote_Call_Interface` auprès de ce service afin que les références statiques puissent être résolues. Ce code est à rapprocher du code que doit rajouter un utilisateur CORBA pour enregistrer une référence d'objet auprès d'un service de noms.

Le service de terminaison active un algorithme de détection de terminaison globale répartie pour toutes les partitions ayant demandé à participer à la terminaison globale [TAR 99]. Certaines partitions peuvent demander à être exclues de cette détection afin de terminer lorsque la terminaison a été détectée localement, comme c'est le cas pour les partitions purement clientes.

3.6. *Mise au point*

L'analyse d'un programme comportant plusieurs flots de contrôle (« threads » en anglais) dans un langage concurrent est plus complexe que l'analyse d'un programme séquentiel. Dans le cadre d'une application répartie, les communications entre partitions ajoutent également à l'entropie du système. À des fins de mise au point, on peut vouloir, tout d'abord, tracer le comportement du sous-système de communication. En positionnant certaines variables d'environnement, l'utilisateur active ou désactive l'affichage de message d'information en provenance de l'exécutif.

Une autre fonctionnalité consiste, lors du développement d'une application, à pouvoir rejouer une exécution de manière similaire à l'exécution initiale ; cela revient à ajouter du déterminisme dans une application non-déterministe.

Une des techniques utilisées pour ce faire est la création, lors de l'exécution initiale, d'un fichier de traces contenant l'ordre dans lequel les événements ont eu lieu sur chaque partition. Ce fichier sera utilisé lors du rejeu pour réordonnancer les événements dans l'ordre initial. Cette technique, détaillée dans [LEB 86] et [MCD 89], se décline en deux variantes :

– Rejeu basé sur le contrôle : ce type de rejeu met en œuvre l'enregistrement du type d'événement et sa date d'arrivée.

– Rejeu basé sur les données : par rapport au précédent, ce type de rejeu enregistre en plus le contenu des paquets de données.

Nous avons choisi de mettre en œuvre dans GLADE le second type de rejeu, plus puissant que le précédent. [LEV 93] soulignent que la place disque nécessaire est plus

5. Common Object Service

importante que dans le premier type de rejeu, mais cela est largement compensé par le fait qu'il est possible de rejouer l'application répartie du point de vue d'une seule partition.

L'enregistrement s'effectue en donnant l'argument `-trace` sur la ligne de commande d'une partition. Des dérivations placées à des points stratégiques de GLADE permettent de sauvegarder les requêtes entrantes d'une partition. Le rejeu se fait par l'intermédiaire de l'argument `-replay`. Ce rejeu profite de l'architecture modulaire de GARLIC et substitue un protocole spécial, appelé Replay, à la place du protocole utilisé en première instance, par exemple, TCP [NER 97].

3.7. *Sécurité*

Comme il a été rapidement évoqué dans le paragraphe précédent, la conception de GARLIC s'est effectuée selon des principes orientés objets. En particulier, GARLIC définit la notion d'objet protocole, ce qui permet à tout développeur de rajouter d'autres protocoles de communication. Parmi les protocoles de communication fournis avec GARLIC, nous trouvons TCP et Replay. D'autres protocoles comme Serial pour ligne série ont été développés sans avoir été toutefois distribués.

Afin d'assurer la confidentialité des données échangées entre partitions, nous avons développé un module de communication particulier utilisant sur SSL[6]. GLADE a été enrichi afin de permettre l'utilisation de multiples protocoles entre partition. Ainsi une partition peut-elle communiquer avec une partition P_1 en utilisant SSL tout en communiquant avec une autre partition P_2 en utilisant TCP.

Une autre possibilité pour obtenir la confidentialité consiste à utiliser un mécanisme de filtrage. GLADE permet de spécifier un filtre à appliquer sur le canal de communication entre deux partitions [PAU 97]. Un exemple simple d'application consiste à compresser les données échangées entre deux partitions. D'autres filtres sont disponibles, comme ceux chiffrant les messages à partir d'une clé commune échangée grâce à un algorithme cryptographique à clé publique.

3.8. *Tolérance aux fautes*

Lorsque la tolérance aux fautes doit être supportée par un système, cela implique que l'exécutif offre cette fonctionnalité ainsi que l'application s'appuyant sur cet exécutif. Pour cela, GLADE fournit des fonctionnalités de tolérance aux fautes qui se complètent harmonieusement avec les fonctionnalités offertes par l'annexe des systèmes répartis d'Ada 95.

