GLADE

Une implémentation de l'annexe des systèmes répartis d'Ada 95

Samuel Tardieu

Thèse soutenue le 26 octobre 1999

Remerciements

Les trois années qu'ont duré cette thèse m'ont permis de rencontrer un grand nombre de personnes qui m'ont apporté énormément, tant sur le plan personnel que sur le plan professionnel. Leurs passions, leurs compétences et leur goût pour la recherche m'ont motivé tout au long de ces travaux, tout en me permettant de m'orienter vers des pistes originales que je n'aurais peut-être pas eu l'heur d'explorer sans eux. Je remercie donc sincèrement...

Ed SCHONBERG, Professeur à l'Université de New-York et Fabrice KORDON, Maître de Conférences à l'Université de Paris VI, pour l'intérêt qu'ils ont manifesté pour mes travaux, et l'honneur et l'amitié qu'ils me font d'avoir accepté de rapporter le présent mémoire. Leurs commentaires judicieux sur la première version de ce document m'ont permis d'en renforcer la cohésion et d'éclaircir certains points techniques.

Laurent PAUTET, Maître de Conférences à l'École Nationale Supérieure des Télécommunications, pour m'avoir accueilli sous sa direction et m'avoir accompagné tout au long de cette thèse.

Christian CARREZ, Professeur au Conservatoire National des Arts et Métiers, pour l'honneur qu'il me fait d'avoir accepté de présider mon jury.

Elie NAJM, Maître de Conférences à l'École Nationale Supérieure des Télécommunications et Jean-François PRADAT-PEYRE, Maître de Conférences au Conservatoire National des Arts et Métiers, pour l'intérêt qu'ils ont porté à mes travaux et l'honneur qu'ils me font d'avoir accepté d'être membre de mon jury.

Les membres du Département Informatique et Réseaux de l'École Nationale Supérieure des Télécommunications, et tout particulièrement le groupe Architecture et Systèmes, pour m'avoir accueilli en leur sein dans une ambiance agréable.

Toute l'équipe d'ADA CORE TECHNOLOGIES et d'ACT EUROPE, avec qui j'ai entretenu une collaboration fructueuse durant ces trois années qui, je l'espère, continuera longtemps.

Mes derniers remerciements, et non les moindres, iront à mes relecteurs, qui ont traqué sans relâche et avec obstination les défauts du manuscrit jusqu'aux plus petits détails : Vincent BATAILLE (CNAM), Cécile BELMONDO (Université Paris Dauphine), Tristan GINGOLD (NetGem), Alexis POLTI (NetGem), Thomas QUINOT (Université Paris VI/ENST) et Nadine RICHARD (INRIA/ENST).

Résumé

D'ordinateurs imposants équipés de périphériques d'entrée-sortie élémentaires, nous sommes arrivés, en une cinquantaine d'années, à des systèmes parfois miniaturisés à l'extrême dont une des caractéristiques principales est d'être communicants. À l'ère d'Internet et des réseaux à haut débit, les systèmes répartis prennent une part de plus en plus considérable dans notre vie quotidienne : les distributeurs de billets de banque, la télévision interactive et le *World-Wide Web* sont autant d'exemples d'applications réparties.

Les architectes du langage Ada 95 ont tenu compte de cette évolution en intégrant la répartition au sein du langage : il est ainsi possible de construire une application répartie en Ada sans jamais sortir du cadre du manuel de référence standardisé par l'ISO. Cependant, cette facilité offerte au développeur d'applications a un coût, supporté à la fois par le compilateur et par un exécutif chargé de la communication entre les nœuds de l'application répartie.

Le présent mémoire de thèse présente Glade, notre implémentation de l'annexe des systèmes répartis d'Ada 95. Nous y traitons aussi bien des aspects relatifs à la compilation que de ceux liés à l'exécutif. Les questions posées par la localisation des ressources, la terminaison d'une application répartie et la résistance aux défaillances trouvent ici des réponses basées sur des algorithmes originaux. La réduction des ressources nécessaires en vue de l'utilisation sur des systèmes embarqués et la conservation des propriétés de temps réel sont abordées et des solutions sont proposées. Nous présentons également les extensions que nous avons apportées au modèle original, ainsi que des suggestions permettant d'améliorer la partie dédiée à la répartition de la future révision de la norme internationale définissant le langage Ada.

Mots-clés : Ada, répartition, GNAT, Glade, Annexe E

Table des matières

Table des figures

Liste des exemples

Introduction générale

Bref historique sur Ada et les systèmes répartis

D'ordinateurs imposants équipés de périphériques d'entrée-sortie élémentaires, nous sommes arrivés, en une cinquantaine d'années, à des systèmes parfois miniaturisés à l'extrême dont une des caractéristiques principales est d'être communicants. À l'ère d'Internet et des réseaux à haut débit, les systèmes répartis prennent une part de plus en plus considérable dans notre vie quotidienne : les distributeurs de billets de banque, la télévision interactive et le *World-Wide Web* sont autant d'exemples d'applications réparties.

Cette évolution des systèmes informatiques s'est accompagnée d'avancées dans le domaine des langages de programmation et des outils de développement. Les langages originels, d'abord prévus pour écrire des logiciels spécialisés de taille peu importante, ont révélé leurs insuffisances. De nouveaux langages se sont notamment tournés vers le Génie Logiciel et la réutilisabilité ; ils permettent aujourd'hui de développer des applications complexes au cycle de vie plus long. La nécessité de s'adapter à des marchés et à des environnements matériels et logiciels en perpétuel changement impose souvent une rigueur plus importante que celle mise en œuvre dans le cadre d'applications figées dédiées à une tâche précise. Le langage Ada, apparu au début des années 80, apparaît dans bien des cas comme le support idéal de ces exigences. Dès sa conception, il a pris en compte la réutilisabilité des composants logiciels, et sa structure se calque naturellement sur celle de méthodes de conception logicielle fortement répandues.

De même, au fil du temps, le besoin de formalisation et de portabilité a conduit au développement de nouveaux outils logiciels. Par exemple, alors que les premiers programmes répartis se satisfaisaient de protocoles dédiés à l'application, les outils de programmation actuels permettent de générer semi-automatiquement des prototypes, prototypes par la suite complétés ou modifiés en fonction de l'application qui en faisait usage.

La rencontre entre applications réparties et langages était donc inéluctable. C'est pourquoi la dernière révision du langage Ada, normalisée en 1995, intègre en son cœur la possibilité de définir des applications réparties sans avoir recours à des outils de communication externes. Les programmes répartis ainsi créés bénéficient, tout au long de leur phase de conception, des techniques de génie logiciel traditionnellement appliquées aux programmes monolithiques. Il est notamment possible de concevoir des composants logiciels répartis évolutifs et réutilisables, économisant ainsi les coûts induits par la création de nouveaux protocoles de communication spécialisés.

Certains avaient anticipé cette intégration entre langage et répartition dès le début des années 80 [Goldsack, 1992]. Des méthodes de conception tout particulièrement adaptés aux systèmes répartis utilisant le modèle orienté objet ont été créés, par exemple DRAGOON [Atkinson et Goldsack, 1989] [Atkinson, 1991]. Ils ont été ensuite partiellement repris par l'équipe chargée de définir la syntaxe et la sémantique des systèmes répartis dans le langage Ada. C'est ainsi que l'annexe E du manuel de référence [RM, 1995, Annexe E] contient aujourd'hui les constructions syntaxiques et sémantiques permettant de programmer une application répartie en Ada sans sortir du cadre du langage.

Objectifs

Cette thèse, commencée en octobre 1996, a un triple objectif : le premier est d'évaluer la faisabilité d'une implémentation de l'ensemble de l'annexe des systèmes répartis du manuel de référence Ada. Le second, de fournir une implémentation souple et robuste, utilisable à la fois dans des contextes industriels, militaires et d'enseignement-recherche. Le dernier point consiste à enrichir cette implémentation par de nouvelles extensions ou de nouveaux services, afin d'augmenter le champ d'application potentiel de ces systèmes répartis, et à proposer de nouvelles pistes en vue d'une prochaine révision de la norme Ada.

Apports et réalisations

Au terme de cette thèse, une implémentation complète de l'annexe des systèmes répartis pour le compilateur Ada GNAT est disponible. Ce logiciel, appelé GLADE, a été validé par un organisme de contrôle indépendant et reconnu, chargé de vérifier la conformité d'une implémentation Ada au vu de la norme. Il est distribué sous la forme d'un logiciel libre, accompagné de ses 55000 lignes de code source.

GLADE est actuellement utilisé pour l'enseignement et la recherche dans plusieurs écoles et universités, françaises comme étrangères. L'intégration de la répartition dans le langage fait d'Ada un support idéal pour modéliser les concepts des applications réparties (appels de sous-programmes à distance, objets répartis) tout en s'affranchissant de la connaissance des protocoles de bas-niveau. Les retours d'expérience que nous recevons nous ont convaincu, s'il en était besoin, des avantages d'une telle approche.

GLADE est également utilisé dans un contexte industriel. EDF, par exemple, l'utilise comme composant de son système d'information interne. Cependant, la jeunesse du produit ne nous permet pas encore de mesurer l'impact réel de cette technologie dans le monde des entreprises, celles-ci adoptant en général une politique plus conservatrice vis-à-vis des technologies novatrices, notamment à cause de leur existant.

Au cours de ces trois années, nous avons également conçu un certain nombre d'extensions au modèle défini par l'annexe des systèmes répartis, afin d'en enrichir les possibilités. Nous avons, par exemple, développé un service de filtrage transparent permettant de compresser ou de chiffrer les données échangées au vol. De nouveaux modèles de terminaison d'une application répartie offrent également la possibilité de concevoir des comportements complexes dans le cadre d'applications client-serveur. Nous avons également conçu et implémenté des politiques de réaction de l'application face aux défaillances de certains nœuds, permettant d'augmenter la résistance aux pannes.

Des services externes, tels qu'un verrou réparti sans point central ou un centre de nommage, permettent aux développeurs de tirer parti des capacités de réutilisabilité intrinsèques à Ada. Un service de transmission de flux multimédia permet également à l'utilisateur d'envoyer des données typées de manière sûre et efficace sur un réseau à haut débit.

Pour finir, le modèle d'applications réparties en Ada 95 est comparé à celui de CORBA, afin de déterminer comment améliorer la prochaine révision de la norme Ada en choisissant le meilleur des deux mondes.

Organisation du mémoire

Ce mémoire, organisé en cinq parties, décrit les méthodes utilisées, les problèmes rencontrés et les solutions qui y ont été apportées :

1. La première partie contient une introduction aux systèmes répartis en général et à ceux basés sur Ada en particulier.
2. La seconde partie expose les modifications opérées dans le compilateur GNAT afin d'intégrer la sémantique définie par l'annexe des systèmes répartis dans la chaîne de compilation.
3. La troisième partie décrit l'exécutif des systèmes répartis, en insistant sur certains points importants comme la terminaison ou l'avortement de requêtes distantes.
4. La quatrième partie offre une présentation des différents outils et extensions que nous avons été amenés à concevoir afin de rendre le développement d'applications réparties en Ada plus accessible aux développeurs.
5. La cinquième partie contient une critique de l'annexe des systèmes répartis ainsi que des propositions d'amélioration en vue de la prochaine révision de la norme définissant le langage Ada. Une comparaison avec CORBA est également présentée.

La conclusion générale récapitule les différents apports de cette thèse ainsi que les travaux qui peuvent y être rattachés.

PREMIÈRE PARTIE

Systèmes répartis et Ada

Dans cette partie, nous introduisons les concepts généraux utilisés dans la suite du document. Le chapitre 1 définit la notion de “système réparti” et tous les concepts connexes. Le chapitre 2 pour sa part présente très brièvement le langage Ada et son histoire, puis décrit les mécanismes de répartition présents dans le cœur même du langage.

1

Systèmes répartis

1.1 Définitions

Les nombreuses définitions d'un système réparti existantes diffèrent légèrement les unes des autres. Une des plus consensuelles est probablement [Coulouris et al., 1994] :

> *Un système réparti est constitué d'un ensemble d'ordinateurs autonomes connectés par un réseau et équipés d'un logiciel spécifique de communication.*

Néanmoins, cette caractérisation d'un système réparti se limite à des ordinateurs fortement interconnectés. Chaque ordinateur est relié à tous les autres par l'intermédiaire d'un maillage total constituant le réseau.

Dans le cadre de ce document, nous avons choisi d'adopter la caractérisation présentée dans [Schroeder, 1993]. Celle-ci, plus précise que la précédente, est celle correspondant le plus à notre conception :

> *Un* ***système réparti*** *est un ensemble d'ordinateurs travaillant ensemble ; un tel système possède trois propriétés :*
>
> **Plusieurs ordinateurs :** *un système réparti comporte au moins deux ordinateurs, chacun possédant un ou plusieurs microprocesseurs, de la mémoire vive locale, éventuellement de la mémoire de stockage (par exemple un disque dur), ainsi que des connexions d'entrée-sortie lui permettant d'être relié à son environnement.*
>
> **Des interconnexions :** *certaines de ces connexions sont utilisées pour relier les différents ordinateurs entre eux. Si les ordinateurs ne sont pas connectés ou ne communiquent pas, on peut toujours prétendre avoir un système réparti, mais celui-ci ne présentera aucun intérêt.*
>
> **Un état partagé :** *les ordinateurs collaborent afin de maintenir un état global. Notamment, si la bonne marche du système peut être décrite sous la forme d'invariants globaux, alors la conservation de ces invariants nécessite en général une opération coordonnée de la part des ordinateurs.*

Le terme d'**ordinateur** tel qu'il est employé ici ne se résume pas à la vision classique d'une station de travail, mais s'applique à n'importe quelle machine pourvue d'une unité de calcul ; un robot mobile connecté par une liaison série entre totalement dans cette catégorie, de même qu'une montre équipée d'un microcontrôleur pouvant communiquer par signaux infrarouges avec le monde extérieur. Dans le même ordre d'idée, la notion de coopération entre ordinateurs est vague et nécessite d'être précisée selon le domaine d'application considéré. Par exemple, tous les ordinateurs connectés à Internet peuvent échanger des informations deux à deux en utilisant la pile de protocoles TCP/IP et on peut donc dire qu'ils agissent de concert. Cependant, cela ne signifie pas qu'on puisse faire quoi que ce soit d'utile à l'aide de deux machines n'ayant rien d'autre en commun que ce protocole.

On peut noter que la définition adoptée exclut volontairement les architectures dites **fortement parallèles**, consistant en plusieurs microprocesseurs qui partagent une mémoire physique. La plupart des résultats généraux obtenus dans le cadre de cette thèse sont directement applicables à de tels systèmes, mais présenteront parfois des performances sous-optimales [Kesavan, 1998, Reed et al., 1991]. Les systèmes auxquels nous nous intéressons sont également appelés **systèmes parallèles faiblement couplés** [Balter et al., 1991]. Ces systèmes, dont la taille peut varier de deux machines (deux consoles de jeu interconnectées) à plusieurs dizaines de millions (l'ensemble des ordinateurs connectés à un moment donné sur Internet), communiquent par le biais de **messages**.

1.2 Terminologie

Dans la suite de ce document, nous nommerons **nœud** toute partie d'un système réparti. Un nœud peut être **physique** (une machine) ou **logique** (une partie d'un programme réparti, plusieurs parties pouvant se trouver sur le même nœud physique). Dans la suite de ce document, le terme "nœud" désignera, sauf spécification contraire, un nœud logique. Dans la terminologie Ada, chaque nœud logique est désigné sous le nom de **partition**.

Lorsqu'un nœud propose un service, ce nœud est appelé un **serveur**. L'utilisateur du service est appelé un **client**. Un système réparti basé uniquement sur des clients et un ou plusieurs serveurs est appelé **système client-serveur**.

On trouve dans la littérature aussi bien le terme **système réparti** (plus courant en français) que **système distribué** (plus proche de l'anglais "*distributed system*"). Pour des raisons d'uniformité, nous utiliserons le premier tout au long de ce document.

Les systèmes répartis que nous étudierons ici nécessitent l'utilisation, explicite ou implicite selon les cas, de modules dédiés à la répartition. Ces modules constituent l'**exécutif** ("*runtime*" en anglais) des systèmes répartis.

1.3 Utilisation des systèmes répartis

Les systèmes répartis sont actuellement déployés pour trois raisons principales : la mise à disposition pour un programme d'une puissance de calcul plus importante, la spécialisation des unités et le partage des ressources, ainsi que la tolérance aux pannes.

Les premiers cas représentent une **répartition par coopération**, dans laquelle les systèmes travaillent ensemble, et le dernier une **répartition par réplication**, dans laquelle les systèmes sont "en concurrence" les uns avec les autres, puisqu'ils effectuent la même tâche.

1.3.1 Performance

La mise en commun de plusieurs ordinateurs permet de réduire le temps nécessaire pour effectuer certains types de calculs en parallélisant des opérations travaillant sur des données indépendantes. Cela permet d'augmenter les ressources utilisées à un moment donné afin de terminer plus rapidement le calcul.

Un compromis doit toutefois être recherché entre le nombre de nœuds du réseau et la complexité des calculs qu'ils ont à traiter. En effet, les temps de communication peuvent devenir supérieurs au gain obtenu en termes de temps de calcul, provoquant une dégradation des performances au lieu d'une amélioration. Il faut donc que la granularité (c'est-à-dire la taille des travaux individuels) de la répartition soit suffisante pour que le rapport entre le temps de calcul et le temps de communication reste élevé.

Les deux exemples suivants présentent des situations dans lesquelles il est courant d'utiliser la répartition pour des raisons de performance. Dans le premier exemple, on souhaite terminer un travail donné en un temps le plus court possible, alors que dans le second on souhaite explorer un plus grand nombre de possibilités. Bien entendu, ces deux cas ne prétendent aucunement refléter toutes les utilisations de la coopération dans les systèmes répartis.

Calcul d'image par suivi de rayons

Une des techniques permettant de modéliser l'image perçue par un utilisateur à partir de la description des objets composant une scène en trois dimensions consiste à utiliser la propriété physique appelée "trajet inverse de la lumière" : plutôt que de suivre le trajet effectué par la lumière provenant des différentes sources d'éclairage, on lance des rayons depuis l'œil de l'utilisateur dans toutes les directions observables et on détermine si ces rayons atteignent ou non une source de lumière. On applique bien entendu toutes les transformations relatives aux changements de couleur, de diffraction et de réfraction introduites par les matériaux composant les objets de la scène.

Cette méthode de calcul est facilement parallélisable : si tous les nœuds d'un système réparti possèdent une copie de la description de la scène et des objets qui la composent, il leur est possible de calculer une portion de l'image de manière totalement décorrélée du résultat obtenu par les autres nœuds. On obtient donc une diminution du temps nécessaire pour construire une image complète.

Attaque brutale d'algorithmes cryptographiques

Un algorithme de chiffrement et l'algorithme de déchiffrement associé nécessitent la connaissance d'une clé pour pouvoir coder et décoder les données chiffrées. Les algorithmes actuels sont pour la plupart basés sur le fait que la complexité du calcul nécessaire pour deviner la clé de déchiffrement est équivalente à la complexité du calcul nécessaire pour décomposer un très grand nombre en facteurs premiers, complexité considérée aujourd'hui comme rédhibitoire. La sécurité de ces algorithmes est également assurée par le fait que l'exploration exhaustive de l'espace de clés prendrait plusieurs années si l'on considère la puissance des processeurs actuels.

C'est pourquoi un projet a vu le jour sur Internet en janvier 1997 ayant pour but d'utiliser les cycles disponibles des machines connectées au réseau [Litzkow et al., 1988] avec l'espoir de parvenir à essayer la totalité des clés sur un message de test. Le but est de démontrer aux gouvernements régulant les moyens cryptographiques que les tailles des clés autorisées à ce jour sont insuffisantes pour assurer une réelle protection des données sensibles.

Cette mise en commun de ressources informatiques constitue un système réparti largement déployé, dont le nombre de nœuds ne cesse d'augmenter avec la popularité du projet. En avril 1999, le nombre de clés de 64 bits testées par seconde pour essayer de décoder un message chiffré avec l'algorithme RC5 est de 70,9 milliards, alors qu'il n'était que de 100 millions deux ans auparavant.

Dans cet exemple précis, le coût de la communication est faible. Les données échangées se limitent au message à déchiffrer et à deux bornes délimitant l'espace de clés à utiliser pour la recherche.

1.3.2 Spécialisation des unités et partage de ressources

À l'opposé de la situation précédente où le système réparti était utilisé pour mettre en commun des ressources similaires (du temps de calcul), il est des cas où l'on souhaite faire collaborer des systèmes aux fonctions très différentes. Tous les nœuds ne sont plus ici équivalents au regard du système.

La figure 1.1 page ci-contre présente un cas fréquemment observé en entreprise : une imprimante, destinée à être utilisée depuis plusieurs machines, ne peut pas être connectée directement au réseau local car elle ne dispose que d'une interface de communication série.

Un serveur d'impression assure alors le lien entre le réseau local, sur lequel l'ensemble des ordinateurs ayant à utiliser l'imprimante sont connectés, et l'imprimante elle-même. L'ensemble composé des ordinateurs, du serveur d'impression et de l'imprimante forme un système réparti faiblement couplé.

On peut également noter que la solution consistant à acheter une imprimante par ordinateur aurait également permis d'imprimer à partir de n'importe quelle machine. On voit ici que les systèmes répartis peuvent, dans certains cas, réduire les coûts d'investissement en offrant de nouvelles possibilités de partage.

1.3.3 Redondance et tolérance aux pannes

Les systèmes répartis sont aussi fréquemment utilisés à des fins de redondance de services et de données. Cette redondance permet soit d'offrir le même service depuis des points d'accès distincts, soit d'offrir un service de manière ininterrompue en particulier en cas de panne.