Par défaut, GLADE ne réplique pas ses serveurs internes comme par exemple les serveurs de noms. Lors de la configuration de l'application ou à l'exécution, l'utilisa-

6. Voir `http://www.netscape.com/eng/ssl3/`

teur peut demander à ce que certaines partitions se chargent d'assurer la réplication de ces serveurs internes. Dès lors, ces partitions se trouvent connectées par un anneau logique qui se charge d'assurer la cohérence des données en cas de panne. Les différents serveurs disposent d'un protocole permettant de régler les conflits et l'algorithme de [LEL 78] permet de traiter le cas de la perte du jeton.

En ce qui concerne la tolérance aux fautes du point de vue de l'utilisateur, l'annexe des systèmes répartis permet l'utilisation des paquetages `Shared_Passive`. Ces paquetages peuvent être configurés sur des supports partagés tolérants aux fautes comme un disque connecté par NFS ou mieux comme une base de données. On peut noter que cette fonctionnalité permet non seulement de traiter des problèmes de tolérance aux fautes mais fournit également une solution pour la persistance des données.

3.9. *Temps-réel*

La répartition d'une application se devait de conserver les nombreuses fonctionnalités temps-réel d'Ada 95. L'exécution supervisée d'un travail permet de stopper celui-ci lorsqu'une condition devient satisfaite, par exemple, lorsqu'un délai a expiré. Cette possibilité a été étendue au cas des applications réparties ; l'avortement, pour quelque raison que ce soit, d'un appel de sous-programme à distance provoque l'envoi d'un message de contrôle permettant d'avorter le travail en cours (au lieu de se contenter d'éliminer des résultats devenus inutiles).

La gestion des priorités constitue, pour sa part, un mécanisme plus complexe : en effet, chaque partition peut utiliser de manière indépendante une politique d'ordonnancement et des priorités différentes. L'utilisateur de GARLIC peut, au choix, propager la priorité de l'appelant à l'appelé en utilisant une table de correspondance permettant de maintenir l'ordre relatif des priorités de l'appelant, ou ne pas propager la priorité de l'appelant, auquel cas toutes les requêtes entrantes seront traitées avec un niveau de priorité constant défini par l'appelé.

Le nombre de tâches anonymes utilisées comme support d'exécution des requêtes entrantes (voir section 2.6) est contrôlé par l'utilisateur au moyen de trois paramètres. Le premier permet de contrôler le nombre total de tâches anonymes utilisées simultanément. Il est ainsi possible de garantir statiquement dans certains cas la possibilité d'ordonnancer l'application. Une valeur 1 permet également de sérialiser les appels entrants. Le second paramètre indique le nombre de tâches dont on souhaite qu'elles soient inoccupées et disponibles pour traiter de nouvelles requêtes entrantes, et permet de contrôler la réactivité de l'application. Le troisième paramètre permet d'indiquer le nombre de tâches anonymes maximum souhaité, mais non exigé, afin de limiter l'empreinte mémoire de la partition.

Des groupes de travail ont défini des profils pour Ada 95 ; ces profils constituent des sous-ensembles du langage, dont on a supprimé les constructions peu ou non déterministes, comme, par exemple, la création dynamique de tâches ou l'allocation dynamique de mémoire. Un tel profil est en cours d'élaboration pour GLADE et devrait être prochainement disponible, permettant ainsi de prédire plus finement le caractère temps-réel d'une application répartie.

3.10. *Contributions*

Si la norme Ada 95 exige le respect des règles du langage et de l'interface du sous-système de communication, elle laisse par contre une certaine liberté quant aux possibles extensions. Grâce à nos diverses contributions, nous avons enrichi le modèle proposé en respectant toutefois la norme dans son esprit. Nous avons, par exemple, proposé un mécanisme général de filtrage des flux de données entre partitions, permettant par exemple de compresser et de chiffrer les communications. Nous avons également défini et mis en œuvre différentes politiques de terminaison, et des mécanismes de reprise sur défaillance. Enfin, nous avons proposé un mécanisme de rejeu des échanges entre partitions, destiné à faciliter la mise au point des applications réparties. Nous espérons que ces contributions seront intégrées dans la prochaine révision de la norme.

De plus, Ada 95 propose des unités de bibliothèque prédéfinies offrant des services tels que les entrées/sorties ou la manipulation de chaînes. Par contre, aucune unité de bibliothèque prédéfinie n'a été proposée par la norme dans le cadre de l'annexe des systèmes répartis. En cela, l'annexe se différencie singulièrement, et en mal, de CORBA. Nous proposons donc un certain nombre de services communément utilisés pour la construction d'applications réparties, à inclure dans la prochaine révision. Certains, tels le service de dénomination ou le service d'événements, s'inspirent des services CORBA équivalents (COS Naming, COS Events). D'autres services, notamment un service de sémaphores répartis basé sur les principes décrits par [LI 86], sont de notre propre cru.

4. Interactions entre Glade et CORBA

4.1. *Interopérabilité*

Les nombreux modèles d'objets répartis actuellement proposés par des industriels ou par des groupes de normalisation, comme CORBA [OMG 98] (OMG) et l'annexe des systèmes répartis d'Ada 95, offrent chacun des fonctionnalités intéressantes mais disponibles uniquement dans le cadre d'un même modèle ; deux systèmes conçus sur deux modèles différents ne peuvent pas actuellement communiquer directement.

L'un de nos axes de recherche est l'interopérabilité entre ces différents modèles, avec pour objectif d'offrir aux concepteurs d'applications réparties les nouvelles fonctionnalités proposées par chaque plate-forme. Dans [PAU 99], nous avons présenté une comparaison entre les fonctionnalités offertes par CORBA et l'annexe des systèmes répartis d'Ada 95. Dans CORBA, les services fournis par un objet réparti sont décrits à l'aide d'un langage spécifique, OMG IDL. L'OMG a défini une projection du modèle d'objets et des définitions IDL vers différents langages hôtes tels que C++, Java et Ada. Un service défini par un contrat IDL peut être indifféremment mis en œuvre dans n'importe quel langage hôte, et être appelé à partir d'un client écrit dans n'importe quel autre langage : le mécanisme d'appel utilisé est indépendant des langages utilisés par le client et le serveur ; les clients et les objets s'interfacent, selon un schéma normalisé, avec un composant logiciel intermédiaire local chargé de la communication, l'ORB.

Les différents ORBs intervenant dans une application répartie communiquent entre eux pour échanger les requêtes des clients et les réponses des objets au moyen d'un protocole également normalisé, **GIOP** (*Generic Inter-ORB Protocol*). Des objets et des clients conçus indépendamment, et réalisés en utilisant des outils différents, peuvent ainsi interopérer facilement. Cette possibilité de faire interagir des environnements hétérogènes est un avantage certain de CORBA. Au contraire, l'annexe des systèmes répartis d'Ada 95 permet seulement de faire communiquer des objets Ada 95.

Pour pallier cet inconvénient, nous souhaitons, par exemple, offrir aux implémenteurs de services la possibilité d'utiliser Ada 95 et l'annexe des systèmes répartis, tout en permettant à des clients CORBA d'accéder à ces services. Par ce moyen, nous pouvons permettre le développement de services en Ada utilisant l'annexe, bénéficiant de ses propriétés de sûreté et de typage fort, mais aussi d'une ouverture vers de nombreux langages et systèmes hôtes apportée par CORBA.

Nous avons donc développé, et continuons à développer, des outils destinés à faciliter la coopération entre ces deux plates-formes pour les applications réparties orientées objets. Nous décrivons dans la suite de cette partie deux outils existants, ADABROKER et CIAO. Nous présentons également un projet en cours de spécification, qui consistera à réimplémenter l'annexe des systèmes répartis d'Ada 95 au-dessus de l'ORB d'ADABROKER.

4.2. *AdaBroker : une suite d'outils CORBA en Ada, pour Ada*

Les normes CORBA définissent une projection du langage OMG IDL sur Ada 95. Il est donc possible de créer des objets et des clients CORBA en Ada, au moyen d'un compilateur IDL vers Ada 95 et de l'ORB associé. Cependant, il n'existe pas actuellement de tels outils disponibles sous forme de logiciel libre, dont le code source soit disponible, et dont l'utilisation ne soit pas soumise à l'achat d'une licence payante. Cela met le langage Ada en position défavorable lorsqu'un développeur souhaite l'évaluer par rapport à d'autres langages comme C++, lors de la phase de prototypage d'un projet. De plus, les produits commerciaux dont le code source n'est pas disponible ne peuvent pas être utilisés dans des applications critiques où la disponibilité de l'ensemble des sources est nécessaire (par exemple dans le cadre d'une procédure de certification ou d'assurance qualité). Enfin, l'absence de produits libres nuit à l'utilisation de CORBA en Ada dans un contexte éducatif.

Si l'on utilise des langages tels que C, C++ ou Java, des ORBs et des compilateurs IDL libres sont disponibles et peuvent être utilisés par l'ensemble de la communauté. Nous avons décidé d'offrir un outil aux fonctionalités similaires pour les développeurs Ada : ADABROKER.