Une panne dans un système réparti peut être due à l'indisponibilité d'un ou plu-

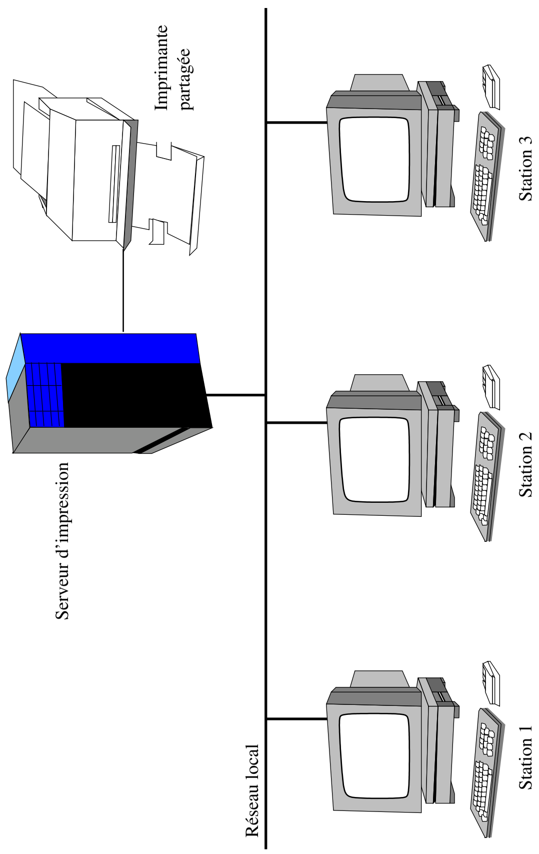


FIG. 1.1 – Serveur d'impression spécialisé

sieurs nœuds, à l'indisponibilité du réseau ou encore à une faute byzantine, c'est-à-dire une situation dans laquelle un des nœuds fonctionne incorrectement et renvoie, volontairement ou non, des résultats erronés. On pourra trouver de nombreux exemples illustrant ces pannes dans [Hadzilacos et Toueg, 1993], [Miranda et al., 1996] et [Wolf et Strohmeier, 1999]. Nous allons toutefois présenter brièvement deux situations de pannes, une panne par indisponibilité puis une panne par mensonge délibéré d'un des nœuds de l'application répartie. Dans ce cas également, nous ne prétendons pas être exhaustif.

Notons que la redondance au niveau du logiciel complète idéalement la redondance au niveau du matériel que l'on peut trouver dans les systèmes dans lesquels une défaillance peut entraîner des blessures ou le décès de personnes. Dans les avions par exemple, tous les systèmes matériels sont dupliqués, ainsi que les systèmes logiciels correspondants.

Reconfiguration automatique

Lorsque deux ordinateurs communiquent par le biais d'Internet, chaque paquet transmis l'est selon un chemin indépendant de celui des autres paquets. Les routeurs, qui sont les nœuds par lesquels transitent les paquets, se contentent d'envoyer chaque paquet vers un autre routeur plus proche de la destination finale. De plus, les routeurs communiquent entre eux afin de s'échanger les informations de routage de proche en proche. Chaque routeur sait donc à qui il peut envoyer un paquet en fonction de sa destination : le routeur destinataire sera obligatoirement plus proche du destinataire que le routeur courant.

Lorsqu'un routeur tombe en panne, ses voisins s'aperçoivent que ce routeur n'est pas utilisable et enlèvent les entrées correspondant à ce nœud dans leur table de routage. Les nouveaux paquets emprunteront donc une route ne passant pas par le routeur en panne, assurant ainsi un service permanent. Nous avons là un exemple d'un système réparti offrant une tolérance aux pannes par le biais de la redondance (s'il n'existait qu'un chemin entre les deux machines, la panne d'un routeur sur le trajet entraînerait irrémédiablement la rupture de la communication) et de la reconfiguration dynamique. La figure 1.2 page suivante montre comment un chemin plus long peut être choisi en cas de panne d'un des routeurs sur le trajet le plus court.

Protection contre les actes de malveillances (fautes byzantines)

Le projet SETI@HOME a pour ambition de repérer des régularités dans le bruit électromagnétique venant de l'espace qui correspondraient à des traces de communication extraterrestres. De nombreux enregistrements ont été produits et sont analysés par des ordinateurs connectés à Internet, comme dans le cas du projet RC5 mentionné plus haut.

Afin de se protéger contre des actes de malveillance, le serveur qui distribue les échantillons à traiter envoie chaque jeu de données à au moins quatre clients distincts. Si un client renvoie un résultat différent de celui des trois autres, il est considéré comme fautif et son résultat est ignoré [Dolev, 1982]. Cette technique permet de bénéficier de la puissance de calcul d'ordinateurs individuels dans lesquels on ne peut placer aucune confiance [Davis, 1998].

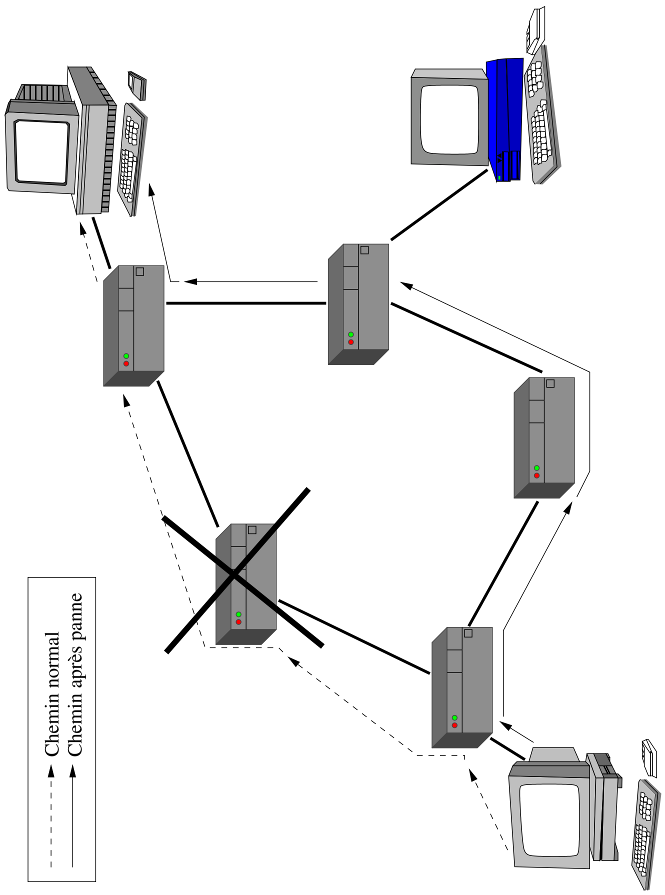


FIG. 1.2 – Redondance et reconfiguration dynamique

1.4 Communications dans les systèmes répartis

La définition choisie dans la section 1.1 ne présume en rien des mécanismes utilisés par les ordinateurs formant un système réparti pour dialoguer entre eux. Nous présentons ici les différentes méthodes utilisées pour faire coopérer des machines différentes, tout en évoquant les problèmes inhérents à un telle coopération.

1.4.1 Systèmes hétérogènes

Un des problèmes les plus anciens dans le domaine des systèmes répartis est dû aux différences d'architecture des différents nœuds physiques d'un système. Par exemple, la représentation en mémoire d'un nombre entier ou en virgule flottante diffère d'un microprocesseur à un autre. De même, la place occupée en mémoire par un nombre entier peut varier de 8 à 128 bits selon la machine considérée, et cette taille ne cesse de croître sur les systèmes modernes.

Afin de rendre interopérables des ordinateurs basés sur des microprocesseurs d'origines différentes, des conventions à base de dictionnaires ont été définies. Par exemple, dans une application répartie, on conviendra que tous les entiers échangés le sont sur 4 octets, le premier octet représentant l'octet de poids fort. Certains formats de dictionnaires ont été largement adoptés par la communauté informatique, comme XDR[1], utilisé notamment pour le protocole de transfert de fichiers NFS[2] [Corbin, 1993], ou CDR[3], utilisé dans les systèmes basés sur le modèle CORBA [OMG, 1998].

Avant de transmettre une donnée à un autre nœud de l'application répartie, l'émetteur la dirige vers une routine d'**emballage** qui normalise son format en fonction de son type. Lorsqu'il reçoit du réseau une donnée dont il connaît le type, le récepteur utilise la routine de **déballage** afin de la récupérer dans son format natif.

1.4.2 Envoi de messages

Les systèmes utilisant les envois de messages forment la classe de systèmes répartis la plus primitive. Chaque nœud peut envoyer de manière asynchrone des messages à un autre nœud, qui doit se tenir prêt à les traiter.

Dans ce type de systèmes, l'utilisateur doit en général appeler explicitement les routines d'emballage et de déballage des paramètres ainsi que les primitives de communication, c'est-à-dire les sous-programmes permettant d'envoyer et de recevoir des paquets. Des systèmes comme PVM[4] [Geist et al., 1994] et MPI[5] [MPI, 1995] [MPI, 1997] normalisent à la fois l'interface, qui fournit les routines d'emballage et de déballage et les points d'entrée du système de communication, et le format des paquets, permettant de faire interopérer deux implémentations fournies par des vendeurs différents.

D'autres systèmes, comme celui incorporé dans le langage de programmation faiblement typé Erlang, gèrent automatiquement l'emballage et le déballage des paramètres en fonction de leur type dynamique [Armstrong et al., 1998]. L'émetteur d'un message

[1] eXternal Data Representation
[2] Network File System
[3] Common Data Representation
[4] Parallel Virtual Machine
[5] Message-Passing Interface

le dépose dans la boîte à lettres du destinataire, qui viendra ensuite le récupérer lorsqu'il sera prêt à traiter un message entrant. Des mécanismes incorporés au langage permettent au destinataire de reconstruire automatiquement les structures de données telles qu'elles existaient chez l'émetteur.

Ce mécanisme n'offre pas de synchronisation forte entre l'appelant et l'appelé, au sens où si le message original nécessite une réponse, cette réponse devra être envoyée sous la forme d'un nouveau message indépendant de la requête. Le système de multi-fenêtrage X-Window, par exemple, est basé sur ces concepts : chaque application peut envoyer à tout moment des messages au programme de multi-fenêtrage (à l'occasion du changement de contenu d'une fenêtre par exemple), qui peut lui-même envoyer des messages aux applications (comme l'ordre de se redessiner) [Nye, 1992].

Exemple d'application de l'envoi de messages

L'envoi de messages est un principe couramment utilisé pour construire des anneaux virtuels. Comme la figure 1.3 page suivante permet de le voir, un anneau virtuel sur lequel se trouvent quatre nœuds (représentés par des ordinateurs bien que ce soient des nœuds logiques) est en fait basé sur des envois de messages de proche en proche entre les différents nœuds. Un message envoyé sur l'anneau depuis le nœud A est en fait transmis au nœud B, qui le transmet lui-même au nœud C, avant d'être finalement délivré au nœud D.

Communication de groupe

Des systèmes comme PVM autorisent l'envoi de messages d'un nœud vers plusieurs autres. Ce mécanisme, appelé **communication de groupe**, permet d'envoyer la même information à tout un groupe de nœuds. Cela est utile lorsque plusieurs nœuds offrent le même service, afin de garantir une certaine tolérance aux défaillances.

1.4.3 Appel de procédure à distance

La synchronisation explicite nécessaire pour développer des applications en utilisant l'envoi de messages nécessite la prise en compte de cette contrainte par l'architecte de l'application répartie dès la phase de conception. Une alternative à ce mécanisme beaucoup plus naturelle pour un développeur consiste à utiliser les appels de procédures à distance[6], ou RPC (de l'anglais "*Remote Procedure Call*"). Chaque appel vers un nœud distant se traduit par un appel de sous-programme ; les paramètres de retour de ce sous-programme correspondent, du point de vue du développeur, aux résultats renvoyés par l'ordinateur distant par l'intermédiaire du canal de communication. La requête et la réponse sont appariées automatiquement par l'exécutif, ajoutant ainsi une synchronisation implicite et une cohérence absentes du modèle précédent.

Les procédures appelables à distance sont définies par une description d'interface, qui précise le contrat entre l'appelant et l'appelé, notamment le nombre et le type des arguments. À partir de cette interface, une application dédiée génère automatiquement

[6]Le terme de "procédure" employé ici est un abus de langage, et devrait être "sous-programme", des fonctions pouvant également être invoquées à distance. Nous l'emploierons toutefois par souci d'homogénéité avec la littérature existante.

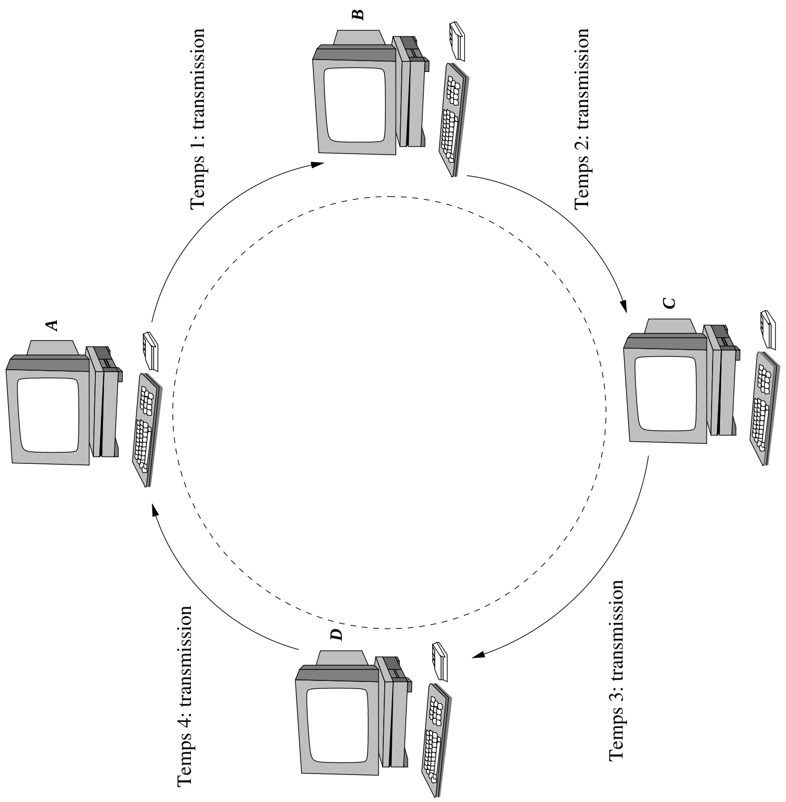


FIG. 1.3 – Anneau à passage de messages

une ou plusieurs **souches**, qui représente la procédure telle que perçue par l'appelant, et un ou plusieurs **squelettes**, étendus par l'appelé.

La souche présente exactement la même interface que le sous-programme appelé à distance, ce qui signifie que l'appel côté client est écrit comme si le sous-programme en question était un sous-programme local. Cette souche construit une requête à partir des arguments et envoie cette requête au serveur.

Côté serveur, le squelette se charge de récupérer la requête entrante sur le canal de communication. Il appelle ensuite le sous-programme désigné après avoir extrait ses arguments et renvoie les résultats au client via la souche [Birrell et Nelson, 1984].

La figure 1.4 page suivante, inspirée de [Tanenbaum, 1992], présente une vue globale de l'ensemble du processus.

1.4.4 Extensions de l'appel de procédure à distance

En plus de la possibilité d'échanger des paramètres d'une manière comparable à un appel de sous-programme dans un langage procédural classique, les appels de procédures à distance permettent, dans certains cas, de refléter certaines caractéristiques de plus haut niveau présentes dans le langage utilisé. Nous présentons ici les extensions les plus intéressantes, en fournissant lorsque cela est possible un pointeur sur la section du manuel de référence Ada où ces extensions sont définies dans le cadre de ce langage.

L'extension la plus importante à notre avis concerne les exceptions : si une exception est levée dans une procédure s'exécutant à distance, alors il est souhaitable de propager cette exception jusqu'à l'appelant, en lui permettant de la rattraper à l'aide d'un traite-exceptions [RM, 1995, E.4(15)]. Notons que dans ce cas, l'exception est considérée comme un résultat possible d'un appel de sous-programme (voir section 7.1.1 page 107). La présence de cette extension permet de rapprocher les modèles répartis et monolithiques, en unifiant les deux modèles de flots de contrôle.

Il est également possible de définir, à partir des appels de procédure à distance, un service similaire à l'envoi de messages, en spécifiant qu'une procédure est **unidirectionnelle** [RM, 1995, E.4(1)], *i.e.* qu'elle ne renvoie aucun résultat. Dans ce cas, l'appelant peut continuer sa propre exécution avant la fin de l'exécution de la procédure distante.

Une autre extension intéressante est la possibilité de rendre transportables des types qui ne le sont pas *a priori*. Par exemple, la possibilité de transformer un type pointeur, dont la sémantique est uniquement locale, en un type pointeur permettant de désigner une entité distante permet d'étendre les possibilités offertes par des appels de sous-programmes distants.

On peut également définir des pointeur sur sous-programmes distants [RM, 1995, E.2.2(9)], permettant de différer la connaissance du destinataire d'un appel distant jusqu'au moment de l'exécution du programme réparti. Cela autorise la création de systèmes dynamiques, permettant la reconfiguration d'une application par le biais du reroutage : en changeant le sous-programme distant désigné par une instance de ce type de pointeur, il est possible de changer en cours d'exécution la cible d'un groupe d'appels distants.

Une autre extension possible de ce modèle d'appel de sous-programme à distance est le contrôle de version automatique. Cela consiste à vérifier la cohérence de l'interface

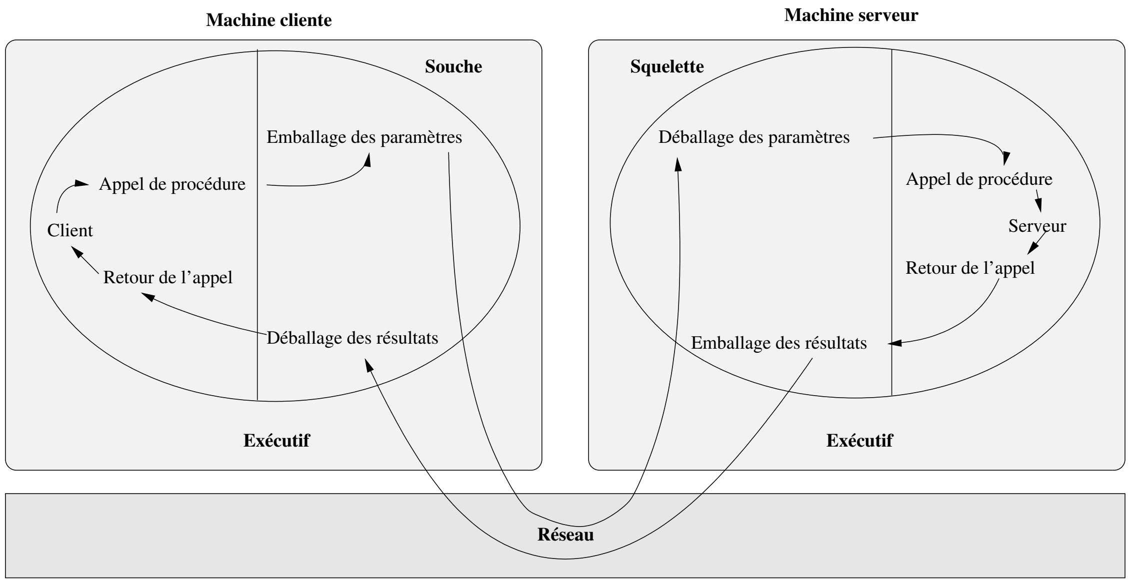


FIG. 1.4 – Appel de sous-programme à distance

utilisée par l'appelant et l'appelé [RM, 1995, E.3(6)], et à signaler une erreur lorsque les deux entités utilisent des versions incompatibles de l'interface.

1.4.5 Envoi de messages *vs.* appels de procédures à distance

Le schéma de la figure 1.5 résume les apports respectifs des deux approches. Il est beaucoup plus aisé pour un développeur ne maîtrisant pas la programmation des couches de communication d'utiliser les appels de sous-programmes à distance plutôt que le passage de messages. De plus, les risques d'erreurs de programmation sont réduits par l'appel automatique des routines d'emballage et de déballage.

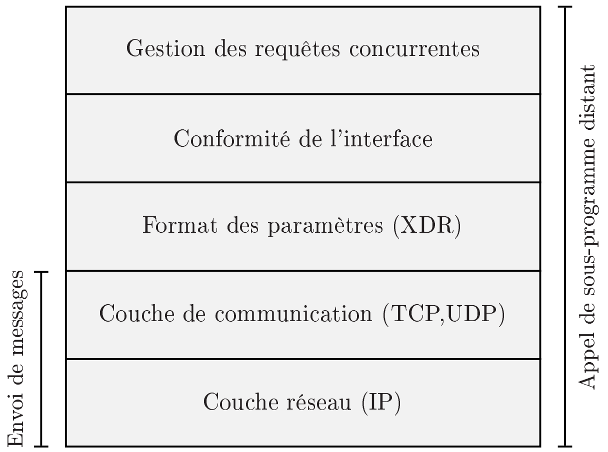


FIG. 1.5 – Fonctions offertes par chaque méthode

L'utilisation d'appels de sous-programmes distants permet de vérifier la cohérence des interfaces utilisées entre l'appelant et l'appelé et de faire prendre en charge par l'exécutif la gestion des requêtes concurrentes. Cela permet de construire beaucoup plus facilement des applications robustes, moins sujettes aux fautes du développeur.

D'un autre côté, l'envoi de messages permet de limiter le trafic réseau au strict minimum. Aucun surcoût n'est caché, au prix d'une plus grande difficulté de conception.

1.5 Objets répartis

La programmation orientée objets a fourni un moyen simple d'organiser une architecture autour des acteurs d'un système plutôt qu'autour de ses fonctionnalités, comme c'était le cas auparavant [Dahl et Nygaard, 1966]. Les objets répartis apportent les mêmes avantages aux systèmes répartis, en offrant une souplesse absente des appels de procédures à distance simples, comme une grande dynamicité. Tous les bénéfices de la programmation orientée objets se retrouvent dans ces architectures réparties. Il est par exemple aisé de faire évoluer une application progressivement en utilisant la programmation par extension, l'héritage et la surcharge [Meyer, 1988].

1.5.1 Concepts et définitions

Les objets répartis représentent une évolution des appels de sous-programmes à distance, en permettant d'invoquer des méthodes d'objets distants sans se préoccuper de leur localisation. Il faut noter que les objets répartis ne se déplacent pas physiquement de nœud en nœud : seules les références sur ces objets sont échangées entre les différentes composantes du système réparti, les invocations de méthodes sur des objets distants par le biais de ces références générant, de manière transparente, un appel de méthode à distance.