ADABROKER[7] comporte une pile GIOP qui a été entièrement écrite en Ada à l'ENST [GIN 99]. Elle intègre également un compilateur IDL vers Ada normalisé, et une implémentation complète du *Portable Object Adapter*. L'**Object Adapter** est le composant d'un ORB responsable de la création, de l'activation et de la destruction des implémentations d'objets. Dans les premières versions de CORBA, il n'existait qu'un seul *Object Adapter*, le BOA (*Basic Object Adapter*).

Dans un système utilisant le BOA, à un objet abstrait (au sens du modèle d'objets de CORBA) correspond exactement un objet d'implémentation (une structure concrète dans le langage cible utilisé pour l'implémentation). En revanche, si l'on dispose d'un POA, un seul objet concret peut représenter un nombre illimité d'objets CORBA. Par exemple, si une application stocke un ensemble d'enregistrements dans une base de données, et qu'on veut pouvoir manipuler individuellement chaque enregistrement comme un objet CORBA, alors une implémentation utilisant le BOA doit créer autant de structures en mémoire qu'il y a d'enregistrements, et enregistrer individuellement chacune de ces structures auprès de l'ORB. Avec le POA, il est possible de ne créer qu'un seul objet d'implémentation, et d'indiquer à l'ORB que cet unique objet physique incarne tous les objets abstraits CORBA représentant les enregistrements de la base de données.

Une plate-forme CORBA disposant d'un POA permet ainsi de créer beaucoup plus simplement des serveurs gérant de très grandes quantités d'objets. Le POA permet également au développeur de contrôler lui-même le mécanisme de création des objets CORBA, sans avoir besoin de notifier à l'ORB la création ou la destruction de chaque objet.

Nous utilisons actuellement ADABROKER comme plate-forme expérimentale pour les travaux de recherche sur l'interopérabilité entre CORBA et l'annexe des systèmes répartis d'Ada 95 ; le développement se poursuit donc activement. Nous envisageons notamment de doter ADABROKER d'un *Interface Repository*, service permettant à des clients de découvrir, au moment de l'exécution, quels sont les services fournis par des objets, ainsi que d'une implémentation de la DII (*Dynamic Invocation Interface*), qui permet d'effectuer des requêtes sur ces services découverts dynamiquement.

7. `http://adabroker.eu.org/`

4.3. *CIAO*

4.3.1. *Présentation générale*

CIAO[8], dont la première diffusion publique aura lieu dans les prochains mois, fournit un générateur automatique de passerelles pour l'accès à des services de l'annexe à partir de clients CORBA. Il s'agit du principal projet de recherche basé sur ADABROKER. Les requêtes émises par les clients CORBA sont traitées par une passerelle, c'est-à-dire un composant logiciel se comportant à la fois comme un objet CORBA, et simultanément comme un client dans l'annexe des systèmes répartis d'Ada 95.

Pour effectuer des requêtes sur un objet de l'annexe, l'utilisateur CORBA a besoin d'une description en OMG IDL du service offert. Ce contrat IDL peut être généré automatiquement si l'on dispose d'un schéma formel de traduction d'une spécification de paquetage de l'annexe vers OMG IDL. Nous générons ensuite une implémentation de ce contrat IDL, qui constitue la passerelle, et qui émet les requêtes de l'annexe correspondant aux requêtes CORBA qu'elle reçoit.

4.3.2. *Traduction des spécifications de services*

En Ada 95, l'abstraction entre l'interface (ou spécification) et l'implémentation d'un service est réalisée au moyen de paquetages. La déclaration d'un paquetage contient des déclarations de types de données et des signatures de sous-programmes, tandis que le corps du paquetage contient le corps de ces sous-programmes, ainsi que les structures internes dont ils ont besoin, mais qui ne doivent pas être exposées aux utilisateurs. L'annexe des systèmes répartis étend naturellement ce mécanisme, en permettant au programmeur d'indiquer que l'interface définie par une déclaration de paquetage doit être considérée comme « distante », au moyen de pragma de catégorisation (cf. partie 2.3).

Nous avons donc défini une application de l'ensemble des spécifications de services de l'annexe (*i. e.* l'ensemble des arbres abstraits Ada 95 correspondant à la déclaration d'une unité de bibliothèque ayant l'une des catégories `Pure`, `Remote_Types` ou `Remote_Call_Interface`) dans l'ensemble des contrats OMG IDL (*i. e.* l'ensemble des arbres syntaxiques OMG IDL).

Cette application est définie dans le même esprit que la projection canonique d'OMG IDL vers Ada. Les constructions « orientées objets » d'Ada sont traduites en leurs équivalents IDL : un type étiqueté privé pour lequel on a défini un type pointeur distant définit un type objet réparti, et le type pointeur distant permet de manipuler des références sur des objets de ce type. Nous représentons un tel type par une interface IDL, et ses sous-programmes primitifs sont représentés par les opérations de cette interface. Un type pointeur distant sur sous-programme est également représenté par une `interface` ayant une seule opération « `Call` ». Enfin, les sous-programmes appelables à distance d'un paquetage `Remote_Call_Interface` sont considérés comme

8. CORBA Interface for Ada (Distributed) Objects

```
#include "ciao.idl"

module CORBA__Correspondants {
   interface Correspondant;

   typedef Correspondant Correspondant_Ref;

   interface Correspondant {
      void Envoyer (in string M);
   };
};
```

(a) IDL `Correspondants`

```
#include "ciao.idl"
#include "correspondants.idl"

module CORBA__Centre_De_Communication {
  interface Remote_Subprograms {
    void
    Enregistrer
     (in ::CORBA__Correspondants
          ::Correspondant_Ref C,
      in string N);

    ::CORBA__Correspondants
      ::Correspondant_Ref
    Rechercher
     (in string N);

    void
    Déconnecter
     (in ::CORBA__Correspondants
          ::Correspondant_Ref C);
  };
};
```

(b) IDL `Centre_De_Communication`

Figure 11. *Exemple de traduction générée par CIAO*

les opérations d'une interface. Une interface « Constants » est également créée, et contient les traductions des constantes définies par un service Ada, sous forme d'opérations sans paramètres retournant la valeur de ces constantes.

La traduction doit également conserver la structure générale du service : à chaque paquetage Ada correspond un module IDL ; un paquetage de bibliothèque est traduit par exactement un fichier source IDL ; si des sous-paquetages sont présents, ils sont traduits par des modules imbriqués.

La figure 11 donne un exemple de traduction, correspondant au service de communication décrit dans la partie 2.7.

Nous avons choisi, autant que faire se peut, de rester proches de la projection canonique d'OMG IDL vers Ada. Nous en avons dévié chaque fois que cela semblait nécessaire ou souhaitable, en considérant que le schéma de traduction devait permettre la mise en œuvre automatisée d'une passerelle entre des clients CORBA et le service de l'annexe, et que l'IDL généré doit être aisément lisible par un programmeur souhaitant écrire un client CORBA.

4.3.3. *Réalisation du traducteur*

Après avoir défini un modèle de traduction des spécifications de service de l'annexe vers OMG IDL, nous avons réalisé un traducteur conforme à ce modèle (cf. [QUI 99]). Pour cela, nous avions besoin de disposer, sous une forme exploitable par un programme, de l'arbre sémantique d'une déclaration de paquetage Ada, en d'autres termes d'un mécanisme de réflexivité en Ada. Un tel mécanisme a été normalisé par l'ISO : ASIS [ISO 98] (*Ada Semantic Interface Specification*). Cette interface de programmation permettant à un outil de génie logiciel d'interagir avec un environnement de compilation Ada. En particulier, ASIS permet à une application de parcourir l'arbre syntaxique produit par un compilateur Ada, et d'obtenir des informations sémantiques sur cet arbre. L'utilisation d'ASIS nous permet de profiter des analyseurs syntaxiques déjà existants dans les compilateurs Ada, et nous évite de devoir développer notre propre analyseur.

Une mise en œuvre d'ASIS (spécifique à un environnement de compilation particulier) offre des types de données permettant de représenter des éléments Ada, ainsi que des fonctions permettant de parcourir l'arbre syntaxique (requêtes syntaxiques) et d'en extraire de nombreuses informations sémantiques. Un itérateur réalisant une traversée systématique en profondeur d'abord de l'arbre est fourni sous forme de la fonction *ASIS.Traverse_Element* ; le programmeur peut indiquer les opérations à effectuer sur chaque nœud au cours de la descente.

La traduction d'une spécification de paquetage Ada vers IDL se déroule en deux phases. Dans un premier temps, une descente de l'arbre sémantique est effectuée à l'aide de l'itérateur standard *ASIS.Traverse_Element*. Cet itérateur maintient un « état » interne, que les fonctions d'exploration peuvent modifier. Dans le traducteur, cet état comprend notamment l'arbre syntaxique IDL en train d'être construit, ainsi qu'un pointeur de « position courante » dans cet arbre.