Un objet réparti est conceptuellement représenté par un **pointeur long**, une entité qui contient à la fois les coordonnées du nœud sur lequel l'objet est physiquement situé et une adresse dans la mémoire locale de ce nœud. Ces deux informations, une fois combinée, permettent d'identifier l'objet de manière non ambiguë (voir figure 1.6). Une URL[7], par exemple, représente un objet distant de manière univoque : `http://www.example.com/object1` contient une partie permettant d'identifier le nœud sur lequel se trouve la ressource (`http://www.example.com`) et une partie permettant d'identifier la ressource sur ce nœud (`/object1`).

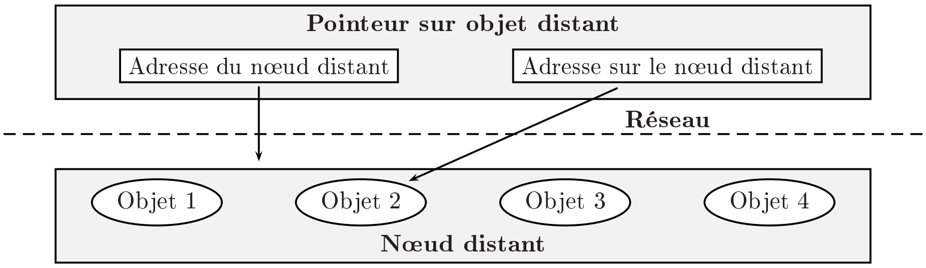


FIG. 1.6 – Représentation théorique d'un pointeur sur objet distant

1.5.2 Exemples de systèmes basés sur les objets répartis

Nous présentons ici les deux systèmes d'objets répartis les plus fréquemment utilisés. Le lecteur intéressé trouvera une comparaison exhaustive des principaux systèmes dans [Orfali et al., 1996]. Le chapitre 13 page 171 de ce document contient également une comparaison de CORBA, décrit brièvement ci-dessous, et de l'annexe des systèmes répartis.

Les deux systèmes présentés ici, CORBA et RMI, sont fondés uniquement sur les objets répartis. Ils offrent la possibilité d'invoquer des méthodes sur des objets distants, mais ne permettent pas d'effectuer des appels de simples sous-programmes distants, c'est-à-dire des sous-programmes non liés à des classes et instances d'objets particulières.

[7]Uniform Resource Locator

CORBA

CORBA[8] [OMG, 1998] est actuellement le plus populaire des systèmes utilisant les objets répartis. Ce succès est dû à la ressemblance syntaxique de son langage d'interface IDL[9] avec C++, ainsi qu'à la participation active de certaines sociétés majeures comme SUN MICROSYSTEMS dans sa définition et sa diffusion. Le développement de la norme a été accompagné d'implémentations de référence, permettant d'évaluer les nouveaux concepts au fur et à mesure de leur introduction dans le standard.

Les différentes parties du programme réparti sont reliées par l'intermédiaire d'un ORB[10], une entité logicielle permettant de gérer les références d'objets répartis. Les différents ORB communiquent *via* Internet en utilisant le modèle de bus logiciel sur lequel circulent des paquets codés à l'aide d'un protocole normalisé appelé IIOP[11]. Un schéma global est donné dans la figure 1.7 page suivante.

Une des originalités de CORBA est qu'il peut, à partir de la description IDL d'une interface, générer des souches et des squelettes dans de nombreux langages cibles [Orfali et Harkey, 1997], tels que Ada, C, C++, Java ou Lisp. Par exemple, un programme écrit en Ada pourra accéder sans effort à un service écrit en Java. Les traductions d'IDL vers un langage de programmation sont normalisées ; cela signifie qu'il est en théorie facile de changer de logiciel CORBA sans changer le code utilisateur. En pratique, on constate que les vendeurs d'implémentations CORBA font souvent appels à des constructions propriétaires en profitant de certaines lacunes de la norme, notamment du côté serveur.

CORBA offre également la possibilité d'invoquer dynamiquement une méthode sur un objet inconnu : il est possible, en utilisant un dépôt d'interfaces ("*Interface Repository*" en anglais), d'obtenir la signature d'une méthode référencée par son nom, et de construire ensuite la requête "à la main", argument par argument. Cela permet d'introduire de nouveaux types d'objets en cours d'exécution, sans interrompre le programme réparti, par le biais de ce mécanisme souple mais dont l'utilisation s'avère toutefois plus complexe que des appels de méthodes traditionnels.

Toutes les fonctionnalités importantes pour un système réparti, comme le service de nommage, sont implémentées dans CORBA comme des services externes. Un grand nombre de services sont normalisés, que ce soient des services à usage général (nommage, sécurité, persistance), ou des services liés à un domaine d'activité précis (bases de données, services médicaux, calculs mathématiques). Cela permet à un architecte d'acquérir auprès d'un vendeur un service permettant par exemple l'analyse d'une image provenant d'un scanner, puis de changer de vendeur si les performances du premier ne le satisfont pas tout en ne changeant rien à son code existant.

RMI

RMI[12] est un système d'objets répartis limité au langage de programmation Java [Javasoft Java RMI Team, 1997]. Java étant est un langage purement objet, il était naturel d'adopter ce paradigme comme base des systèmes répartis écrits dans ce lan-

[8] Common Object Request Broker Architecture
[9] Interface Description Language
[10] Object Request Broker
[11] Internet Inter-ORB Protocol
[12] Remote Method Invocation

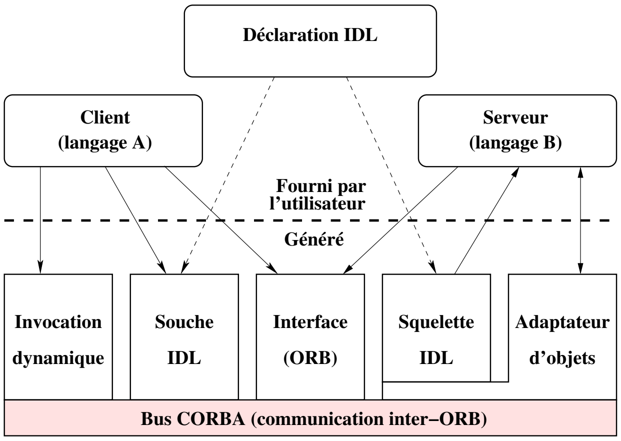


FIG. 1.7 – Schéma global d'un programme réparti CORBA

gage. Cependant, le succès actuel de CORBA limite les probabilités de survie à long terme de RMI.

De par sa conception, un code source Java est en général compilé pour une **machine virtuelle** [Lindholm et Yellin, 1996], c'est-à-dire une machine n'existant pas réellement mais émulée sur un autre système. Le code exécutable est alors portable d'une plate-forme physique à une autre sans recompilation ni modification. C'est pourquoi, en plus des possibilités traditionnelles offertes par les objets répartis, RMI permet de transporter certains objets ainsi que la classe les définissant (c'est-à-dire leur code) entre les différents nœuds composant un système. Il est donc possible de rapatrier le code correspondant à une classe sur un nœud et de l'exécuter localement plutôt que d'utiliser des appels de méthodes distantes.

Ce concept est appelé **migration de code** et n'est applicable sur des architectures hétérogènes que dans le cadre de langages purement interprétés ou basés sur des machines virtuelles. Cependant, de plus en plus de langages permettant la programmation orientée objet, comme par exemple Ada [Comar et al., 1997, Millet et Baker, 1998] et Python [Van Rossum, 1998], peuvent être compilés pour la machine virtuelle conçue initialement pour Java, et offrent donc de nouvelles possibilités d'interaction entre langages et de migration de code.

2

Ada et les systèmes répartis

2.1 Le langage Ada

2.1.1 La première version

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. L'équipe du français Jean Ichbiah, de CII-HONEYWELL-BULL, a remporté la phase finale de sélection. 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, 1983]. Elle est devenue une norme internationale grâce à son adoption par l'ISO[1] en 1987 [RM, 1987].

Le langage Ada a comme propriétés principales :

Typage fort : Ada est un langage fortement typé, c'est-à-dire qu'il est possible de définir plusieurs types ayant la même représentation (par exemple un type entier allant de 1 à 5) incompatibles entre eux (on ne peut pas utiliser un de ces types à la place d'un autre sans conversion explicite). Cela permet de détecter dès la compilation une erreur courante en programmation, à savoir l'utilisation d'une variable à la place d'une autre. Si les types des deux variables diffèrent, le compilateur refusera de compiler le code source.

Abstraction : 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 l'implémentation des services du paquetage, dont les détails ne seront jamais propagé à l'extérieur du corps lui-même. L'exemple 2.1 page suivante illustre ces concepts.

Concurrence : il est possible d'écrire en Ada des programmes comportant plusieurs tâches, une tâche étant un flot de contrôle s'exécutant indépendamment des

[1] International Standards Organization

Exemple 2.1 Abstraction et encapsulation en Ada

```
package Paquetage_Compteur is

   --  Cette partie est visible de l'extérieur

   type Compteur is private;
   --  Une unité extérieure utilisant ce paquetage
   --  ne "voit" pas la manière dont le type Compteur
   --  est implémenté, mais peut le manipuler.

   procedure Initialisation (C : in out Compteur);
   --  Initialise un nouveau compteur

   procedure Incremente (C : in out Compteur);
   --  Incrémente un compteur

   function Representation (C : Compteur) return String;
   --  Retourne une représentation affichable du compteur

private

   --  Cette partie contient les détails d'implémentation

   type Compteur is new Natural;
   --  Le type Compteur est un entier positif ou nul

end Paquetage_Compteur;

package body Paquetage_Compteur is

   --  Cette partie contient l'implémentation des
   --  sous-programmes.

   procedure Initialisation (C : in out Compteur) is
   begin
      C := 0;
   end Initialisation;

   procedure Incremente (C : in out Compteur) is
   begin
      C := C + 1;
   end Incremente;

   function Representation (C : Compteur) return String is
   begin
      return Compteur'Image (C);
   end Représentation;

end Paquetage_Compteur;
```

autres sauf en cas de synchronisation explicite. Chaque tâche possède sa propre pile et ses variables locales, mais peut accéder à l'ensemble de la mémoire de l'application.

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. L'exemple 2.2 page suivante présente des tâches utilisant des rendez-vous.

Facilité de lecture : contrairement à d'autres langages de programmation, l'utilisation d'Ada facilite la tâche de l'utilisateur qui doit relire du code existant plutôt que celle du concepteur qui écrit le code initial. Si Ada est parfois perçu comme un langage prolixe, à cause de ses exigences fortes en terme de syntaxique, une plus grande facilité de maintenance est attestée par les utilisateurs [Jones, 1998].

Bibliothèques standardisées : un ensemble de bibliothèques normalisées dans le manuel de référence permettent de disposer de certaines fonctionnalités courantes sans avoir besoin de réécrire les paquetages nécessaires (manipulations de chaînes de caractères et entrées/sorties par exemple).

Contrôle de l'élaboration : les différents paquetages formant une application Ada doivent être **élaborés**, c'est-à-dire que leurs déclarations sont évaluées avant de lancer le programme principal afin de positionner les variables à leur valeur initiale et de lancer les tâches de plus haut niveau. Ada fait en sorte que lorsqu'un paquetage en utilise un autre, l'élaboration du second ait lieu avant celle du premier.

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

Ces caractéristiques font d'Ada un langage puissant adapté à un grand nombre de situations. Il satisfait aux exigences du cahier des charges initial demandant à ce qu'il puisse être utilisé dans tous les domaines de la défense ou de l'industrie.

Le ministère de la défense américain a obligé, par le biais d'une directive, tous ses programmeurs et ses contractants à utiliser Ada pour les nouveaux développements et les mises à jour importantes de logiciels existants. Les dérogations à cette règles nécessitaient des justifications tellement importantes qu'Ada est devenu à cette époque l'unique langage utilisé dans le monde de la défense américain.

2.1.2 Une nouvelle norme

En 1995, après un long processus de révision commencé en 1988, une nouvelle norme ISO correspondant au langage Ada a été adoptée [RM, 1995], et introduit un grand nombre de nouveaux concepts [Intermetrics, 1995]. Parmi ceux-ci, on trouve :

Programmation orientée objet : Ada supporte le polymorphisme et l'aiguillage 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 c'est une unité de plus haut niveau) il est possible d'étendre un paquetage par la création d'un paquetage fils, qui obtient la visibilité sur la partie privée de son père.

Exemple 2.2 Programme Ada comprenant des tâches

```
with Text_IO; use Text_IO;

procedure Proc_Avec_Tache is

   task Affiche is
      entry Affiche_Cela (S : in String);
   end Affiche;
   --  Cette tâche affiche ce qu'on lui donne en
   --  paramètre de l'entrée Affiche_Cela.

   task body Affiche is
   begin
      --  Accepte les requêtes indéfiniment
      loop
         select
            --  Accepte un rendez-vous sur l'entrée Affiche_Cela ...
            accept Affiche_Cela (S : in String) do
               Put_Line (S);
            end Affiche_Cela;
         or
            --  ... ou disparaît s'il ne reste rien à faire
            terminate;
         end select;
      end loop;
   end Affiche;

   task Affiche_Bonjour;
   --  Cette tâche affichera Bonjour! toutes
   --  les deux secondes et s'arrêtera au bout de
   --  10 itérations.

   task body Affiche_Bonjour is
   begin
      for I in 1 .. 10 loop
         --  Provoque un rendez-vous avec l'autre tâche
         Affiche.Affiche_Cela ("Bonjour");
         delay 2.0;
      end loop;
   end Affiche_Bonjour;

begin
   Put_Line ("Les tâches sont lancées");
end Proc_Avec_Tache;
```

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.

Compatibilité ascendante : si un code source Ada 83 valide peut être compilé à l'aide d'un compilateur Ada 95, alors l'effet de la compilation sera le même qu'avec un compilateur Ada 83. Cela respecte le principe de "moindre surprise", cohérent avec l'esprit du langage.

Objets protégés : les possibilités de synchronisation ont été enrichies grâce à l'introduction des objets protégés. Ces objets permettent de sérialiser les modifications apportées à des données, tout en protégeant leur consultation afin de la rendre exclusive vis-à-vis des modifications. De plus, des entrées protégées permettent de ne pénétrer dans la section critique d'un objet que lorsque certaines conditions sont vérifiées.

Annexes optionnelles : ces annexes, qui ne sont pas obligatoirement implémentées dans tous les compilateurs Ada validés, doivent, si elles le sont, respecter la norme. Des annexes existent entre autres pour le temps-réel, la sécurité et la répartition.

2.2 Sémantique de l'annexe des systèmes répartis

L'annexe optionnelle qui nous intéresse plus particulièrement est l'annexe E, appelée "**annexe des systèmes répartis**" ("*Distributed Systems Annex*" dans le manuel de référence). Cette annexe définit un moyen de développer des applications réparties sans sortir du cadre du langage ; 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.

La suite de cette section, après avoir présenté les idées principales ayant guidé la définition de l'annexe des systèmes répartis, décrit les différentes entités utilisées par sa mise en œuvre. Mais tout d'abord, nous définissons quelques termes qui seront utilisés par la suite.

2.2.1 Préambule

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

Une **directive de compilation** (ou "*pragma*" en anglais) est une construction syntaxique indiquant certaines propriétés au compilateur sans effet sémantique. La présence de la directive de compilation `Inline` appliquée à un sous-programme par exemple 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. Son rôle est de définir un gabarit : les **types concrets** dérivés d'un type abstrait devront obligatoirement redéfinir toutes les méthodes déclarées comme abstraites du type de base. Un type ou une méthode sont identifiés par la présence du mot clé `abstract`.

Un **attribut** est une construction syntaxique et sémantique permettant d'accéder à une propriété spécifique d'un objet donné. La syntaxe en Ada est Objet'Attribut. Par exemple, Integer'Image dénote une procédure permettant de transformer un entier en chaîne de caractères, tandis que Racine'Class est un type représentant l'ensemble de la hiérarchie des types dont un ancêtre (ou le type lui-même) est Racine.

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 monolithiques et les programmes répartis. Ainsi, il est trivial de passer d'une application non répartie à une application répartie et inversement par simple changement d'outil : les mêmes sources conviendront aux deux modèles.

C'est pourquoi la norme Ada n'introduit aucun nouveau mot-clé spécifique à la répartition ; seules quelques nouvelles **directives de compilation** permettent de décrire les propriétés 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 différentes partitions (nœuds logiques) sont transportables sans perte de signification (les types transportables sont définis dans la section 2.2.6 page 49). 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 une autre partition.

Ada distingue deux types de partitions :

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.

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

Tous les appels entre partitions actives se font par le biais d'appels de sous-programmes ou de méthodes d'objets répartis. Ada n'offre pas la possibilité d'utiliser 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.

2.2.3 Les types de flux

Ada 95 définit un nouveau type de flux abstrait Ada.Streams.Root_Stream_Type un flux étant un objet dans lequel on peut écrire des données et relire ces mêmes données par la suite, à l'image d'un tampon. Entre-temps, le flux a pu être transporté d'une partition à une autre ou encore sauvegardé dans un fichier, fournissant ainsi un moyen de conserver des données entre deux exécutions d'un programme.

Le type Root_Stream_Type étant un type abstrait, seules de nouvelles dérivations concrètes peuvent être utilisées. Par exemple, le paquetage Ada.Streams.Stream_IO définit un type de flux concret destiné à être écrit dans un fichier. L'écriture et la lecture d'un type contraint[2] se font respectivement à l'aide des attributs Read et Write de ce type. L'exemple 2.3 page 46 illustre l'utilisation de l'attribut Read. On suppose

[2] C'est-à-dire un type dont la taille et les discriminants sont connus à la compilation.

que le fichier à partir duquel est construit le flux contient successivement un entier puis un flottant. L'utilisation directes des flux nécessite de connaître précisément le type des données qui y ont été stockées (nous verrons plus tard que, dans le cadre d'appels de sous-programme distants, l'usage de ces flux est masqué par le compilateur).

Pour les types contraints ou non contraints, deux autres attributs, appelés Input et Output, permettent respectivement d'écrire et de lire dans un flux tous les éléments nécessaires à la reconstruction de l'objet manipulé. Par exemple, l'utilisation de l'attribut Output sur un type tableau écrira les bornes avant d'écrire le contenu du tableau. On notera qu'à la différence de l'attribut Read qui dénote une procédure, l'attribut Input désigne une fonction ; ceci est nécessaire dans le cas de l'extraction d'une valeur d'un type non contraint. L'exemple 2.4 page suivante illustre l'utilisation de l'attribut Output.

Il est possible de sauvegarder un objet polymorphe sans savoir quel est son type exact en utilisant l'attribut Output sur le type spécial couvrant l'ensemble de la classe, Racine'Class ; cela donne des expressions enchaînées comme Racine'Class'Output (S, X). Lors de la lecture par le biais de l'attribut Input d'un objet de type indéterminé, l'exception Ada.Tags.Tag_Error est levée lorsque le type de l'objet précédemment stocké est inconnu. Cela peut arriver lorsqu'un programme écrit un objet appartenant à une hiérarchie dans un flux stocké dans un fichier, et qu'un autre programme, connaissant la racine de la hiérarchie mais pas cette dérivation particulière, tente de récupérer l'objet depuis ce fichier.

2.2.4 Catégorisation des paquetages

L'unité de base en Ada réparti est l'unité de bibliothèque[3], c'est-à-dire qu'il n'est pas possible d'avoir un paquetage présent en partie sur un nœud et en partie sur un autre. Les paquetages, dans le langage Ada, sont catégorisés à l'aide de directives de compilation, dont la hiérarchie est présentée sur la figure 2.1 page 47. Une flèche d'une catégorie C vers une catégorie C' indique qu'un paquetage catégorisé C peut dépendre sémantiquement d'un paquetage catégorisé C'.

Ce schéma montre qu'un paquetage marqué comme Shared_Passive ne pourra avoir de visibilité que sur d'autres paquetages avec la même catégorisation ou sur d'autres paquetages avec une 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 [RM, 1995, E.2].

Les différentes catégories de paquetages[4] signifient respectivement :

Remote_Call_Interface : le paquetage contient dans sa partie visible la déclaration des sous-programmes pouvant être appelés à distance et de types transportables. C'est à partir de tels paquetages que l'on générera les souches et les squelettes composant l'infrastructure de l'application répartie.

Remote_Types : le paquetage ne contient, dans la partie visible de sa déclaration, que des types pouvant être transportés entre les différents nœuds d'un système

[3] Nous rappelons qu'une unité de bibliothèque est un paquetage de plus haut niveau, non imbriqué dans une partie déclarative.

[4] Une autre catégorisation (Preelaborate) existe mais ne concerne pas la répartition et ne sera pas évoquée ici.

Exemple 2.3 Manipulation de flux

```
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
procedure P is
   F : File_Type;
   S : Stream_Access;
   X : Integer;
   Y : Float;
begin
   Open (F, In_File, "mon_fichier");
   --  Le fichier F est ouvert en lecture seule
   --  (mode In_File)

   S := Stream (F);
   --  S est le flux correspondant au fichier F

   Integer'Read (S, X);
   --  Lecture d'un entier dans le flux

   Float'Read (S, Y);
   --  Lecture d'un nombre en virgule flottante

   [...]
end P;
```

Exemple 2.4 Manipulation de flux sur un type non contraint

```
with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
procedure P is
   F : File_Type;
   S : Stream_Access;
   X : constant String := Read_String_From_Keyboard;
begin
   Create (F, Out_File, "mon_fichier");
   --  Le fichier F est ouvert en écriture
   --  (mode Out_File)

   S := Stream (F);
   --  S est le flux correspondant au fichier F

   String'Output (S, X);
   --  Écrit dans le flux la chaîne X de taille
   --  inconnue. Cela est équivalent à faire:
   --     Integer'Write (S, X'First);
   --     Integer'Write (S, X'Last);
   --     for J in X'range loop
   --        Character'Write (S, X (J));
   --     end loop;
   --
   --  La lecture se fera à l'aide de 'Input, comme dans:
   --     T : constant String := String'Input (S);
end P;
```

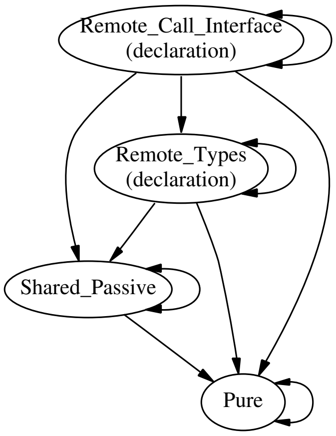


FIG. 2.1 – Dépendances possibles entre les différentes catégories de paquetages

réparti. On pourra notamment définir des pointeurs sur sous-programmes ou objets distants.