Chaque élément Ada est exploré, et l'élément IDL correspondant est construit. Il est ensuite greffé à la position courante dans l'arbre IDL. S'il doit être rempli par des sous-nœuds, il devient alors l'élément courant, et le parcours de l'arbre Ada se poursuit. Lorsque le traitement d'un élément Ada est terminé, ainsi que celui de tous ses éléments enfants, le pointeur de nœud courant est restauré à la valeur qu'il avait à l'entrée dans l'élément. Nous construisons ainsi progressivement une image complète en mémoire de l'arbre IDL décrivant le service de l'annexe que l'on cherche à traduire.

4.3.4. *Génération de code*

La seconde phase de la traduction consiste à parcourir cet arbre IDL, et à produire en sortie le texte source correspondant. L'arbre est donc simplement parcouru en profondeur d'abord, en émettant le texte représentant chaque nœud au moment où il est traversé. Nous produisons également le code des différents paquetages mettant en œuvre le contrat IDL. Pour chaque interface de ce contrat, une implémentation est produite. Cette implémentation est conforme à la projection normalisée d'IDL en Ada « côté serveur » ; elle est destinée à s'agencer avec les squelettes produits par le traducteur IDL d'ADABROKER à partir du texte source IDL généré.

Le traitement réalisé par les implémentations de méthodes CORBA que nous générons se déroule en trois temps. En premier lieu, les arguments sont convertis des types issus de la projection normalisée de CORBA sur Ada 95 en les types natifs d'Ada, qui sont utilisés par l'annexe. La conversion des types de base est assurée par des fonctions de la bibliothèque d'exécution de CIAO. Pour les types composites, nous générons un paquetage de conversion par module, qui effectue la traduction des types

CORBA projetés en types natifs. Un appel de l'annexe est ensuite émis. Enfin, une éventuelle valeur de retour est retraduite en type CORBA et retournée à l'appelant.

La génération de code Ada nécessite une information précise sur le système Ada dans lequel ce code doit s'intégrer. La seule information disponible dans l'arbre IDL lui-même n'est donc pas suffisante ; nous utilisons alors des annotations sémantiques ajoutées à l'arbre abstrait IDL lors de la première phase, qui sont des pointeurs vers les éléments Ada d'origine. La génération du code est donc dirigée dans sa structure générale par l'arbre IDL, mais contrôlée dans certains détails fins par des informations sur la sémantique Ada obtenues à travers ASIS.

4.3.5. *Fonctionnement de la passerelle*

Le traducteur CIAO génère un arbre abstrait IDL, à partir duquel le générateur de relais produit des paquetages convertissant les types de données CORBA projetés en types natifs Ada, ainsi que des paquetages d'implémentation d'interfaces CORBA. Pour obtenir une application Ada répartie interopérable avec CORBA, il reste à agencer ces paquetages et à les enregistrer auprès d'un ORB, afin de constituer une *partition relais*, c'est-à-dire une partition au sens de l'annexe qui soit également un serveur incarnant des objets CORBA.

Chaque paquetage-relais correspondant à un paquetage de l'annexe comporte une procédure d'initialisation du service CORBA. La procédure principale de la partition relais commence par créer et initialiser un ORB, puis elle appelle la procédure d'initialisation des différents relais.

Pour un paquetage `Remote_Call_Interface`, cette procédure instancie l'objet d'implémentation et demande à l'ORB d'activer cette instance et de lui attribuer une IOR (*Interoperable Object Reference*). Elle associe ensuite cette référence à un nom facilement manipulable par l'utilisateur au moyen du service de nommage CORBA. Tout client CORBA peut ainsi obtenir une référence d'objet correspondant à l'interface qui traduit un paquetage `Remote_Call_Interface`.

De telles références sont essentielles pour l'initialisation de l'application. En effet, dans un système réparti, chaque participant doit utiliser un service du système pour prendre connaissance d'une « première » référence d'objet, qui lui permette ensuite d'interroger un service de nommage, un annuaire ou autre service fournisseur de références d'objets. Dans l'annexe, ce sont les paquetages `Remote_Call_Interface` qui jouent ce rôle, car un paquetage `Remote_Call_Interface` est instancié exactement une fois, sur une partition bien déterminée, dans une application Ada répartie ; à l'exécution, le sous-système de communication sait les localiser (voir section 3.5). Ils peuvent donc être utilisés pour fournir aux autres partitions le moyen d'enregistrer, de publier et d'acquérir des références d'objets.

Ainsi, dans une application utilisation CIAO, une référence CORBA est associée à tout paquetage `Remote_Call_Interface`. Un client CORBA peut alors interroger le service de nommage CORBA pour obtenir cette référence, puis effectuer des appels sur ce paquetage pour obtenir des références d'objets répartis.

Pour un paquetage définissant un type objet réparti Ada, le paquetage relais contient, là encore, une implémentation d'objet CORBA. Son activation utilise le POA (*Portable Object Adapter*), décrit dans la partie 4.2.

En l'occurrence, nous créons un seul *servant* pour tous les objets répartis Ada d'un même type. Nous enregistrons ce *servant* dans un POA créé spécialement, mettant en œuvre la politique USE_DEFAULT_SERVANT. Ainsi, tout appel de méthode sur un objet de cette classe sera traité comme un appel sur une méthode du *servant*. Lors de la conversion des arguments CORBA vers les types Ada correspondants, l'exécutif de CIAO extrait un identificateur d'objet. La spécification du POA permet d'utiliser toute séquence d'octets comme identificateur. Nous y indiquons donc tout simplement la référence d'objet de l'annexe, sous sa forme emballée par la bibliothèque d'exécution de GLADE. Cela permet de réaliser un relais sans état, que l'on peut redémarrer s'il se termine anormalement ; par ailleurs, nous n'introduisons pas de fuite de mémoire sur la partition relais.

4.4. *Vers une nouvelle implémentation de Glade*

L'intégration entre l'annexe des systèmes répartis et CORBA proposée par CIAO est asymétrique : un client CORBA émet des requêtes vers un objet de l'annexe. En revanche, nous n'offrons pas actuellement la possibilité aux clients utilisant l'annexe de faire appel à des objets CORBA. Nous cherchons maintenant à généraliser le mécanisme proposé pour parvenir à une interopérabilité symétrique entre les deux plates-formes.

Dans cette perspective, nous nous proposons d'évaluer expérimentalement l'utilisation de GIOP comme protocole sous-jacent pour implémenter l'annexe des systèmes répartis d'Ada 95. Cette expérimentation se fondera sur la modification de GLADE ; nous utiliserons ADABROKER comme ORB sous-jacent, et nous exploiterons et étendrons le travail fait sur CIAO pour projeter le modèle d'appel de l'annexe sur les entités du protocole GIOP. Nous pourrons ainsi rechercher si une interopérabilité totale et transparente est possible ; il se peut toutefois que ce résultat ne puisse pas être obtenu sans restriction sur le champ des possibilités de l'annexe, car celle-ci contient des fonctionnalités qui ne peuvent pas simplement être exprimées dans le modèle de traitement de CORBA ; l'annexe offre une sémantique beaucoup plus riche que les fonctionnalités offertes par un ORB. Certaines fonctions de haut niveau telles que le nommage correspondent, dans CORBA, à des services définis comme des objets décrits par des contrats IDL. L'implémentation de l'annexe sur GIOP nécessitera donc la définition en termes d'objets OMG IDL d'un certain nombre de services utilisés par l'exécutif de l'annexe :

– gestion des identificateurs de partition et de la localisation (qui pourra faire intervenir le service de nommage normalisé de CORBA) ;

– service d'avortement de requête en cours ;

– service de gestion de la terminaison des partitions.

5. Conclusion

Ada 95 est le premier langage objet à avoir été normalisé. Les fonctionnalités temps-réel déjà disponibles dans la norme précédente d'Ada 83 ont été enrichies par de nouvelles notions comme les types protégés. Le modèle objet a été intégré tout en prenant soin de préserver les propriétés temps-réel existantes.

Enfin, le langage introduit la possibilité de développer aisément des applications réparties. Le modèle cherche à atténuer les différences entre le mode de développement des applications monolithiques et celui des applications réparties (« *distribution boundary* » en anglais). Ce modèle permet d'allier harmonieusement les concepts d'orienté objets, de temps-réel et de répartition. De plus, ce modèle peut être qualifié de très riche à l'aune de modèles comme CORBA ou RMI. En effet, le modèle propose la notion d'objets partagés, c'est-à-dire d'objets se trouvant sur un support partagé.

Le point faible principal de ce modèle est l'absence de garantie d'interopérabilité. Pour cela, nos travaux de recherche visent à proposer une mise en œuvre de l'annexe des systèmes répartis sur un bus logiciel. Plusieurs étapes ont été franchies à ce jour. Nous disposons d'un environnement de développement CORBA pour Ada 95 sous forme de logiciel libre. De plus, nous avons développé le traducteur CIAO permettant de rendre accessibles des entités réparties Ada 95 à des utilisateurs CORBA.

Une autre activité de recherche de notre équipe consiste à proposer des extensions au modèle de répartition, en Ada 95 en vérifiant la pertinence de celles-ci par une mise en œuvre dans le cadre de GLADE, implémentation de l'annexe pour GNAT/GCC. Ces recherches s'orientent principalement sur le thème du temps-réel et s'effectuent en collaboration avec des groupes travaillant sur le temps-réel. Plus généralement, nous proposerons ces contributions lors de la prochaine révision de la norme.