Shared_Passive : le paquetage ne contient aucun flot de contrôle par lui-même, ni aucun type permettant de créer un flot de contrôle (par exemple un type définissant une tâche). Les variables définies dans la partie visible d'un tel paquetage seront aisément accédées depuis plusieurs partitions du système réparti.

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.

Les sections suivantes décrivent plus en détail les différentes catégorisations utilisées dans le cadre de l'annexe des systèmes répartis.

2.2.5 Catégorisation Remote_Call_Interface

Les paquetages catégorisés comme Remote_Call_Interface contiennent, dans leur partie déclarative, des spécifications de sous-programmes pouvant être appelés à distance. L'exemple 2.5 page suivante présente un tel paquetage.

Le principe d'utilisation est simple : comme expliqué ci-dessus, un programme réparti écrit en Ada est composé de plusieurs partitions actives ou passives. Chaque partition active contient zéro, un ou plusieurs paquetages portant la catégorisation Remote_Call_Interface. Lorsqu'un programme Ada fait référence à un sous-programme se trouvant dans un de ces paquetages, deux cas peuvent se produire :

1. Si le paquetage appellable à distance se trouve sur la même partition que le code y faisant référence, un appel local traditionnel est utilisé ; il n'y a aucune communication entre les partitions. Un cas particulier existe en présence de la directive de compilation All_Calls_Remote, traité plus bas.
2. Si le paquetage appellable à distance se trouve sur une autre partition que le code y faisant référence, l'exécutif gérant les systèmes répartis (abrégé en **PCS** pour

Exemple 2.5 Paquetage appellable à distance et appel distant

```
package RCI is

   pragma Remote_Call_Interface;
   --  Les sous-programmes définis dans ce
   --  paquetage peuvent être appelés à
   --  distance.

   function Factorielle (N : Natural) return Positive;
   --  Calcule N!. Ce calcul peut être effectué à
   --  distance automatiquement. L'exception
   --  Constraint_Error sera levée si la valeur de la
   --  factorielle excède la capacité d'un Positive.

end RCI;


package body RCI is

   function Factorielle (N : Natural) return Positive is
   begin
      if N < 2 then
         return 1;
      else
         --  Cet appel récursif est un appel local, car le
         --  paquetage RCI, dans lequel est définie la
         --  fonction Factorielle, est le paquetage courant!

         return N * Factorielle (N-1);
      end if;
   end Factorielle;

end RCI;

with Ada.Text_IO, RCI;
--  Déclare la visibilité sur les paquetages Ada.Text_IO (local)
--  et RCI (distant).

procedure Test is
begin
   --  Affiche localement 10!, qui a été calculé à distance si
   --  la procédure Test et le paquetage RCI sont placés sur des
   --  partitions différentes.

   Ada.Text_IO.Put_Line (Positive'Image (RCI.Factorielle (10)));
end Test;
```

"*Partition Communication Subsystem*") génère un appel de sous-programme à distance tel que décrit dans la section 1.4.3 page 29.

Notons que chacun des paquetages catégorisés comme Remote_Call_Interface ne peut se trouver que sur une seule partition active de l'application répartie à la fois. Toutefois, cette unicité n'est que sémantique, au sens où il est possible d'utiliser plusieurs réplicats d'un tel paquetage du moment que cette réplication active est transparente pour l'utilisateur.

En plus de la directive Remote_Call_Interface, deux autres directives de compilation peuvent être utilisées dans un paquetage catégorisé par cette première :

All_Calls_Remote : cette directive de compilation s'applique sur l'ensemble d'un paquetage appellable à distance. Sa présence signifie que l'utilisateur souhaite que tous les appels depuis d'autres unités Ada soient routés à travers l'exécutif de l'annexe des systèmes répartis (le PCS), même si le paquetage et le code y faisant référence sont sur la même partition.

Asynchronous : une procédure indiquée comme étant Asynchronous est une procédure unidirectionnelle. Elle ne renvoie aucun argument, toutes les exceptions levées sont définitivement perdues et l'exécution de l'appelant peut continuer avant la fin de l'exécution de la procédure appelée.

Ces deux directives sont présentées dans l'exemple 2.6.

Exemple 2.6 Utilisation de la directive de compilation Asynchronous

```
package Advanced is

   pragma Remote_Call_Interface;

   pragma All_Calls_Remote;
   --  Tous les appels devront être routés par
   --  l'intermédiaire du PCS.

   procedure Log (Event : in String);
   pragma Asynchronous (Log);
   --  Les appels à la procédure Log se feront
   --  de manière unidirectionnelle et seront
   --  par conséquent non bloquants.

end Advanced;
```

Afin d'alléger certaines phrases, nous appellerons parfois **paquetage appellable à distance** un paquetage catégorisé Remote_Call_Interface. Bien entendu, cela ne rend pas l'entité "paquetage" adressable pour autant.

2.2.6 Catégorisation Remote_Types

L'échange d'un pointeur entre deux partitions n'aurait que peu de sens. En effet, un pointeur désigne un espace réservé dans l'environnement d'exécution ; le transporter sans précautions reviendrait à pointer au hasard dans l'environnement d'exécution de la partition cible.

Cependant, un programmeur peut souhaiter faire des appels de sous-programmes en passant comme paramètres des structures de données complexes, en particulier des structures pouvant contenir des pointeurs. Ceci est possible en Ada grâce à la **surcharge**[5] des attributs Read et Write de ces types, qui permettent de décrire la manière dont un type complexe peut être décomposé en succession de types simples ainsi que la transformation inverse [RM, 1995, 13.13.2(35)].

La présence de la directive de compilation Remote_Types dans la déclaration d'un paquetage Ada demande au compilateur de vérifier que tous les types présents dans la partie visible de cette déclaration sont transportables, c'est-à-dire qu'ils possèdent les attributs Read et Write (voir section 4.7 page 79). Tous les types simples du langage possèdent ces attributs, ainsi que tous les types composites ne contenant que des types ayant eux-mêmes ces attributs. Par exemple, une chaîne de caractères étant un simple tableau de caractères, le type String possède des attributs Read et Write, de même qu'un enregistrement composé d'une chaîne de longueur donnée et d'un nombre en virgule flottante. Les types limités et les pointeurs ne possèdent pas ces attributs par défaut, et l'utilisateur doit les définir explicitement s'il souhaite rendre ces types transportables [RM, 1995, 13.13.2(36)].

L'exemple 2.7 présente un paquetage catégorisé à l'aide de la directive de compilation Remote_Types ; il contient la déclaration d'un type pointeur sur chaîne, ainsi que les attributs associés permettant d'en faire un type transportable. Le corps de ces attributs se trouve sur l'exemple 2.8 page ci-contre.

Exemple 2.7 Définition d'un type pointeur transportable

```
with Ada.Streams;

package RT is

   pragma Remote_Types;
   --  Les types définis ici seront transportables

   type String_Access is access String;
   --  Ce type "pointeur sur chaîne de caractères"
   --  n'est pas transportable a priori.

private

   procedure Read
     (S : access Ada.Streams.Root_Stream_Type'Class;
      O : out String_Access);
   procedure Write
     (S : access Ada.Streams.Root_Stream_Type'Class;
      O : in String_Access);
   for String_Access'Read  use Read;
   for String_Access'Write use Write;
   --  Grâce à ces deux attributs, le type est rendu
   --  transportable.

end RT;
```

[5] La surcharge est ici le mécanisme permettant de remplacer les attributs par défaut fournis par le langage par des attributs définis par l'utilisateur.

Exemple 2.8 Implémentation des attributs Read et Write

```
package body RT is

   procedure Write (S : access Root_Stream_Type'Class;
                    O : in String_Access)
   is
   begin
      --  Par convention, on choisit d'écrire le
      --  booléen False en premier dans le flux
      --  si le pointeur est un pointeur nul. Sinon,
      --  on écrira True suivi de la chaîne de caractères.

      if O = null then
         Boolean'Output (S, False);
      else
         Boolean'Output (S, True);
         String'Output (S, O.all);
      end if;
   end Write;

   procedure Read (S : access Root_Stream_Type'Class;
                   O : out String_Access)
   is
   begin
      --  Si le booléen que l'on lit est False, alors
      --  nous sommes en présence d'un pointeur nul.
      --  Sinon, on crée une nouvelle chaîne de
      --  caractères à partir de la chaîne lue et on
      --  renvoie un pointeur sur cette chaîne.

      if Boolean'Input (S) then
         O := new String'(String'Input (S));
      else
         O := null;
      end if;
   end Read;

end RT;
```

2.2.7 Catégorisation Shared_Passive

Les paquetages catégorisés Shared_Passive correspondant à un état global d'un programme réparti. Ces paquetages peuvent être placés indifféremment sur des partitions actives ou passives. Les variables définies dans ces paquetages sont visibles depuis les différentes partitions. Un paquetage catégorisé comme Shared_Passive peut également contenir des objets protégés simples, c'est-à-dire des objets protégés ne possédant que des fonctions et procédures d'accès, et aucune file d'attente.

2.2.8 Catégorisation Pure

Cette catégorisation, qui n'est pas spécifique à l'annexe des systèmes répartis, permet la déclaration de paquetages déterministes et sans effet de bord. Le déterminisme indique que l'appel d'un sous-programme donné avec les mêmes arguments en entrée produira systématiquement les mêmes arguments en sorties, tandis que l'absence d'effet de bord implique que l'on peut omettre des appels à ces sous-programmes lorsque les résultats sont déjà connus (grâce à un appel précédant) ou seront inutilisés par la suite.

2.2.9 Références sur entités distantes

Il existe deux types d'entités sur lesquels il est possible d'obtenir une référence distante en Ada : les types étiquetés limités privés, utilisés pour la programmation orientée objet, 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.

Références sur objets distants

Rappelons qu'un **type étiqueté** ("*tagged type*" en anglais) est une classe dans la terminologie objet. Un **type limité** ("*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** ("*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 catégorisation Remote_Types ou Remote_Call_Interface est un pointeur sur objets distants [RM, 1995, E.2.2(9)] 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).

On trouvera un tel pointeur sur objets distants dans l'exemple 2.9 page 54. Un objet de type RACW (de l'anglais "*Remote Access to Class-Wide*") peut pointer aussi bien sur un objet local que sur un objet distant. Toute utilisation de ce pointeur 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 :

1. il est interdit de les déréférencer en dehors d'un appel de méthode [RM, 1995, E.2.2(16)] ;
2. il est interdit de les convertir en un type pointeur local [RM, 1995, E.2.2(15)].

En effet, ces deux opérations reviendraient à pouvoir accéder localement 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 section 2.2.5 page 47 décrit les effets de la directive Asynchronous lorsqu'elle est appliquée à un sous-programme. Cette même directive peut être appliquée sur un type pointeur sur objet distant ; cela revient à rendre unidirectionnelles toutes les méthodes pouvant potentiellement l'être, à savoir les procédures n'ayant aucun paramètre de mode out ou in out.

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. Un exemple de déclaration est présenté dans l'exemple 2.10 page suivante.

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

1. ils ne peuvent pointer que sur des sous-programmes appellables à distance, c'est-à-dire définis dans un paquetage catégorisé à l'aide de la directive Remote_Call_Interface [RM, 1995, E.2.2(12)] ;
2. ils ne peuvent être convertis qu'en un autre type pointeur sur sous-programme distant [RM, 1995, E.2.2(11)].

La directive de compilation Asynchronous peut être appliquée sur ce type pointeur, rendant ainsi unidirectionnels les appels aux procédures pointées. Cela fournit un moyen permettant au développeur de transformer une procédure initialement bidirectionnelle en procédure unidirectionnelle.

2.2.10 Généricité et paquetages catégorisés

Les constructions de haut niveau habituellement mises à la disposition du développeur par le langage Ada restent disponibles. Parmi elles, les paquetages génériques sont utilisables avec les différentes catégorisations de l'annexe des systèmes répartis.

L'exemple 2.11 page suivante montre de quelle manière un paquetage générique catégorisé comme Remote_Call_Interface peut être instancié. Notons que l'instance doit elle aussi être catégorisée explicitement, les catégorisations n'étant pas héritées lors d'une instantiation de paquetage générique.

Exemple 2.9 Définition d'un pointeur sur objet réparti

```
package RACW is

   pragma Remote_Types;

   type Designated is tagged limited private;

   procedure Do_Something (D : in out Designated);

   type Pointer is access all Designated'Class;
   --  Pointer est un pointeur distant pointant sur
   --  les objets d'un type dérivant de Designated.

private

   type Designated is tagged limited record
      Counter : Natural := 0;
   end record;

end RACW;
```

Exemple 2.10 Définition d'un pointeur sur sous-programme distant

```
package RAS is

   pragma Remote_Types;

   type Proc_Access is
     access procedure (X : in String; O : out Natural);
   --  Le type Proc_Access peut pointer sur n'importe
   --  quelle procédure appelable à distance et
   --  présentant la signature in String, out Natural.

end RAS;
```

Exemple 2.11 Paquetage générique catégorisé accompagné d'une de ses instances

```
generic
   type T is range <>;
   --  Le type T est un type entier
package GRCI is
   pragma Remote_Call_Interface;
   function Double (X : T) return T;
end GRCI;

with GRCI;
package Integers is new GRCI (Integer);
pragma Remote_Call_Interface (Integers);
--  La catégorisation doit être répétée ici
```

2.2.11 Traitement des exceptions

Mis à part les changements introduits par l'utilisation de la directive Asynchronous, le traitement des exceptions dans une application répartie écrite en Ada ne diffère aucunement de celui disponible 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 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. De la même manière, sur la partition P_2, où ce type d'exception ne peut pas être désigné nommément, il est possible d'en obtenir le nom, tel que connu sur P_1, par le biais de l'attribut Exception_Name, comme dans la situation d'une application monolithique.

2.3 Le paquetage System.RPC

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

Le manuel de référence définit l'interface à utiliser entre le compilateur et l'exécutif de l'annexe des systèmes répartis [RM, 1995, E.5]. Un paquetage unique, System.RPC, définit certains types et sous-programmes qui devront être utilisés comme points d'entrée de l'exécutif. La déclaration de ce paquetage est reproduite dans l'annexe B page 191.

L'intention qui a amené les concepteurs du langage à adopter une séparation aussi nette était de permettre à plusieurs compilateurs d'origines différentes d'utiliser le même exécutif pour la répartition. Nous verrons dans la section 12.1 page 161 que malgré ces efforts, il paraît difficile de partager un exécutif entre plusieurs compilateurs.

Le processus consistant à rassembler les différents paquetages en partitions est un processus post-compilatoire non normalisé [RM, 1995, 10.2(24)]. L'outil extérieur GNATDIST décrit dans la section 3.2 page 65 permet de réaliser cette phase.

La suite de cette section détaille les différentes entités définies dans le paquetage System.RPC.

2.3.1 Un identificateur unique pour chaque partition

Chaque partition d'un programme réparti Ada se voit attribuer, selon des modalités non définies dans l'annexe des systèmes répartis, un identificateur numérique unique de type Partition_ID. Cet identificateur, qui peut être récupéré au moyen de l'attribut du même nom, a deux utilités :

1. il permet de désigner de manière non équivoque la partition vers laquelle un message sortant doit être aiguillé [RM, 1995, E.5(17)] ;

2. il permet éventuellement à l'utilisateur d'obtenir des informations sur une partition par le biais de paquetages fournis par l'implémentation [RM, 1995, E.1(16)].

2.3.2 Le type de flux Params_Stream_Type

Un nouveau type de flux (voir section 2.2.3 page 44) appelé Params_Stream_Type est chargé de recueillir les données à transmettre lors d'un appel de sous-programme à distance ainsi que les résultats associés. Ce type est accompagné des deux méthodes Read et Write[6], qui doivent être surchargées pour toute dérivation concrète de la racine des types de flux Root_Stream_Type [RM, 1995, 13.13].

Ce type comporte de plus un discriminant de type entier nommé Initial_Size, dont le but est de fournir par anticipation à l'exécutif une estimation de la taille des données que l'on souhaite insérer dans le flux. Ceci permet d'allouer un bloc de la bonne taille dès la création du flux, en évitant autant que possible des copies de tampons qui ralentiraient les requêtes.

2.3.3 Les procédures Do_RPC et Do_APC

Deux procédures, appelées Do_RPC et Do_APC, sont définies dans le paquetage d'interface System.RPC. Elles doivent être appelées depuis le code généré par le compilateur Ada pour effectuer, respectivement, un appel de sous-programme distant et un appel de procédure unidirectionnel [RM, 1995, E.5(17-20)].

2.3.4 Le type RPC_Receiver

Étant donné que l'esprit de l'annexe des systèmes répartis envisageait la possibilité de changer de compilateur tout en conservant le même exécutif, un type RPC_Receiver, pointeur sur procédure, a été défini. Cela permet à l'exécutif de l'annexe des systèmes répartis de transmettre les requêtes entrantes à une procédure spécifique générée par le compilateur ; cette procédure extrait les paramètres de chaque requête, fait un appel au sous-programme concerné et emballe le résultat. Une procédure Establish_RPC_Receiver est utilisée par le compilateur pour mettre en place cette **procédure de rappel** ("*callback*" en anglais) [RM, 1995, E.5(21)].

2.4 Les autres systèmes répartis en Ada

Si le sujet de ce mémoire porte sur l'annexe des systèmes répartis d'Ada 95, ce document serait incomplet s'il ne citait pas les alternatives qui ont été étudiées et parfois implémentées. Cette section contient un bref historique de la répartition en Ada ; cet état de l'art ne prétend pas être exhaustif, mais a pour vocation de servir de point d'entrée aux recherches sur les différents systèmes.

Avant même sa première normalisation en 1983, Ada a été utilisé pour créer des systèmes répartis. Les militaires américains et leurs contractants devaient développer des

[6] Il s'agit bien ici de deux méthodes de la classe racine Root_Stream_Type, à ne pas confondre avec les attributs Read et Write définis dans la section 2.2.3 page 44.

systèmes communicants et n'avaient guère le choix du langage, imposé par leur ministère. En outre, les supporters d'Ada souhaitaient bénéficier des apports du typage fort et de la sécurité qui y est rattachée pour développer des systèmes répartis robustes. Le nombre de mécanismes développés était tel que dès 1985 une étude comparative a été présentée dans [Armitage et Chelini, 1985].

Déjà à cette époque, une polémique sur l'unité de base pour la répartition est apparue : certains préconisaient de répartir les paquetages d'un programme Ada [Jessop, 1982] tandis que d'autres trouvaient plus naturel de répartir des tâches [Clemmensen, 1982]. Cette polémique subsiste encore de nos jours : des systèmes récents, basés sur Ada 95, répartissent des tâches plutôt que des paquetages, tel que Drago [Miranda et al., 1996]. Ces différentes politiques ont été étudiées en détail dans [Volz, 1990].

Il semble que les systèmes Ada répartis spécifiques aux applications embarquées aient vu le jour en 1983 [Cornhill, 1983] [Reynolds et al., 1983]. La notion de partitions autonomes s'exécutant sur des processeurs hétérogènes a depuis été largement reprise [Kamrad et al., 1987] [Jha et al., 1989a] [Tomayko et Smith, 1990], même si le cas spécifique d'une architecture homogène continue à être considéré [Kordon et Sens, 1991], notamment en raison des possibilités d'optimisation liées au partage d'une même représentation des données en mémoire. À titre d'anecdote, on pourra noter que [Hutcheon et Wellings, 1988], [Knight et Rouleau, 1988] et [Jha et al., 1989b] ont réintroduit le terme de partition, qui semble avoir quasiment disparu de la littérature technique pendant quelques années (la dernière référence avant 1988 semble être [Dapra et al., 1984]).

Différents modes de communication ont été étudiés dans le cadre de ces systèmes. L'envoi de messages, tel que décrit dans [Rosenblum, 1987], a progressivement cédé du terrain face aux appels de procédures à distance [Sacha, 1989] [Gargaro et al., 1992]. [Kermarrec et Pautet, 1993] ont également conçu une mémoire partagée répartie pour Ada.

Les aspects liés aux applications temps réel, une des cibles de prédilection du langage Ada, n'ont pas été oubliés : [Bamberger et al., 1988] et [Carey, 1988] présentent DARK, un noyau temps réel réparti écrit en Ada permettant la répartition sans modification du langage. Plus tard, [Kermarrec et Pautet, 1992] ont proposé un ensemble de composants logiciels permettant de développer des applications Ada temps réel réparties.

L'idée d'intégrer la répartition au cœur du langage a sérieusement été envisagée en 1989 [Volz et al., 1989] [Gargaro et al., 1990] [Dobbing, 1990]. Cette intégration avait comme ambition de supprimer l'utilisation de traducteurs qui transformaient un code source Ada en insérant des appels aux routines chargées de la communication. De telles traductions sont présentées dans [Bishop et al., 1987], [Krishnan et al., 1988] et [Boussand et al., 1992].

Finalement, un modèle partiellement inspiré de la méthode orientée objets DRAGOON [Maio et al., 1989] [Atkinson et Goldsack, 1989] [Atkinson et al., 1990] a été retenu pour Ada 95, combinant l'appel de sous-programmes à distance avec les objets répartis.

2.5 Conclusion

L'annexe des systèmes répartis normalise à la fois les aspects sémantiques liés à la répartition et l'interface entre le code généré par le compilateur et l'exécutif des systèmes répartis. La diversité des possibilités offertes par cette annexe permet d'envisager son application à une classe de problèmes très large. Il est notamment possible d'utiliser :

1. des appels de sous-programme distants, en conservant toute la sémantique du langage ;
2. des objets répartis, similaires à ceux trouvés dans CORBA ou RMI ;
3. des entités partagées, perçues comme une zone de mémoire partagée entre partitions ;
4. des objets de type complexes, qui seront transformés de manière à les rendre transportables.

L'intégration de la répartition au sein du langage permet à un développeur connaissant déjà le langage Ada d'être immédiatement opérationnel, sans avoir à apprendre de nouvelles constructions syntaxiques. En particulier, la répartition *a posteriori* d'une application monolithique ne nécessite pas le déclenchement systématique d'une nouvelle phase de conception. Les paradigmes couramment utilisés tels que l'appel de sous-programme ou l'appel de méthode d'un objet sont disponibles sous leur forme répartie. De plus, la possibilité offerte par le langage d'utiliser des paquetages génériques est également applicable à une application répartie.

DEUXIÈME PARTIE

Phase de compilation

Cette partie présente dans le chapitre 3 le compilateur GNAT et la chaîne de développement associée, ainsi que Glade, son implémentation de l'annexe des systèmes répartis. Le chapitre 4 détaille, pour sa part, la partie du compilateur dédiée à l'interface avec Glade.

3

GNAT et GLADE

3.1 La chaîne de développement : GNAT

GNAT[1] est non seulement un compilateur Ada, mais également une chaîne complète de développement de programmes Ada [Schonberg et Banner, 1994]. Il a tout d'abord été développé à l'université de New-York, puis a été repris par la société commerciale Ada Core Technologies, qui en assure l'évolution ainsi que la maintenance.

GNAT fait partie intégrante du projet GNU[2], dont le but ultime est de fournir un système d'exploitation complet et tous ses outils. C'est un **logiciel libre**, distribué sous licence GPL[3] : tout le monde a le droit d'en modifier les sources, mais doit redistribuer ses modifications en même temps que le produit modifié. De surcroît, ces modifications doivent utiliser la licence d'origine, et constituent également du code source sous forme de logiciel libre.

3.1.1 Le compilateur

Le compilateur Ada vient se greffer au cœur de l'architecture GCC[4] ; cette architecture, prévue initialement pour compiler du code écrit en langage C, a évolué en un générateur de code multi-langages et multi-cibles. Il permet ainsi de construire un grand nombre de compilateurs natifs et croisés pour plusieurs langages de programmation, tels que C, C++, Fortran, Pascal et Ada. La partie frontale est spécifique au langage considéré, alors que la partie dorsale se chargeant de la génération de code est commune à tous les langages mais spécifique à une plate-forme cible.

La partie frontale du compilateur Ada est elle-même écrite en Ada. Cela implique que pour compiler GNAT, il faut déjà disposer de GNAT lui-même ; cette phase est appelée le "*bootstrapping*" en anglais. Cela ne pose pas de problème en pratique pour les développeurs du compilateur, étant donnée la facilité de générer des compilateurs

[1] GNAT signifiait à l'origine "*GNU NYU Ada Translator*", mais les développeurs de GNAT ont souhaité abandonner cet acronyme, et GNAT ne signifie plus maintenant que GNAT.

[2] Acronyme autoréférant signifiant "*GNU's Not Unix*".

[3] General Public License

[4] "*GNU C Compiler*" ou, depuis peu, "*GNU Compiler Collection*"

croisés pour des architectures supportées par GCC ; il suffit en effet de partir d'une version tournant sur une plate-forme déjà supportée et de construire un compilateur croisé pour la nouvelle plate-forme, puis d'utiliser ce compilateur croisé pour construire un compilateur natif pour la plate-forme cible.

La figure 3.1, inspirée de [Comar et al., 1994], décrit les différentes phases de la compilation d'un programme source Ada. Le code source est tout d'abord analysé lexicalement et syntaxiquement afin de générer un arbre syntaxique. GNAT n'utilise pas les outils tels que bison ou yacc pour analyser le programme source comme le reste des compilateurs de la famille GCC, mais possède son propre analyseur, qui offre, en plus de ses performances, de meilleurs possibilités d'analyse et de présentation des erreurs dans le code source. L'arbre syntaxique est ensuite décoré par la phase d'analyse sémantique, puis étendu lors de la phase d'expansion. L'arbre décoré obtenu est ensuite transmis à GiGi[5] qui le passe par fragments au dorsal de GCC en vue de générer du code en langage d'assemblage, lui-même transmis à l'assembleur.

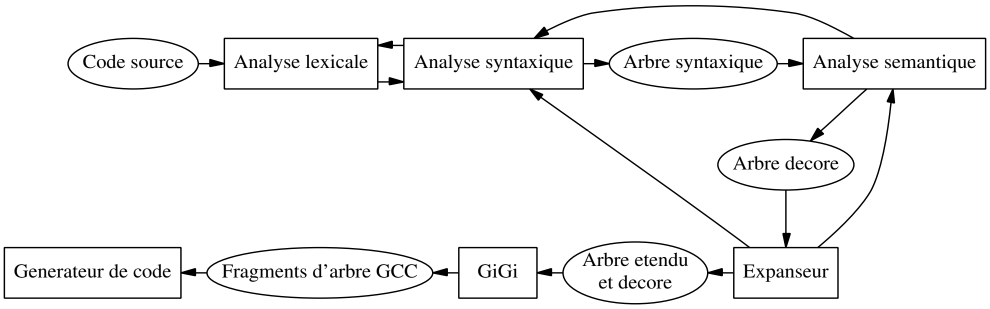


FIG. 3.1 – Phases de compilation Ada

Il est à noter que les phases d'analyse sémantique et d'expansion peuvent elles-mêmes recourir à certaines phases antérieures (cette méthode est notamment présentée dans [Aho et al., 1989]). Ce mécanisme permet de traduire les constructions de haut niveau, comme celles relatives à la gestion des tâches, en simples appels de procédures appartenant à l'exécutif de GNAT, sans surcharger le compilateur en y intégrant la gestion de la concurrence. Par conséquent, GNAT peut être aisément adapté à une nouvelle plate-forme avant que l'exécutif y ait été porté, puisque le compilateur n'a pas besoin de connaître les spécificités matérielles de la machine cible, spécificités qui seront prises en compte plus tard par l'exécutif.

Les exemples 3.1 et 3.2 page 64 présentent respectivement un code Ada et sa version expansée. La version expansée est une construction interne qui n'apparaît que lorsque certains arguments sont donnés sur la ligne de commande du compilateur. On voit clairement que des dépendances supplémentaires ont été ajoutées sur des paquetages appartenant à l'exécutif de GNAT, comme system.pack_03. Des appels à des sous-programmes définis dans ces paquetages sont également générés en lieu et place de l'affectation d'un composant (get_03 et set_03). Notons que le code expansé n'est pas du code Ada correct : il contient des clauses use implicites, des identificateurs commençant par des caractères de soulignement ("*underscores*") ainsi que d'autres

[5] GiGi signifie "GNAT *to* GCC ".

extensions, bien qu'il reste proche d'un code source Ada traditionnel. La possibilité de visualiser le code expansé n'est là que pour faciliter la tâche des développeurs du compilateur lui-même, ainsi que pour donner aux utilisateurs curieux la possibilité d'observer le travail effectué par le compilateur ; l'expansion véritable est faîte au niveau de l'arbre sémantique décoré.

3.1.2 Les outils

Un compilateur Ada nu est d'une utilité restreinte ; il ne peut par exemple pas calculer de lui-même l'ordre d'élaboration des différentes unités d'un programme, étant donné que le choix du sous-programme principal est un processus post-compilatoire. C'est pourquoi GNAT est livré comme un ensemble sans cesse grandissant d'outils, permettant de couvrir l'ensemble du cycle de développement d'une application Ada.

Les outils intervenant lors du processus de développement d'un programme Ada sont présentés sur la figure 3.2. Le compilateur gnat1 appelé depuis le programme principal gcc transforme le fichier source en code objet ; la cohérence des différents objets est vérifiée par gnatbind qui choisit également un ordre d'élaboration pour l'ensemble du programme. Le tout est assemblé par l'éditeur de liens gnatlink qui génère un programme exécutable [RM, 1995, 10.1.4].

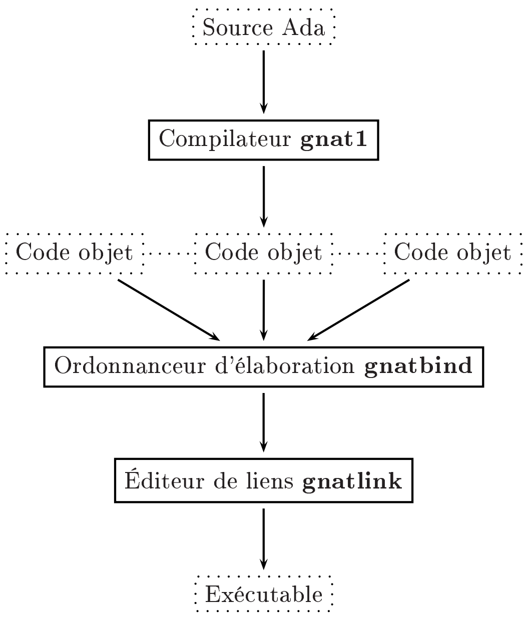


FIG. 3.2 – Vue partielle de la chaîne de développement

Cet exemple simplifié n'utilise pas tous les outils de la chaîne de développement ; leur liste est présentée dans l'annexe C page 193. Il est rare par exemple d'invoquer le compilateur directement ; on utilisera plutôt l'outil de plus haut niveau gnatmake, qui recompile l'ensemble des unités ayant besoin d'être recompilées.

Exemple 3.1 Unité Ada avant expansion

```
with Ada.Text_IO; use Ada.Text_IO;

procedure T is

   type R is record
      A, B, C : Boolean;
   end record;
   pragma Pack (R);
   for R'Size use 3;

   type P is array (0 .. 77) of R;
   pragma Pack (P);

   X : P := (others => (others => False));

begin
   X (10) .A := True;
end T;
```

Exemple 3.2 Unité Ada après expansion

```
with ada;
with ada.text_io;
use ada.text_io;
with system.unsigned_types;
with system.pack_03;
with system;

procedure t is
   type r is record
      a : boolean;
      b : boolean;
      c : boolean;
   end record;
   pragma pack (r);
   for r'size use 3;
   type p is array (0 .. 77) of r;
   pragma pack (p);
   freeze r []
   freeze TpB []
   freeze p []
   freeze T4b []
   A5b : T4b;
   for I6b in 0 .. 77 loop
      set_03 (A5b'address, (integer(I6b - 0)), bits_03!((
         a => false,
         b => false,
         c => false)));
   end loop;
   x : p := A5b;
begin
   T9b : r := r!(get_03 (x'address, (10)));
   T9b.a := true;
   set_03 (x'address, (10), bits_03!(T9b));
   return;
end t;
```

3.2 L'annexe des systèmes répartis : GLADE

De la même manière que les fichiers de l'exécutif Ada sont séparés du compilateur, les fichiers de l'exécutif de l'annexe des systèmes répartis n'interviennent pas directement dans le processus de compilation. Afin de pouvoir remplacer cet exécutif et respecter l'esprit de la norme (comme expliqué dans la section 2.3 page 55), un logiciel dédié à la répartition des applications Ada est distribué de manière indépendante, sous le nom GLADE[6].

GLADE est composé de deux sous-ensembles distincts : l'exécutif appelé GARLIC[7], et l'outil de partitionnement GNATDIST. La partie du compilateur chargée de générer les souches et les squelettes, sera présentée dans le chapitre 4 page 67. Ces souches et ces squelettes font appel à GARLIC en tant qu'exécutif pour les systèmes répartis. GARLIC est décrit dans le chapitre 5 page 87. Le langage de description et le fonctionnement de GNATDIST sont décrits dans le chapitre 10 page 143.

Le projet GLADE est né d'une collaboration entre ADA CORE TECHNOLOGIES, COMPUTER SCIENCES CORPORATION, l'université du Texas, l'ENST de Bretagne et l'ENST à la fin de l'année 1994. Un premier prototype a été développé sous le nom d'ADEPT[8] [Kermarrec et al., 1995, Gargaro et al., 1996]. Ce prototype a été entièrement réécrit par l'équipe actuelle, composée de ADA CORE TECHNOLOGIES et de l'ENST, en profitant de l'expérience acquise lors de la conception de la première version. GNATDIST a également fait son apparition lors de cette seconde période et la génération de souches et de squelettes dans GNAT a été repensée et réécrite.

[6]GNAT Library for Ada Distributed Execution
[7]Generic Ada Reusable Library for Interpartition Communication
[8]Ada Extended with Partitions

4

Compilation et répartition

4.1 Introduction

Comme indiqué précédemment, GNAT procède de manière enchevêtrée à une phase d'analyse sémantique et à une phase d'expansion de code. Ces deux phases ont dû être modifiées de manière à pouvoir générer les souches et les squelettes utilisés dans le cadre d'un système réparti écrit en Ada.

Au commencement de cette thèse, un mécanisme incomplet de production de code pour la répartition existait déjà dans le compilateur. La phase de production est illustrée figure 4.1. L'originalité de la technique utilisée était qu'au lieu de produire directement des fichiers en langage d'assemblage, GNAT, appelé avec des paramètres spéciaux, produisait du code Ada correct correspondant aux souches et aux squelettes, code qui était ensuite compilé indépendamment.

FIG. 4.1 – Première implémentation de la génération de code

Cette méthode avait l'avantage de faciliter le déverminage du code intermédiaire, le code Ada généré pouvant être modifié pour tester de nouveaux schémas d'expansion. Cependant, le fait de se limiter à du code Ada légal pour l'expansion limitait les possibilités d'expression. Notamment, il aurait été extrêmement difficile d'implémenter efficacement les objets répartis sans violer certaines règles du langages (voir section 4.5 page 72), de même que les paquetages génériques catégorisés Remote_Call_Interface.

Pour cette raison, nous avons réécrit les phases d'analyse et d'expansion afin de générer non plus du code Ada intermédiaire mais directement du langage d'assemblage, en utilisant les possibilités d'expansion offertes par GNAT présentées dans la section 3.1.1 page 61. Le schéma actuel est présenté figure 4.2.

FIG. 4.2 – Implémentation actuelle de la génération de code

Après avoir brièvement évoqué l'analyse sémantique, nous présentons les différents mécanismes d'expansion mis en œuvre lors de l'utilisation de l'annexe des systèmes répartis. Les pointeurs sur objets et sous-programmes distants sont détaillés, ainsi que le traitement de la directive de compilation All_Calls_Remote et de la catégorisation Shared_Passive.

4.2 Analyse sémantique

Les unités du compilateur GNAT chargées de réaliser l'analyse sémantique de l'arbre provenant de l'analyse syntaxique sont identifiées sous le nom Sem_ChX, ou X correspond au chapitre du manuel de référence dans lequel ces constructions sémantiques sont définies. Ce modèle est décrit dans [Schonberg et Banner, 1994]. Lorsque les définitions couvrent plusieurs chapitres du manuel de référence, les unités sont alors nommées Sem_YYY, ou YYY désigne le rôle de l'unité (comme Sem_Disp pour le traitement des opérations d'aiguillage[1] des méthodes d'objets). C'est donc tout naturellement que nous avons localisés la majorité de nos ajouts relatifs à la répartition de programmes Ada dans l'unité Sem_Dist sous la forme de sous-programmes appelés depuis divers endroits du compilateur. Cette découpe claire du rôle de chaque unité facilite grandement les ajouts et les expérimentations [Miranda et al., 1999].

La plupart des restrictions applicables aux appels de sous-programmes distants ne peuvent être vérifiés qu'au moment de l'exécution (voir la section 4.7 page 79 concernant les vérifications dynamiques) ; c'est pour cela que le paquetage Sem_Dist est consacré principalement aux pointeurs sur objets et sous-programmes distants. Par exemple, une des opérations réalisées dans ce paquetage consiste à vérifier qu'une référence pointant sur un objet potentiellement distant n'est pas convertie en un type pointeur ne permettant que de désigner un objet local.

4.3 Expansion des constructions complexes

Comparée à la phase sémantique, la phase d'expansion est beaucoup plus complexe. En effet, chaque paquetage catégorisé comme Remote_Call_Interface doit être compilé en deux versions distinctes, sous forme de souche et sous forme de squelette. De plus, certaines vérifications de validité ne peuvent être faites qu'à l'exécution, et il est nécessaire de générer les constructions nécessaires à ces vérifications dynamiques. Les pointeurs sur objets et sous-programmes distants, qui peuvent être déclarés dans des paquetages Remote_Call_Interface ou Remote_Types, nécessitent la génération d'une routine d'aiguillage destinée à traiter les appels entrants.

Lors de la phase de conception de GLADE, il nous est vite apparu que l'annexe des systèmes répartis était, sous un certain regard, beaucoup plus statique que ce que nous aurions espéré [Gargaro et al., 1996]. Les aspects nous ayant le plus gênés concernent principalement la simplicité extrême de l'interface entre le compilateur et l'exécutif, interface réduite au paquetage normalisé System.RPC [RM, 1995, E.5].

Les appels aux sous-programmes d'une unité catégorisée à l'aide de la directive de compilation Remote_Call_Interface sont aiguillés vers la partition sur laquelle cette

[1] "*dispatching*" en anglais.

Exemple 4.1 Service de nommage utilisé par le compilateur

```
function Get_Local_Partition_ID return RPC.Partition_ID;
--  Return the Partition_ID of the current partition

procedure Register_Receiving_Stub
   (Name     : in Unit_Name;
    Receiver : in RPC.RPC_Receiver;
    Version  : in String := "");
--  Register the fact that the Name receiving stub is
--  now elaborated. Register the access value to the
--  package RPC_Receiver procedure.

generic
   RCI_Name : String;
package RCI_Info is
   function Get_RCI_Package_Receiver
     return Interfaces.Unsigned_64;
   function Get_Active_Partition_ID
     return RPC.Partition_ID;
end RCI_Info;
--  RCI package information caching
```

unité a été élaborée [RM, 1995, E.1(9)]. Cependant, il n'existe aucun mécanisme permettant au compilateur de connaître, lors de l'exécution, la partition en question. Le partitionnement étant un processus post-compilatoire [RM, 1995, E.2.3(17)], l'outil de partitionnement doit, d'une manière ou d'une autre, indiquer au compilateur sur quelle partition se trouve chaque paquetage ainsi catégorisé. Cela contredit la notion de dynamicité souhaitable dans un système réparti : les partitions doivent pouvoir être lancées sur des machines différentes sans avoir à recompiler ou réexécuter une phase d'édition de liens (ces arguments sont détaillés dans la section 12.1 page 161).

Nous avons donc introduit un nouveau paquetage System.Partition_Interface en plus du paquetage System.RPC, permettant d'obtenir certaines informations sur le programme réparti en cours. Cette unité, appelée depuis le code généré par la phase d'expansion, offre notamment un service de nommage.

L'exemple 4.1, tiré de la déclaration de ce nouveau paquetage, offre un aperçu de son utilisation. Le code généré lors de la création d'un squelette appelle la procédure Register_Receiving_Stub afin d'enregistrer, auprès de l'exécutif de GLADE, le nom du paquetage catégorisé Remote_Call_Interface courant ainsi que deux champs supplémentaires, Receiver et Version, correspondant respectivement à la procédure d'aiguillage d'appel entrant à destination de ce paquetage et à la version qui sera utilisée pour la vérification de cohérence du système réparti [RM, 1995, E.3(6)].

Inversement, la souche correspondant à un tel paquetage aura, dans le code généré par le compilateur, une instantiation du paquetage générique RCI_Info, qui permettra de déterminer notamment sur quelle partition le squelette de ce paquetage a été élaboré. Cette information est fournie par l'exécutif de GLADE et utilisée pour des appels futurs à Do_RPC. Pour des raisons d'encapsulation et de clarté, ce paquetage est le seul, avec System.RPC, qui puisse être utilisé par GNAT dans le cadre de la génération de code pour l'annexe des systèmes répartis (voir figure 4.3 page suivante). Les possibilités apportées par cette nouvelle interface sont telles que la violation de l'esprit du manuel de référence nous a paru secondaire.

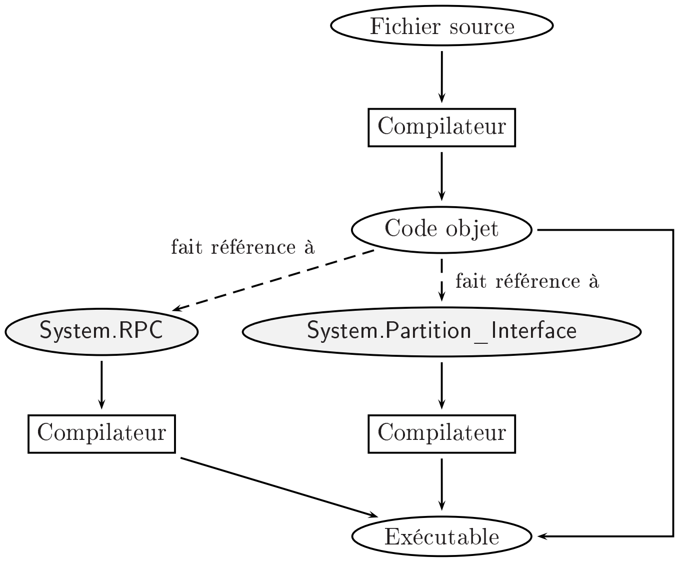


FIG. 4.3 – Deux paquetages d'interface

4.4 Appel de sous-programme distant

4.4.1 Du côté de l'appelant

L'appel d'un sous-programme distant est, du point de vue du compilateur, une opération triviale. Après avoir obtenu, à l'aide des routines présentées précédemment, le numéro de partition sur laquelle le paquetage contenant le sous-programme a été élaboré, après avoir également obtenu l'adresse de l'aiguilleur correspondant à ce paquetage sur la partition distante et le numéro du sous-programme appelé à l'intérieur du paquetage, le compilateur génère l'emballage des arguments suivi d'un appel à Do_RPC puis déballe le résultat (le terme "résultat" est à interpréter au sens large, étant donné qu'une exception ou la valeur finale d'un paramètre de mode out sont considérées comme faisant partie des résultats).

Dans le reste du chapitre, nous montrerons les expansions correspondant au paquetage Transport dont le code est donné dans le fragment 4.2 page ci-contre. La première expansion, présentée dans l'exemple 4.3 page suivante, correspond au code généré pour la souche.