6. Bibliographie

[BRI 72] BRINCH HANSEN P., « Structured Multiprogramming », *Communications of the ACM*, vol. 15, n° 7, 1972, p. 574–578.

[COR 93] CORBIN J., *The Network File System For System Administrators*, Sun Microsystems, Inc., Mountain View, Californie, USA, 1993.

[GAR 96] GARGARO A., KERMARREC Y., PAUTET L., TARDIEU S., « PARIS — Partitioned Ada for Remotely Invoked Services », *Lecture Notes in Computer Science*, vol. 1031, 1996.

[GIN 99] GINGOLD T., « Broca, an Ada CORBA implementation », Master's thesis, ENST Paris, 1999.

[HOA 74] HOARE C. A. R., « Monitors : An Operating System Structuring Concept », *Communications of the ACM*, vol. 17, n° 10, 1974, p. 549–557, Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.

[INT 95] INTERMETRICS, Ed., *Ada 95 Rationale : The Language, The Standard Libraries*, 1995.

[ISO 98] ISO, *Information Technology – Programming Languages – Ada Semantic Interface Specification (ASIS)*, ISO, 1998.

[KER 95] KERMARREC Y., PAUTET L., TARDIEU S., « GARLIC : Generic Ada Reusable Library for Interpartition Communication », *Proceedings Tri-Ada'95*, Anaheim, California, USA, 1995, ACM.

[KER 96] KERMARREC Y., NANA L., PAUTET L., « GNATDIST : a configuration language for distributed Ada 95 applications », *Proceedings of Tri-Ada'96*, Philadelphia, Pennsylvania, USA, 1996.

[LEB 86] LEBLANC T. J., MELLOR-CRUMMEY J. M., « Debugging Parallel Programs with Instant Replay », Technical Report n°TR194, 1986, University of Rochester, Computer Science Department.

[LEL 78] LELANN G., « Algorithms for Distributed Data Sharing Systems which use Tickets », *Third Berkeley Workshop*, 1978, p. 259–272.

[LEV 93] LEVROUW L. J., AUDENAERT K. M. R., « Reducing the space requirements of Instant Replay », *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, 1993, p. 205–207.

[LI 86] LI K., HUDAK P. R., « Memory Coherence in Shared Virtual Memory Systems », *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, New York, NY, 1986, ACM, p. 229–239.

[MCD 89] MCDOWELL C. E., HELMBOLD D. P., « Debugging Concurrent Programs », *ACM Computing Surveys*, vol. 21, n° 4, 1989, p. 593–622.

[NER 97] NERI D., PAUTET L., TARDIEU S., « Debugging Distributed Applications With Replay Capabilities », *Proceedings of the TRI-Ada'97 Conference, November 9–13, 1997, St. Louis, MO*, New York, NY, USA, 1997, ACM Press, p. 189–196.

[OMG 98] OMG, Ed., *The Common Object Request Broker : Architecture and Specification, revision 2.2*, February 1998, OMG Technical Document formal/98-07-01.

[PAU 97] PAUTET L., WOLF T., « Transparent Filtering of Streams in GLADE », *Proceedings of the TRI-Ada'97 Conference, November 9–13, 1997, St. Louis, MO*, New York, NY, USA, 1997, ACM Press, p. 11–21.

[PAU 98] PAUTET L., TARDIEU S., « Inside the Distributed Systems Annex », *Proceedings of AdaEurope'98*, Uppsala, Sweden, June 1998, Springer Verlag.

[PAU 99] PAUTET L., QUINOT T., TARDIEU S., « CORBA & DSA : Divorce or Marriage ? », *Proceedings of AdaEurope'99*, Santander, Spain, 1999.

[QUI 99] QUINOT T., « Mapping the Ada 95 Distributed Systems Annex to OMG IDL — Specification and implementation », Master's thesis, ENST Paris, Aug 1999.

[RM 83] *ANSI/MIL-STD 1815 A*, ANSI, 1983.

[RM 87] *Information Technology – Programming Languages – Ada*, ISO, 1987, ISO 8652 :1987.

[RM 95] *Information Technology – Programming Languages – Ada*, ISO, 1995, ISO/IEC/ANSI 8652 :1995.

[SUN 90] SUN MICROSYSTEMS, INC., « xdr – library routines for external data representation », 1990.

[TAR 99] TARDIEU S., « GLADE, une implémentation de l'annexe des systèmes répartis d'Ada 95 », Thèse de doctorat, École nationale supérieure des télécommunications, Oct 1999.