On retrouve dans ce code les grandes phases exposées ci-dessus. On remarque également la présence d'une vérification menaçant de lever Program_Error si une mystérieuse condition n'est pas satisfaite ; toute la lumière sera apportée sur ce sujet dans la section 4.7 page 79. On notera également que l'appel à Reraise_Occurrence n'est pas conditionné par la présence ou non d'une exception ; en fait, l'exception spéciale Null_Occurrence est placée dans le flux lorsque le sous-programme distant s'est terminé

Exemple 4.2 Paquetage d'exemple catégorisé Remote_Call_Interface

```
package Transport is

   pragma Remote_Call_Interface;

   type R is tagged null record;
   --  Ce type simple est trivialement transportable.

   procedure Remote_Proc (P : in R'Class);

end Transport;
```

Exemple 4.3 Souche correspondant au paquetage Transport

```
package transport is
   pragma remote_call_interface;
   type r is tagged null record;
   procedure remote_proc (p : in r'class);

   package R17s is new rci_info (rci_name => "Transport");

   procedure remote_proc (p : in TrC) is
[...]
   begin
      P20s : constant partition_id := R17s.get_active_partition_id;
      S21s : aliased params_stream_type (0);
      R22s : aliased params_stream_type (0);
      E23s : exception_occurrence;
      [program_error when not get_remotely_callable (p._tag)]
      unsigned_64'write(S21s'access, R17s.get_rci_package_receiver);
      subprogram_id'write(S21s'access, 0);
      TrC'output(S21s'access, p);
      do_rpc (P20s, S21s'access, R22s'access, paramsF => 2, resultF =>
        2);
      exception_occurrence'read(R22s'access, E23s);
      reraise_occurrence (E23s);
      return;
[...]
   end remote_proc;

end transport;
```

normalement (ceci se voit clairement dans le squelette de l'exemple 4.4 page suivante, squelette qui sera bientôt examiné en détail).

Bien entendu, si la procédure appelée était une procédure unidirectionnelle, l'appel à Do_RPC serait remplacé par un appel à Do_APC et aucun code ne serait généré pour traiter de paramètre de retour. En effet, dans ce cas les exceptions éventuellement levées par l'appelé sont définitivement perdues (voir section 2.2.5 page 47).

4.4.2 Du côté de l'appelé

Du côté de l'appelé, un plus grand nombre d'opérations est à effectuer. En effet, le squelette doit :

1. créer des routines de déballage des paramètres et d'emballage des résultats pour tous les sous-programmes définis dans le paquetage ;
2. créer une routine d'aiguillage permettant, en fonction du sous-programme appelé, d'exécuter le code correspondant parmi les routines citées ci-dessus ;
3. enregistrer cette routine d'aiguillage et la version du paquetage auprès de l'exécutif de Glade.

L'exemple 4.4 page suivante contient des portions du squelette correspondant au paquetage Transport présenté page précédente. Les parties n'ayant pas de rapport avec la répartition ont été supprimées.

On distingue dans ce code expansé, sous le nom F25s, la routine chargée de déballer les paramètres pour la procédure Remote_Proc, d'appeler celle-ci et d'emballer l'exception (Null_Occurrence sera utilisée si aucune exception n'a été levée). P20s pour sa part correspond à l'aiguilleur : si le numéro de sous-programme extrait du flux est zéro (premier sous-programme du paquetage), alors l'appel entrant sera aiguillé vers la procédure F25s présentée ci-dessus. On voit que le passage d'un numéro de sous-programme invalide ne provoque pas d'erreur, mais est ignoré silencieusement. Ceci est dû à une volonté d'empêcher un déni de service, qui résulterait du passage “à la main” de mauvais paramètres à cette procédure d'aiguillage afin d'arrêter une partition jouant le rôle de serveur. Aucune exception n'est renvoyée à l'appelant, étant donné qu'une telle situation ne peut jamais se produire dans le cadre d'une utilisation normale d'un système réparti en Ada, notamment en raison du contrôle automatique de versions, qui sera décrit dans la section 6.2.1 page 96.

Le paquetage interne P80s pour sa part ne joue un rôle que dans l'élaboration du paquetage englobant, c'est-à-dire le squelette. Il est chargé, par le biais des commandes situées dans son corps, d'enregistrer le squelette auprès de l'exécutif de Glade en utilisant la procédure Register_Receiving_Stub.

4.5 Objets répartis

Il nous a semblé clair, dès le début, que les objets répartis allaient prendre une importance considérable ; ceci a été confirmé par la suite avec le succès de CORBA [OMG, 1998]. Nous avons donc souhaité implémenter ce concept de la manière la plus efficace possible, notamment sans ajouter de surcoût pour un appel de méthode sur un objet local.

Exemple 4.4 Squelette correspondant au paquetage Transport

```
[...]
procedure F25s (S21s : access params_stream_type; S22s : access
  params_stream_type; S21sF : natural; S22sF : natural)
is
   F52s : finalizable_ptr := null;
begin
[...]
   P23s : constant TrC := TrC'input(S21s);
[...]
   begin
      remote_proc (p => P23s);
      exception_occurrence'write(S22s, null_occurrence);
   exception
      when E24s : others =>
         exception_occurrence'write(S22s, E24s);
   end;
   return;
[...]
end F25s;

procedure P20s (S17s : access params_stream_type; R18s : access
  params_stream_type; S17sF : natural; R18sF : natural) is
   P19s : subprogram_id;
begin
   subprogram_id'read(S17s, P19s);
   case P19s is
      when 0 =>
         F25s (S17s, R18s, S21sF => S17sF, S22sF => R18sF);
      when others =>
         null;
   end case;
   return;
end P20s;

package P80s is
end P80s;

package body P80s is
begin
   register_receiving_stub ("Transport", P20s'unrestricted_access,
     transport'version);
end P80s;
[...]
```

4.5.1 Première implémentation

Une première implémentation, proposée initialement par Tucker Taft et basée sur des travaux présentés par la suite dans [Nana Tchamnda, 1996], consistait en la création, sur chaque partition, d'un nouveau type spécifique héritant du type pointé par le pointeur sur objet réparti. La figure 4.4 représente un arbre de hiérarchie tel qu'il pourrait être imaginé sur une partition.

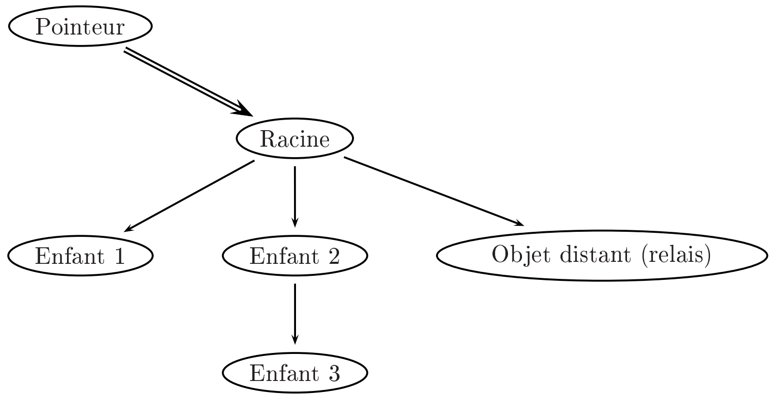


FIG. 4.4 – Arbre de hiérarchie utilisé pour les objets répartis

Chaque fois qu'une méthode était appelée sur un objet cible d'un pointeur réparti, une des deux alternatives était automatiquement choisie :
– si l'objet est un objet local, alors le pointeur pointe directement sur l'objet et la méthode correspondante est exécutée ;
– si l'objet est un objet distant, alors le pointeur pointe sur un relais, c'est-à-dire un type spécifique dérivé du type racine dont les champs supplémentaires comportent les coordonnées de l'objet distant et dont les méthodes sont en fait des appels de sous-programmes distants.

Cette structure, en plus de sa grande efficacité, avait l'avantage de la simplicité : un simple héritage supplémentaire dans le code généré suffisait pour pouvoir adresser, sans modification du code appelant, n'importe quel objet distant. Malheureusement, en testant des cas complexes, nous nous sommes heurtés à deux problèmes insolubles inhérents à ce modèle :

1. Chaque objet relais, en plus des champs permettant de localiser l'objet distant, contenait une copie de chaque champ défini dans le type racine (propriété de l'héritage). Si le type racine est volumineux, alors on peut perdre énormément de place à chaque référence d'objet distant.
2. Si l'objet racine contient des composants dont l'initialisation est obligatoire, comme un discriminant, alors il faut fournir une valeur valide pour chacun de ces composants, ce qui ne peut pas être fait de manière automatique.

Nous avons donc adopté un nouveau schéma, plus original, permettant de prendre en compte l'intégralité du problème mais qui exige de pouvoir réaliser la phase d'expansion directement à l'intérieur du compilateur (voir section 4.1 page 67 à ce sujet).

4.5.2 Implémentation actuelle

Plutôt que de faire dériver l'objet relais de la racine de la hiérarchie, nous en avons fait un type parallèle avec une table de méthodes virtuelles “compatible” avec celle du type racine. Cela signifie que les méthodes correspondantes sont situées au même endroit dans la table d'aiguillage virtuel.

Lorsque le pointeur pointe sur un objet local, les méthodes utilisées sont les méthodes de l'objet local. Lorsque le pointeur pointe sur un objet distant, il pointe en fait sur un objet de structure totalement différente présentée dans l'exemple 4.5, chaque méthode de ce type correspondant à l'appel de la souche correspondante.

Exemple 4.5 Type relais utilisé pour tous les objets distants

```
type RACW_Stub_Type is tagged record
   Origin       : RPC.Partition_ID;
   Receiver     : Interfaces.Unsigned_64;
   Addr         : Interfaces.Unsigned_64;
   Asynchronous : Boolean;
end record;
```

Le champ Origin contient l'identificateur de la partition sur laquelle l'objet a été créé. Receiver contient l'adresse, sur la partition Origin, du programme d'aiguillage permettant de reconstituer l'appel de méthode localement. Addr représente l'adresse de l'objet sur la partition Origin, tandis que Asynchronous indique si le type pointeur a été enrichi d'une directive de compilation pragma Asynchronous. On notera que le type System.Address n'est pas utilisé ; on lui a préféré le type Interfaces.Unsigned_64, assez grand pour contenir une adresse locale sur toutes les architectures pour lesquelles GNAT est disponible.

Comme la figure 4.5 le montre, ceci serait extrêment difficile à faire en Ada, c'est-à-dire sans passer par les possibilités supplémentaires offertes par l'expansion ; en effet, le pointeur doit pouvoir pointer sur deux types d'objets distincts, un objet de la hiérarchie descendant de racine ou un objet de type relais. Les conversions appropriées sont générées pendant cette phase d'expansion.

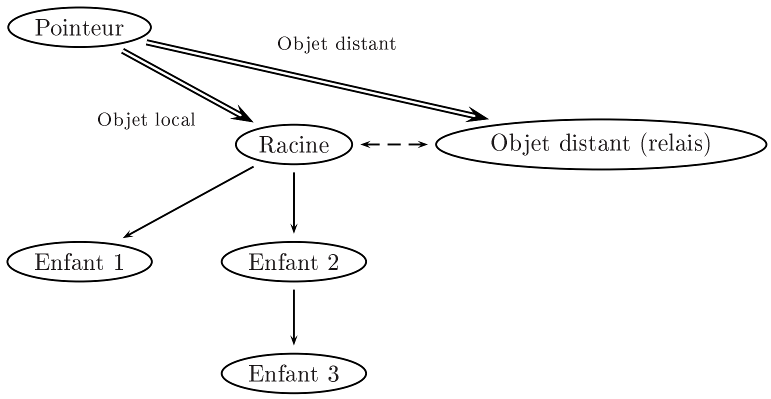


FIG. 4.5 – Implémentation d'un type parallèle

4.5.3 Caractère local d'un objet potentiellement distant

Les attributs Read et Write générés automatiquement pour chaque type pointeur sur objet distant assurent la transmission correcte de ce type de pointeur. Lorsqu'un pointeur potentiellement distant doit être transmis d'une partition à une autre, l'attribut Write correspondant à ce type procède aux opérations suivantes :

Si le pointeur désigne un objet distant, la structure existante décrivant cet objet est recopiée et transmise au destinataire.

Si le pointeur désigne un objet local, une structure décrivant l'objet comme s'il était distant est construite, et transmise au destinataire.

À la réception d'un tel pointeur, l'attribut Read procède, pour sa part, aux opérations suivantes :

Si le pointeur désigne un objet local, l'adresse locale de l'objet est extraite du champ correspondant et est utilisée comme valeur de retour. Il s'agit d'un objet dont une référence a été transmise précédemment à une autre partition, puis qui est revenue sur la partition initiale. Le champ Origin permet de connaître la partition d'origine de l'objet.

Si le pointeur désigne un objet distant, on construit un pointeur sur objet distant à partir des champs présents dans le flux. Si un pointeur sur le même objet a déjà été créé précédemment sur cette partition, on réutilise la structure relais existante en la récupérant dans une table de hachage. Sinon, on créé une nouvelle structure que l'on enregistre dans cette table afin de pouvoir la réutiliser par la suite.

Chaque référence sur un objet distant correspond donc à un objet local de type relais ; malheureusement, il est impossible de désallouer un objet à distance, chaque nouvel objet créant donc une fuite de mémoire. L'instantiation du paquetage prédéfini Ada.Unchecked_Deallocation est en effet rendue impossible par [RM, 1995, E.2.2(17)] qui interdit d'utiliser un type pointeur sur entité distante comme paramètre d'un paquetage générique qui attend un pointeur classique. Ce phénomène indésirable, mais inévitable en l'absence d'un ramasse-miettes ou d'un comptage de référence[2], a été limité par l'utilisation d'une table de hachage, comme expliqué ci-dessus. Chaque objet distant entrant est comparé aux entrées de la table, et s'il est déjà connu, le même objet relais est réutilisé.

4.6 Pointeurs sur sous-programme distant

Les pointeurs sur sous-programmes distants sont moins puissants que les pointeurs sur objets distants ; ces derniers peuvent être utilisés partout où les premiers semblent nécessaires. De plus, un sous-programme pointé est obligatoirement défini dans un paquetage catégorisé comme Remote_Call_Interface [RM, 1995, E.2.2(12)], alors qu'un nouveau type d'objet réparti est plus facilement défini dans n'importe quelle partition sans introduction de nouveau paquetage.

[2] Le mécanisme de comptage de références pour les pointeurs distants n'a pas été étudié : son utilisation dans GNAT pourrait permettre le développement de nouveaux services tels que la migration d'objets.

Pour cette raison, nous nous sommes peu intéréssés aux pointeurs sur sous-programmes distants ; nous nous sommes contentés d'en obtenir une implémentation qui fonctionnait. Cependant, nous avons des idées (exposées ci-dessous) quant à une implémentation plus efficace et plus souple.

4.6.1 Implémentation actuelle

L'implémentation telle qu'elle existe dans GNAT transforme chaque pointeur sur sous-programme distant en une structure exposée dans l'exemple 4.6. Les champs représentent, dans l'ordre, l'adresse du sous-programme local ou zéro si le sous-programme est distant, la partition, l'adresse de l'aiguilleur et le numéro de sous-programme correspondant au sous-programme distant, ainsi qu'un drapeau indiquant si le type de pointeur est unidirectionnel ou non.

Exemple 4.6 Expansion d'un type pointeur sur sous-programme distant

```
type func_access is record
   ras : unsigned_64;
   origin : integer;
   receiver : unsigned_64;
   subp_id : natural;
   async : boolean;
end record;
```

Le code actuellement en place ne fait aucun usage du champ RAS, et utilise l'exécutif des systèmes répartis pour chaque déréférence d'un pointeur sur sous-programme distant, même s'il est local (RAS et Subp_Id sont redondants). Il n'est pas prévu de poursuivre dans cette voie, étant donnée la solution que nous envisageons ci-dessous.

4.6.2 Implémentation future

La solution actuelle présente le désavantage d'avoir à manipuler des structures de données plus volumineuses qu'un simple pointeur. Il est toutefois possible de réduire cette structure en unifiant les références sur sous-programmes distants et les références sur objets distants.

Un sous-programme appelable à distance est d'un certain point de vue similaire à un appel de méthode : seul l'élément représentant l'objet est manquant dans la signature du sous-programme. C'est pourquoi nous avons pensé à représenter chaque type de pointeur sur sous-programme distant par un pointeur sur objet distant, et chaque instance de ce type de pointeur par un objet sans état appelable à distance. Les appels à ces sous-programmes seraient donc traduits en un appel à la seule méthode de l'objet, provoquant un appel à distance si l'objet est distant et appelant à travers un relais le sous-programme local si l'objet est local.

L'exemple 4.7 page suivante expose le schéma proposé pour la traduction d'un type pointeur sur sous-programme distant en type pointeur sur objet distant. Notons qu'il n'y a aucun problème de contexte : un pointeur sur objet réparti peut être déclaré à tous les endroits dans lesquels un pointeur sur sous-programme distant peut l'être

[RM, 1995, E.2.2(9)]. Les exemples donnés ici représentent une expansion symbolique : l'implémentation réelle dans GNAT utiliserait des identificateurs uniques ne risquant pas de générer de conflits avec des déclarations venant de l'utilisateur.

Exemple 4.7 Équivalence des types pointeurs

```
type RAS is access function (X : Integer) return Integer;
--  Déclaration originale de pointeur sur sous-programme
--  distant.

type RAS_Object is tagged limited private;
function Call
   (O : access RAS_Object;
    X : Integer)
  return Integer;
type RAS_Pointer is access all RAS_Object'Class;
--  Déclaration du type équivalent RAS_Pointer
```

La construction d'un objet pointeur sur sous-programme distant peut avoir lieu de deux manières :

1. Utilisation de l'attribut Access sur un sous-programme appelable à distance.
2. Copie d'un autre objet pointant sur un sous-programme distant.

L'exemple 4.8 donne un exemple du traitement du premier cas, le second cas étant trivial. Un nouveau type d'objet est dérivé du type général, et la méthode Call est surchargée afin d'appeler la fonction pointée.

Exemple 4.8 Construction d'un pointeur sur sous-programme distant

```
R : RAS := My_Function'Access;
--  Déclaration d'une variable pointant sur My_Function

type RAS_My_Function is new RAS_Object with null record;
function Call
   (O : access RAS_My_Function;
    X : Integer)
  return Integer;
RAS_My_Function_Object : RAS_My_Function;
R : RAS_Pointer := RAS_My_Function_Object'Access;
--  Déclaration équivalente en utilisant le type RAS_Pointer

function Call
   (O : access RAS_My_Function;
    X : Integer)
  return Integer
is
begin
   return My_Function (X);
end Call;
--  Corps de la méthode Call correspondant au véritable appel
--  de fonction.
```

L'exemple 4.9 page suivante représente la traduction d'un appel de sous-programme distant en appel de méthode à distance. La seule différence entre les deux consiste en l'ajout d'un paramètre contrôlant permettant de choisir l'objet appelé.

Exemple 4.9 Déréférence d'un pointeur sur sous-programme distant

```
A := R (7);
--  Appel de la fonction pointée

A := Call (R, 7);
--  Equivalent de l'appel de la fonction pointée
```

4.7 La notion de type transportable

Les règles de dépendance sémantique présentées dans la section 2.2.4 page 45 ont pour conséquence que les types de tous les arguments formels d'un sous-programme (ou méthode) appelable à distance sont transportables. En effet, les quatre règles suivantes s'appliquent conjointement :

1. il n'est pas possible de définir un type pointeur dans un paquetage Pure ou Shared_Passive ;
2. si un type de pointeur local ou un type limité est défini dans la partie visible d'un paquetage Remote_Types, alors le programmeur doit fournir des attributs d'emballage (Write) et de déballage (Read) pour ce type, le rendant ainsi transportable ;
3. tout type agrégat (structure ou tableau) défini dans la partie visible d'un paquetage Remote_Types doit être soit composé uniquement de types transportables, soit fournir les attributs cités ci-dessus, en faisant ainsi un type transportable à son tour ;
4. tout type agrégat défini dans la partie visible d'un paquetage portant la catégorisation Remote_Call_Interface doit être composé uniquement de types transportables.

Cependant, toutes ces précautions, qui peuvent être vérifiées lors de la phase sémantique, ne s'avèrent pas suffisantes pour empêcher le programmeur d'essayer de transporter des types non-transportables entre partitions.

Considérons de nouveau l'exemple 4.2 page 71, qui définit un paquetage dont le sous-programme Remote_Proc est appelable à distance, ainsi qu'un type R dont l'ensemble de la hiérarchie peut être utilisée comme type de l'argument passé à cette procédure. Rien n'empêche un développeur de définir un nouveau type, comme présenté dans l'exemple 4.10 page suivante, qui hérite du type transportable et l'enrichit avec une partie non transportable (ici un pointeur). Or, il est impossible de savoir, au moment de la compilation de l'unité Transport, avec quel type la procédure Remote_Proc sera invoquée. De même, si l'on dispose dans une unité d'un objet de type R'Class obtenu lors d'un appel de sous-programme, il est impossible de dire lors de la compilation si l'appel de Remote_Proc se fera avec un type transportable ou non.

C'est pour cette raison que le manuel de référence impose une vérification au moment de l'exécution ; le type utilisé doit vérifier une des quatres conditions suivantes, sous peine de voir levée l'exception Program_Error [RM, 1995, E.4(18)] :

1. le type a été défini dans un paquetage Pure ;
2. le type a été défini dans un paquetage Shared_Passive ;

Exemple 4.10 Donnée non transportable

```
with Transport;

package Non_Transport is

   type My_R is new R with record
      A : access Integer;
   end record;
   --  Ce type n'est plus transportable, il contient
   --  un pointeur!

end Non_Transport;
```

3. le type a été défini dans la partie visible d'un paquetage Remote_Types ;
4. le type a été défini dans la partie visible d'un paquetage Remote_Call_Interface.

Cela revient à dire que le type en question aurait pu être utilisé directement comme paramètre d'un sous-programme déclaré dans un paquetage Remote_Call_Interface. De manière concrète, cela signifie que chaque type étiqueté doit contenir un drapeau indiquant s'il vérifie ou non une de ces conditions.

À l'occasion de la réflexion qui a précédé l'implémentation de cette règle, nous avons identifié une faille dans le manuel de référence : une construction comme celle présentée dans l'exemple 4.11 ne subit pas les mêmes limitations, alors qu'elle peut aboutir au même résultat, à savoir la tentative de transport d'un type non transportable. Cela a donné lieu à une demande de révision de la règle [RM, 1995, E.4(18)] afin que celle-ci inclut le cas des paramètres de retour des fonctions[3].

Exemple 4.11 Construction légale mais potentiellement dangereuse

```
package Transport_Func is

   pragma Remote_Call_Interface;

   type R is tagged null record;

   function Remote_Func return R'Class;
   --  Cette fonction risque de retourner un type
   --  étendu par dérivation qui sera non transportable.

end Transport_Func;
```

4.8 Traitement de la directive All_Calls_Remote

La directive de compilation All_Calls_Remote demande à ce que tous les appels à des sous-programmes définis dans un paquetage Remote_Call_Interface soient routés à travers l'exécutif chargé de la répartition, même si ces appels sont normalement

[3]Cette requête a été acceptée dans l'AI95-00215/02

résolus en appels locaux (voir section 2.2.5 page 47). Cela signifie que la souche (pour générer les appels à l'exécutif) et le squelette sont présents là où le squelette seul serait engendrés en l'absence de cette directive de compilation.

Afin d'éviter un conflit de noms entre les véritables sous-programmes (localisés sur le squelette en temps ordinaire) et les sous-programmes jouant le rôle de souches, ces derniers sont renommés. Tous les appels venant d'unités externes utilisent ces nouveaux noms. Cela permet de traiter les appels venant du paquetage lui-même et de ses unités enfants différemment, sans utiliser l'exécutif.

4.9 Mémoire partagée

Les paquetages catégorisés à l'aide de la directive Shared_Passive définissent des espaces de mémoire partagée entre partitions. L'exemple 4.12 donne un exemple de tel paquetage, qui peut contenir des déclarations de types, de variables, et d'objets protégés ne contenant que des procédures et des fonctions (mais pas d'entrées).

Exemple 4.12 Exemple de paquetage de mémoire partagée

```
package SP is
   pragma Shared_Passive;
   --  Ce paquetage définit une zone de mémoire partagée

   X : Integer := 0;
   --  Définition d'un entier partagé entre les partitions

end SP;
```

L'annexe des systèmes répartis ne précise pas dans quelles conditions ces paquetages peuvent être utilisés. Ils ont été prévus par le comité de normalisation pour partager des informations entre des partitions situées sur la même machine physique, dans le cadre d'architectures basées sur des multi-processeurs. Nous avons choisi, pour notre part, de limiter leur utilisation aux partitions partageant un même système de fichiers, ce qui couvre un nombre de cas plus grand que l'idée originale.

Chaque variable est stockée dans un fichier séparé de même nom que la variable qualifiée. Par exemple, l'entier X présenté sur cet exemple utilisera le nom de fichier "sp.x". Le principe de fonctionnement lors d'une référence à la variable est simple : si le fichier existe, alors la valeur qu'il contient est lue, sinon la variable conserve sa valeur (ici sa valeur initiale). Lorsqu'une variable est modifiée, le fichier correspondant à cette variable est mis à jour ou créé pour contenir la nouvelle valeur. L'exemple 4.13 page suivante contient le code expansé pour la variable X définie ci-dessus. Ce code consiste en deux sous-programmes ; le premier est appelé avant chaque lecture de la variable partagée, le second après chaque écriture.

Le traitement des objets protégés procède de la même technique ; il utilise de plus le verrouillage des fichiers offert par le système d'exploitation sous-jacent afin d'assurer la sérialisation de l'accès aux données. Les systèmes de fichiers partagés actuels comme NFS [Corbin, 1993] ou plus récemment CODA [Satyanarayanan et al., 1987] sont suffisamment efficaces pour que l'accès à ces données, variables ou objets proté-

Exemple 4.13 Expansion d'une variable partagée

```
[...]
   procedure sp__xR is
      S : ada__streams__stream_io__stream_access := null;
   begin
      S := system__shared_memory__shared_mem_rfile ("sp.x");
      if S /= null then
         integer'read(S, sp__x);
      end if;
      return;
   end Sp__xR;

   procedure sp__xA is
   begin
      integer'write(system__shared_memory__shared_mem_wfile ("sp.x"),
        sp__x);
      return;
   end sp__xA;
[...]
```

gés, s'effectue en un temps raisonnable.

Notons que cette technique permet d'utiliser les paquetages Shared_Passive pour introduire de la persistance dans un programme Ada, qu'il soit monolithique ou réparti. En effet, si les fichiers correspondant aux variables définies dans de tels paquetages ne sont pas effacés avant l'exécution, les variables reprendront automatiquement leur valeur précédente. De la même manière, il est possible de conserver les fichiers correspondant à une exécution particulière, afin de reproduire plus tard la suite d'une exécution dans les conditions initiales. Cela permet de définir des **points de reprise**, autorisant le redémarrage d'un calcul interrompu prématurément sans le recommencer depuis le début.

4.10 Conclusion

Les mécanismes d'expansion du compilateur GNAT permettent de transformer des constructions propres à la répartition d'application Ada en des appels à l'exécutif des systèmes répartis. Cette réécriture en pseudo-Ada permet d'insérer dans le code expansé des constructions qui seraient impossibles ou difficiles à représenter à l'aide de code Ada légal.

Nous avons réécrit la quasi-totalité de la phase d'expansion relative à la répartition afin de nous affranchir des limites de l'implémentation initiale. Les pointeurs sur objets potentiellement distants ont été conçus et développés dans un but de souplesse et d'efficacité lors de la manipulation d'objets locaux. Nous avons également conçu un modèle de pointeurs sur sous-programmes qui permettra, une fois implémenté, de bénéficier de cette efficacité pour les appels à des sous-programmes locaux représentés comme sous-programmes potentiellement distants.

La directive de compilation All_Calls_Remote est traité par un mécanisme spécifique permettant d'accéder aux sous-programmes concernés à travers l'exécutif des systèmes répartis même lorsqu'ils sont locaux. Le cas des paquetages catégorisés comme Shared_Passive est traité de manière originale en utilisant le système de fichiers partagés sous-jacent, et permet d'étendre les possibilités prévues par le manuel de référence afin d'ajouter de la persistence à des applications Ada, qu'elles soient monolithiques ou réparties.

TROISIÈME PARTIE

L'exécutif de GLADE

Cette partie décrit l'exécutif de Glade, l'architecture adoptée ainsi que les techniques utilisées. Afin d'augmenter la clarté de l'exposé, nous utiliserons les réseaux de Petri colorés comme support pour illustrer les chemins d'exécution empruntés par les requêtes. Le lecteur non familier avec les notations utilisées dans ce contexte pourra consulter une brève introduction à ce formalisme dans l'annexe A. La chapitre 5 décrit l'architecture générale de l'exécutif de Glade. Nous présentons ensuite dans le chapitre 6 les méthodes utilisées pour localiser les ressources utilisées dans une application répartie. Le chapitre 7 détaille la manière dont a été implémenté l'appel de sous-programme à distance. Enfin, le chapitre 8 pose le problème de la terminaison d'une application répartie et présente la solution que nous y avons apportée.

5
Architecture de l'exécutif

5.1 Introduction

Globalement, le cycle de vie d'une partition est toujours le même, comme résumé par le réseau de Petri de la figure 5.1 page suivante. Une partition se connecte à l'application répartie en contactant un point d'accueil dont l'adresse est présente dans le fichier de configuration ou sur la ligne de commande, et obtient un numéro de partition tout en indiquant ses propres coordonées, puis enregistre ses paquetages Remote_Call_Interface locaux. Une fois cette phase d'élaboration terminée, elle effectue deux tâches simultanément :

1. Elle exécute son propre programme principal, représentant son existence inviduelle.
2. Elle traite les appels entrants et les exécute (voir section 7.1 page 107).

Lorsque la première de ces deux actions est terminée, si aucune opération entrante n'est en cours, la partition est localement prête à terminer. Lorsque l'ensemble des partitions d'une application répartie sont prêtes à terminer localement et que cet état est permanent, la terminaison globale est négociée et exécutée.

Toutes ces opérations sont traitées par l'exécutif de GLADE. Cependant, une conception modulaire nous a permis d'isoler les différentes étapes du processus dans des groupes de paquetages différents.

5.2 Une architecture modulaire

La figure 5.2 page 89 présente une vue globale des principaux paquetages de l'exécutif de GLADE. Les unités entourées d'un trait gras sont les paquetages appelés depuis le code généré par le compilateur GNAT, tandis que les paquetages entourés de pointillés sont les paquetages de bas niveau permettant aux différentes partitions de communiquer. Une flèche d'un paquetage A vers un paquetage B indique que A utilise un ou plusieurs services fournis par B.

L'ensemble des paquetages de l'exécutif de GLADE sont réunis sous la hiérarchie Sys-

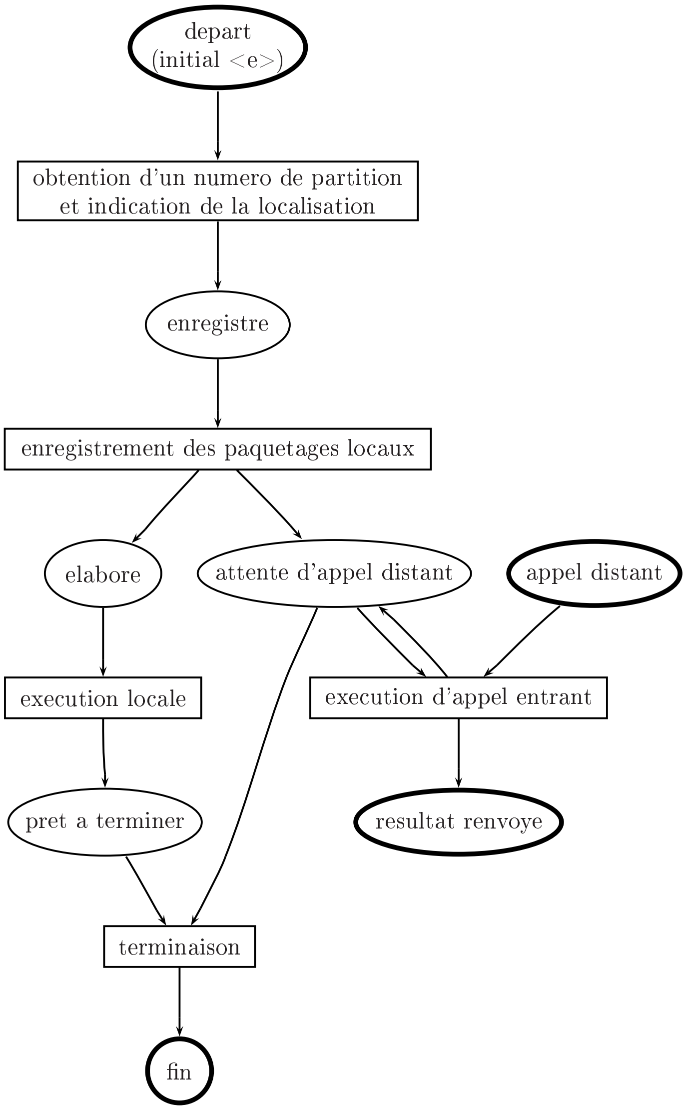


Fig. 5.1 – Cycle de vie d'une partition

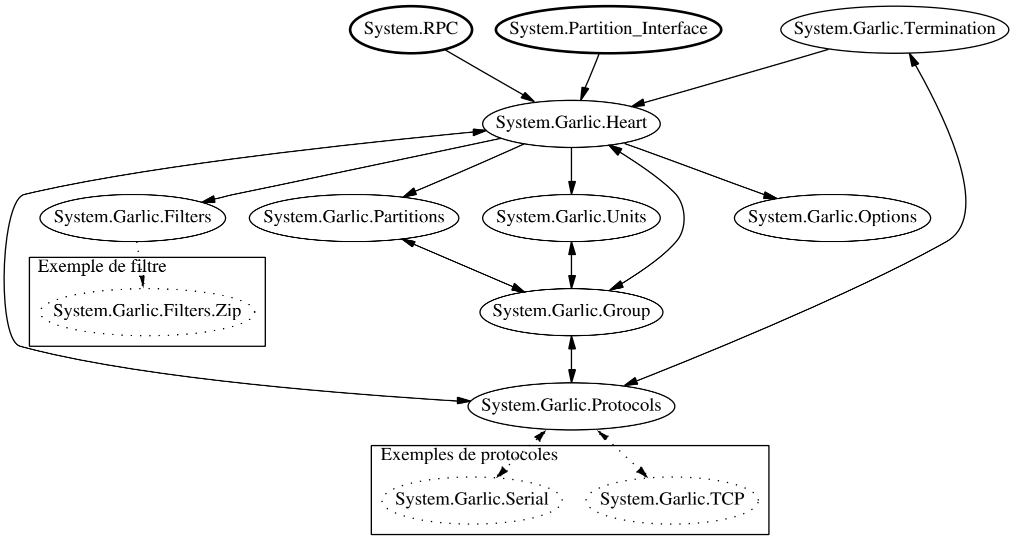


FIG. 5.2 – Exécutif de GLADE

tem.Garlic, afin de ne provoquer aucun conflit de nom ni avec les unités de l'utilisateur ni avec celles de l'exécutif du compilateur. Cette hiérarchie est divisée en trois blocs principaux :

1. Le cœur de l'exécutif effectue toutes les opérations de haut niveau nécessaires au bon déroulement d'un appel de procédure à distance.
2. Les protocoles et les filtres assurent la communication entre partitions.
3. La partie communication de groupe gère les tables des partitions et des unités.

Ces différents blocs sont décrits ci-dessous. Lorsqu'une opération particulière est décrite en détail dans une autre partie de ce document, les références appropriées sont fournies.

5.3 Le cœur de l'exécutif

Les paquetages System.RPC et System.Partition_Interface, décrits dans la section 4.3 page 68, font appel au cœur de l'exécutif de GLADE pour lui transmettre, ou en obtenir, des informations de haut niveau. Les appels à la procédure System.RPC.Do_RPC, notamment, sont transmis à System.Garlic.Heart, qui se charge de générer les appels de sous-programmes à distance sous forme de requêtes décrites dans la section 7.1.1 page 107. Les requêtes d'avortement (voir section 7.1.2 page 109) sont également traitées à ce niveau.

Le cœur de l'exécutif ne communique pas directement avec les partitions distantes ; il le fait par l'intermédiaire des protocoles, décrits ci-dessous, ou de la communication de groupe, décrite dans la section 5.5 page suivante. Des services additionnels mais indépendants font également partie de ce cœur, comme la gestion de la terminaison, qui sera détaillée dans le chapitre 8.

5.4 La communication inter-partitions

Il nous a paru important, dès la première version de GLADE, de séparer nettement la partie gérant la communication proprement dite du reste de l'exécutif de l'annexe des systèmes répartis, afin de pouvoir notamment adopter plusieurs protocoles différents [Gargaro et al., 1996]. Le paquetage System.Garlic.Protocols définit un type de protocole abstrait, dont les seules caractéristiques sémantiques sont d'être fiable et ordonné, c'est-à-dire que les paquets transmis sont supposés arriver en un temps fini (mais non spécifié) dans l'ordre de leur émission. Des types concrets sont dérivés de ce type abstrait et fournissent des implémentations de protocoles réels.

La fiabilité mentionnée ci-dessus implique la détection de panne et de rupture de communication. Si une partition P_i meurt, tous ses voisins doivent le détecter en un temps fini.

Un protocole comme TCP [Feit, 1993] remplit ces conditions et a été choisi comme protocole de référence en raison de son déploiement important. Mais il également possible de construire un protocole ayant les caractéristiques demandées au dessus d'un protocole non fiable tel qu'UDP en ajoutant une couche de fiabilité [Renesse et al., 1994].

Par exemple, dans le cadre du projet SPIF[1] [Dupouy et al., 1997], nous avons développé un protocole basé sur une communication par liaison série entre une station de travail et un robot mobile. Les différentes stations de travail de l'application communiquaient par TCP, tandis qu'une station communiquait en utilisant une liaison série radio avec le robot mobile, comme récapitulé sur la figure 5.3 page ci-contre.

Ces protocoles sont enrichis par la possibilité de définir, de manière séparée, des filtres. Chaque filtre est un paquetage de la hiérarchie System.Garlic.Filters et possède des fonctions permettant de l'initialiser (négociation d'une clé entre partitions par exemple dans le cas d'un filtre de chiffrement ou d'authentification) et de filtrer et défiltrer des paquets de données opaques.

Les filtres, comme les protocoles, n'ont aucune connaissance directe du contenu des paquets qu'ils manipulent. Leur tâche est uniquement de les transmettre, filtrés ou non, à une partition distante selon la demande du cœur de l'exécutif. C'est pourquoi il est aisé, pour un utilisateur, d'ajouter de nouveaux protocoles de transports ou de nouveaux filtres sans connaître la structure interne de GLADE. Les filtres sont décrits dans la section 9.4 page 134.

5.5 La communication de groupe

Une application répartie en Ada doit maintenir deux tables distinctes de manière cohérente :

1. la table des unités contient l'état de tous les paquetages Remote_Call_Interface et l'identificateur de la partition sur laquelle chaque paquetage a été élaboré ;
2. la table des partitions contient la liste des adresses de chaque partition.

Ces deux tables constituent le "service de nommage" de l'application répartie. Elles permettent de localiser physiquement un service (paquetage Remote_Call_Interface) à partir de son nom symbolique.

[1] Système de Prototypage à Intégration rapide et Faible coût

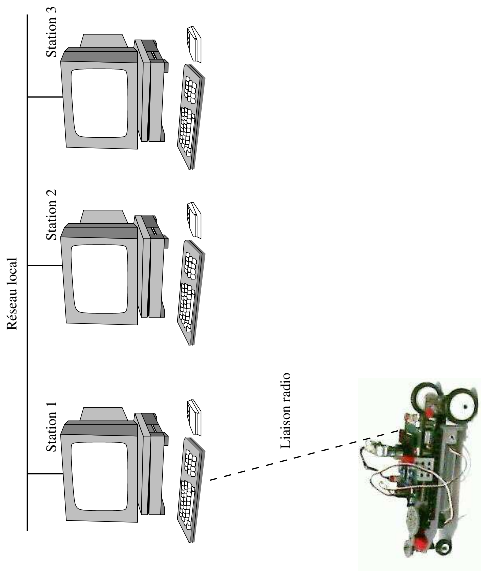


FIG. 5.3 – Communication avec un robot mobile

Dans les premières versions de GLADE, une partition dite **principale** et jouant un rôle privilégié, était indispensable au bon fonctionnement du système. La disparition de cette partition empêchait de nouvelles partitions de se joindre au système réparti et provoquait de plus la terminaison immédiate de l'ensemble des partitions de l'application répartie, car l'algorithme de négociation de la terminaison (voir chapitre 8) reposait entièrement sur la présence de cette partition. De plus, cette partition constituait l'unique serveur de noms de l'applications ; il n'était donc plus possible de déterminer sur quelle partition se trouvait un paquetage catégorisé Remote_Call_Interface lorsque cette partition avait disparu.

Nous avons, dans la version actuelle de GLADE, autorisé la présence de plusieurs **points d'accueil**. Un point d'accueil est une partition, participant à une application répartie, à laquelle de nouvelles partitions peuvent se connecter afin de se joindre au système. Les points d'accueil sont organisés logiquement sous forme d'anneau, comme présenté figure 5.4 page ci-contre. Une partition souhaitant se joindre à l'application répartie contacte un des points d'accueil et l'utilise ensuite pour obtenir les informations de nommage dont elle a besoin.

On notera que la représentation des nœuds logiques par des ordinateurs sur la figure 5.4 n'est que picturale. En effet, plusieurs nœuds peuvent s'exécuter sur la même machine.

Bien évidemment, les tables de nommages localisées sur les différents points d'accueil doivent être cohérentes et synchronisées. Rien n'interdit en effet que deux partitions contactent au même moment deux points d'accueil différents et tentent d'enregistrer des informations incompatibles. C'est alors qu'interviennent les protocoles de communication de groupe, décrits dans les sections 6.3 et suivantes, qui assurent l'intégrité des données et la résolution des conflits. Cette partie de l'exécutif est elle-même totalement indépendante des deux autres, au sens où une partie des données échangées est opaque, et où n'importe quel protocole de communication remplissant les conditions explicitées dans la section précédente peut être utilisé.

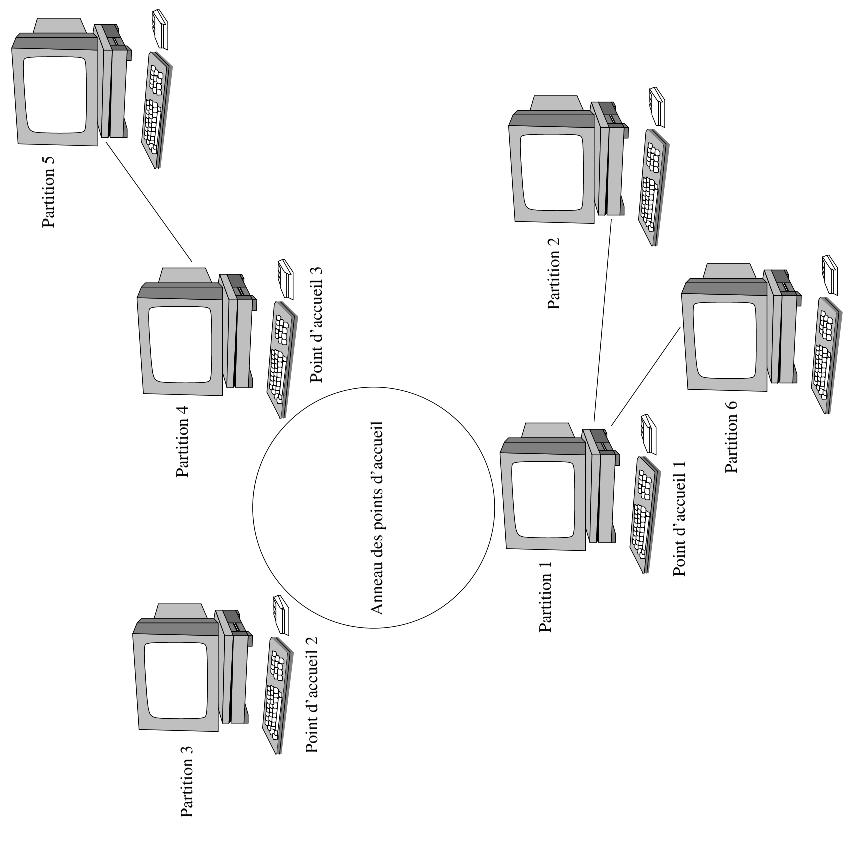


FIG. 5.4 – Points d'accueil multiples

6

Élaboration d'une partition

Dans tout système réparti, la localisation des différents services est une exigence incontournable. En effet, il n'est pas possible de communiquer avec un tiers sans savoir comment le contacter. Cette localisation peut être statique, lorsqu'un service se trouve en permanence à la même adresse, ou dynamique, lorsque la position du service n'est pas connue *a priori*.

6.1 Exposé du problème

Le mode de répartition adopté dans l'annexe des systèmes répartis d'Ada simplifie volontairement la tâche du programmeur : à l'exception des directives de compilation indiquant comment découper les paquetages, le code n'a pas à tenir compte de la répartition potentielle de l'application. Pour cette raison, l'annexe des systèmes répartis ne définit aucun service de localisation ; cette tâche est confiée à l'exécutif gérant les programmes répartis.

Un programme réparti écrit en Ada requiert l'utilisation successive de deux services de localisation. Tout d'abord, chaque paquetage catégorisé à l'aide de la directive de compilation Remote_Call_Interface se trouve sur une et une seule partition de l'application répartie, partition inconnue des clients cherchant à utiliser ce paquetage. Ensuite, chaque partition doit être localisée en tant que programme autonome possédant un ou plusieurs points de communication permettant de la contacter.

6.2 Architecture centralisée

Les premières versions de GLADE utilisent un serveur de nom centralisé unique, situé sur une partition appelée "partition principale". Cette partition a également la charge de lancer les autres partitions du système réparti lorsqu'il lui a été demandé de le faire. Cela permet de lancer une application répartie de la même manière qu'une application monolithique, en exécutant, du point de vue de l'utilisateur, un programme unique.

La partition principale possède systématiquement l'identificateur 1, et sera donc notée

P_1. Elle possède deux tables permettant de localiser des services[1] distants :

1. la table des partitions associe à chaque partition un ou plusieurs points de contact permettant de s'y connecter afin d'effectuer des requêtes, ainsi que des informations sur l'état de la partition ;
2. la table des unités fait correspondre à chaque paquetage catégorisé comme Remote_Call_Interface le numéro de partition sur laquelle le corps de ce paquetage a été élaborée, ainsi que la version de ce paquetage.

6.2.1 Connexion d'une nouvelle partition

Lorsqu'une nouvelle partition (notée $P_?$) souhaite participer à une application répartie, elle doit tout d'abord obtenir un numéro de partition et enregistrer la liste des paquetages appelables à distance dont elle a la charge.

Cette nouvelle partition se connecte à la partition principale, dont la localisation est indiquée soit dans le fichier de configuration de l'application, soit sur la ligne de commande de l'exécutable correspondant à la partition. Elle obtient alors son propre numéro de partition attribué par P_1, et devient donc la partition P_i, comme illustré sur la figure 6.1 page suivante.

Elle enregistre ensuite auprès de P_1 la liste des paquetages Remote_Call_Interface qu'elle possède ainsi que leur version. Cette phase est présentée figure 6.2 page ci-contre. Ensuite, comme indiqué sur la figure 6.3, P_i vérifie auprès de P_1 qu'elle est bien considérée comme gestionnaire des paquetages qu'elle a enregistrés. Dans le cas contraire, la partition termine avec l'exception Program_Error, car cela signifie qu'une autre partition a déjà déclaré gérer ces paquetages.

Une fois cette phase de connexion terminée, la partition P_i est prête à recevoir des requêtes entrantes à destination de ses paquetages appelables à distance. Elle ne possède toutefois aucune information sur les autres partitions du système.

6.2.2 Appel de sous-programme distant

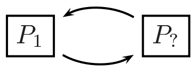
Lorsqu'une partition P_j souhaite faire un appel de sous-programme à distance, elle doit tout d'abord déterminer sur quelle partition P_i se trouve le paquetage contenant ce sous-programme. Si elle ne l'avait pas fait avant, elle envoie une requête à P_1 afin d'obtenir cette information, en indiquant le numéro de version souhaité du paquetage.

Tout d'abord, P_1 attend qu'une partition P_i déclare posséder ce paquetage. En effet, il est possible que P_i ne soit pas encore connecté à l'application répartie au moment de la demande de P_j. Pendant ce temps, la demande de P_j reste en suspens.

Une fois que P_i a fourni l'information nécessaire, les numéros de version enregistrés et demandés pour ce paquetage sont comparés. S'ils sont égaux, P_1 répond à P_j que le paquetage demandé se trouve sur la partition P_i, précédemment enregistrée. S'ils sont différents, l'application répartie est incohérente [RM, 1995, E.3(6)] et la partition P_j est considérée comme fautive, par choix de conception. Elle est donc déconnectée du système. La figure 6.4 page suivante représente un tel échange dans lequel les numéros de version concordent (situation normale).

[1] "service" est à prendre ici au sens large, et désigne aussi bien un paquetage Remote_Call_Interface qu'une partition.

1 – Je n'ai pas d'identificateur



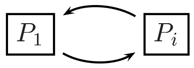
2 – Maintenant tu es P_i

FIG. 6.1 – Attribution d'un numéro de partition

Voici un paquetage Remote_Call_Interface (π, v)

FIG. 6.2 – Enregistrement des paquetages

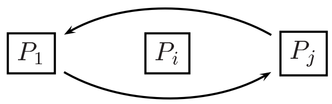
1 – Où se trouve le paquetage (π, v) ?



2 – π se trouve sur P_i

FIG. 6.3 – Vérification de la cohérence

1 – Je cherche (π, v)

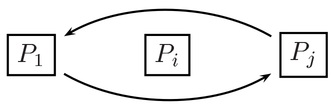


On suppose ici que P_j est déjà totalement élaborée

2 – π se trouve sur P_i

FIG. 6.4 – Localisation d'un paquetage

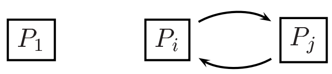
1 – Où se trouve P_i ?



2 – Voici l'adresse de P_i

FIG. 6.5 – Localisation d'une partition

2 – Voici le résultat



1 – J'appelle ce sous-programme

FIG. 6.6 – Appel direct de sous-programme distant

Une fois l'identité de la partition P_i obtenue, P_j cherche à la contacter afin d'effectuer l'appel de sous-programme à distance. Si la localisation physique de P_i est déjà connue par P_j, alors l'appel de sous-programme peut se faire normalement comme détaillé dans le chapitre 7 page 107. Dans le cas contraire, P_j demande à P_1 la liste des points de contact de P_i lors d'un dialogue tel que celui détaillé en figure 6.5 page précédente.

Une fois P_i localisé physiquement, P_j s'y connecte directement et effectue l'appel de sous-programme à distance. P_i enverra la réponse fournie par ce sous-programme directement à P_j, sans passer par P_1 (voir figure 6.6 page précédente).

Lorsque le code d'une partition comporte un appel statique à un sous-programme présent dans un paquetage Remote_Call_Interface, ce paquetage distant est localisé pendant la phase d'élaboration de la partition. Cela permet d'éliminer le délai inhérent à la localisation d'une unité lors de la tentative d'exécution d'un sous-programme distant dont on savait à l'avance qu'il pouvait être appelé. Dans le cas d'appels dynamiques, effectués à l'aide de pointeurs sur objets ou sous-programme distants, cette phase ne peut bien sûr pas avoir lieu durant l'élaboration et est retardée jusqu'au moment de l'appel.

On notera que le modèle de vérification d'élaboration propre à Ada, empêchant un paquetage d'appeler un sous-programme appartenant à un autre paquetage non encore élaboré, a été étendu aux partitions. Une partition ne peut pas traiter d'appel entrant tant que son élaboration n'est pas totalement terminée [RM, 1995, E.4 (14)]. Il est donc possible de créer des situations de blocage en répartissant une application monolithique.

6.2.3 Fonctionnement de la partition principale

Le fonctionnement de la partition principale est résumé par le réseau de Petri de la figure 6.7. La place Repos, ne jouant aucun rôle actif, symbolise le fait que les opérations se déroulent séquentiellement.

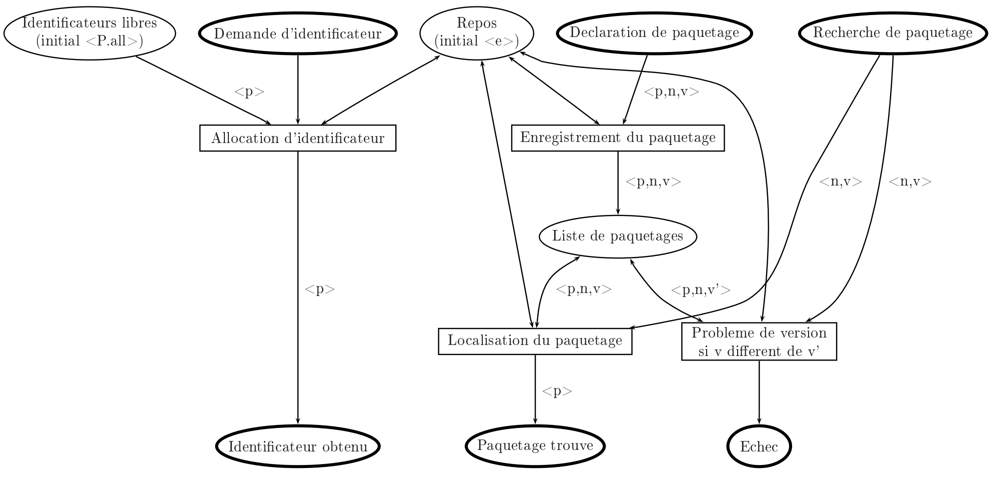


FIG. 6.7 – Partition principale

Comme on peut le voir sur ce schéma, la disparition de la partition principale d'une application répartie peut avoir des conséquences irrémédiables sur le fonctionnement du système : aucune nouvelle partition ne pourra se connecter à l'application répartie, et aucune partition ne sera plus en mesure de localiser une autre partition avec laquelle elle souhaite dialoguer.

6.3 Points d'accueil multiples

Les versions récentes de GLADE offrent au programmeur la possibilité d'utiliser plusieurs points d'accueil. Une partition souhaitant se joindre à une application répartie peut se connecter à n'importe quel point d'accueil, et non plus seulement à la partition principale. De plus, ces points d'accueil maintiennent des copies cohérentes des tables associant les paquetages Remote_Call_Interface à leur numéro de partition, et négocient entre eux les attributions de numéros de partitions afin que des connexions simultanées sur deux points d'accueil ne provoquent pas l'attribution du même identificateur.

Les différents points d'accueil d'une application répartie sont organisés en anneau. Cela signifie notamment que chaque point d'accueil a un successeur et un prédécesseur uniques. Dans le cas d'un point d'accueil unique, il est lui-même son propre successeur et prédécesseur. L'exemple de la figure 6.8 montre un anneau composé de $\{P_2, P_5, P_6, P_4\}$, ainsi que trois autres partitions, P_1, P_3 et P_7. On remarquera que ces partitions ne se sont pas toutes connectées au même point d'accueil.

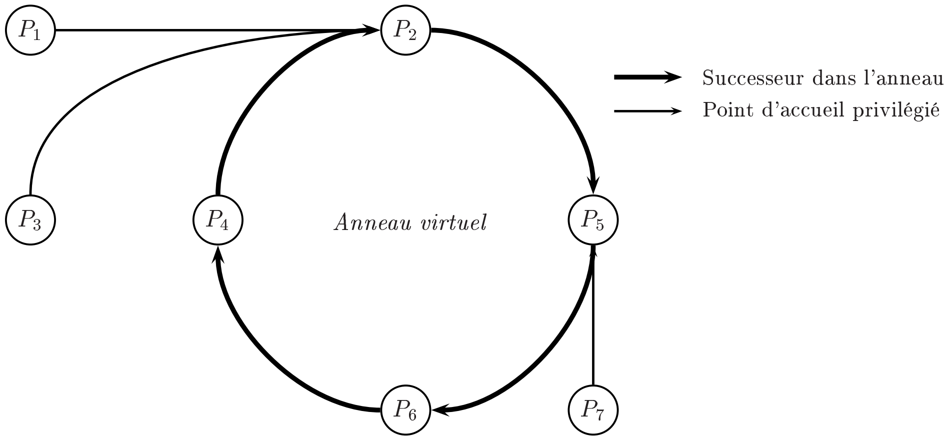


FIG. 6.8 – Anneau des points d'accueil

Un point d'accueil privilégié est associé à chaque partition P_i ne faisant pas partie des points d'accueil. C'est à lui que s'adressera P_i pour les requêtes relatives aux autres partitions du système, ainsi que pour toutes les demandes concernant les paquetages Remote_Call_Interface.

Lorsqu'un point d'accueil disparaît, de par une terminaison normale ou en cas de défaillance, toutes les partitions dont il était le point d'accueil privilégié en choisissent un nouveau et s'y connectent. C'est à lui qu'elles s'adresseront pour les futures demandes de renseignement. L'anneau est également reconfiguré en supprimant la partition manquante. Inversement, lorsqu'un nouveau point d'accueil apparaît dans le système, il est inséré dans l'anneau entre deux points d'accueil existants.

Le point d'accueil avec le plus petit identificateur (P_2 sur notre exemple) est celui qui a pour charge d'initier le calcul de l'état de terminaison globale. S'il disparaît, celui qui aura à cette date le plus petit identificateur (ici P_4) assumera ce rôle.

6.4 Connexion négociée d'une nouvelle partition

La phase de connexion d'une nouvelle partition est plus complexe qu'avec le modèle centralisé expliqué précédemment. Étant donné qu'on ne connaît pas *a priori* le nombre de partitions qui essayeront de se connecter sur chaque point d'accueil, il est impossible de donner aux points d'accueil une plage d'identificateurs dans laquelle ils peuvent attribuer de nouveaux numéros pour les partitions nouvellement créées.

Une négociation, sous la forme d'un jeton propagé sur l'anneau, a donc lieu systématiquement. Cette négociation assure deux propriétés :

1. Si l'identificateur de partition est accepté, il restera unique durant toute la vie de l'application répartie.
2. Si l'identificateur de partition envisagé est déjà utilisé, le point d'accueil se verra demander explicitement d'en allouer un autre.

6.4.1 Négociation d'un identificateur libre

La figure 6.9 page suivante représente l'algorithme utilisé sur un point d'accueil *self*. Les points d'entrée sont les place "Demande d'allocation", qui sert à attribuer un nouvel identificateur, et "Réception du prédecesseur", qui correspond à un message entrant en provenance de l'anneau. Les points de sortie sont la place "Identificateur alloué", utilisable lorsqu'un identificateur a été négocié, et la place "Envoi au successeur", qui correspond à l'envoi d'un jeton sur l'anneau.

Le principe de l'algorithme est le suivant : lorsqu'un point d'accueil souhaite allouer un nouvel identificateur (place "demande d'allocation"), il franchit tout d'abord la transition "verrouillage" qui consomme le jeton de la place "verrou". Cela garantit qu'un point d'accueil ne négocie qu'un identificateur à la fois. Un jeton sera replacé dans la place "verrou" une fois la négociation terminée.

Un identificateur est extrait de la liste des "identificateurs libres" lors du franchissement de la transition "allocation". Un nouveau jeton $\langle self, n\rangle$ est ajoutée à la place "associations identificateur↔partition", enregistrant le fait que ce point d'accueil *self* a demandé à réserver cet identificateur n. Un message indiquant cette demande d'association est envoyé au nœud suivant de l'anneau (place "envoi au successeur").

Lorsqu'un message avec l'association $\langle o, n\rangle$ arrive sur un point d'accueil *self*, on teste si o et *self* sont égaux. Si c'est le cas, le message a fait un tour complet de l'anneau et l'association n'a été rejetée par aucun des participants. On renvoie donc le numéro de partition n dans la place "identificateur alloué" en franchissant la transition "acceptation".

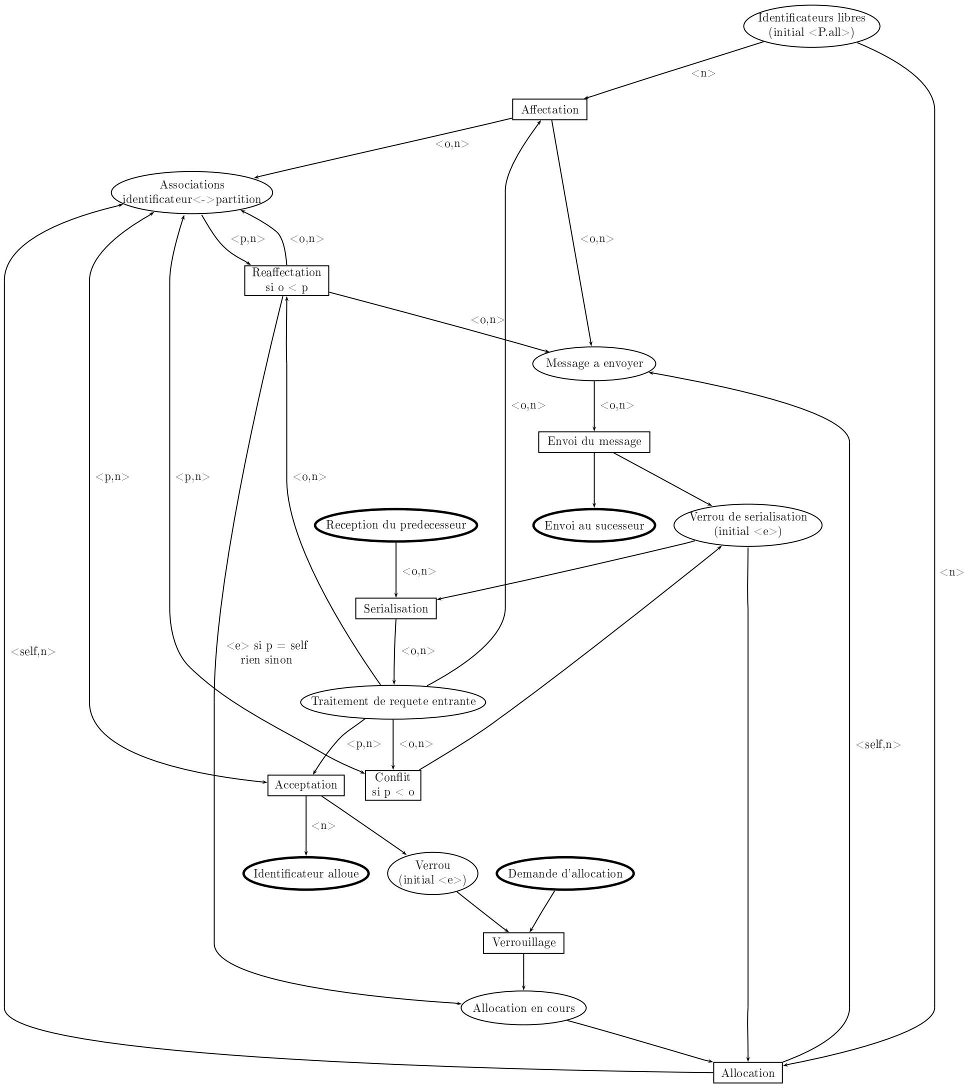


Fig. 6.9 – Négociation d'un identificateur de partition

Si o et *self* sont différents, on est en présence d'un message venant d'un point d'accueil cherchant à valider une association. Dans ce cas, si n n'était pas déjà associé à un point d'accueil, on enregistre l'association $\langle o, n\rangle$ avant de propager le message au nœud suivant de l'anneau.

Par contre, si n est déjà en cours d'association à une partition p, trois cas peuvent se présenter :

1. Si $p < o$, alors on accorde à l'association enregistrée la priorité par rapport à la nouvelle association proposée, et on oublie silencieusement le message entrant qu'on ne propage pas au suivant (transition "conflit"). Nous verrons plus tard que ce cas ne peut se produire que sur la partition p, qui a envoyé l'association $\langle p, n\rangle$ auparavant.
2. Si $p > o$ et $p = \mathit{self}$, alors un point d'accueil plus prioritaire cherche à utiliser un identificateur que nous cherchons également à utiliser. Étant donné que o est plus prioritaire, on sait que l'association $\langle \mathit{self}, n\rangle$ sera rejetée, et on alloue donc, par l'intermédiaire de la place "allocation en cours", un nouvel identificateur. On propage la nouvelle association ainsi que le message entrant.
3. Si $p > o$ et $p \neq \mathit{self}$, alors on conserve uniquement la nouvelle association $\langle o, n\rangle$ en oubliant la précédente. On envoie le message au nœud suivant de l'anneau.

Cet algorithme exploite une propriété des protocoles de communication de GLADE : la fiabilité et le non-déséquencement des messages (voir section 5.4 page 90). En effet, si un point d'accueil P_i demande l'association $\langle P_i, n\rangle$ et qu'un autre point d'accueil P_j plus prioritaire a besoin d'allouer également un nouveau numéro de partition, les cas suivants peuvent se produire :

1. P_j a déjà reçu l'association $\langle P_i, n\rangle$. Dans ce cas, n est marqué comme déjà utilisé et ne sera pas réutilisé.
2. P_j n'a pas encore reçu le message contenant l'association $\langle P_i, n\rangle$. Dans ce cas, P_j propage sur l'anneau l'association $\langle P_j, n\rangle$. Ceci entraîne trois conséquences :
   (a) P_i recevra l'association $\langle P_j, n\rangle$ avant le tour de complet de $\langle P_i, n\rangle$ (grâce aux propriétés des protocoles de communication rappelées ci-dessus). P_i sait qu'il est moins prioritaire et renvoie une demande $\langle P_i, n'\rangle$, tout en continuant à propager $\langle P_j, n\rangle$.
   (b) Lorsque P_j reçoit l'association $\langle P_i, n\rangle$, il s'aperçoit du conflit et cesse de propager cette association, sachant que P_i en aura généré une nouvelle.
   (c) Les partitions ayant reçu $\langle P_i, n\rangle$ et recevant $\langle P_j, n\rangle$ enregistrent la nouvelle association.

Notons qu'il est impossible qu'une partition autre que P_j ait reçu l'association $\langle P_j, n\rangle$ avant $\langle P_i, n\rangle$. En effet, en raison du non-déséquencement des messages, l'association $\langle P_i, n\rangle$ aurait nécessairement été détruite par le nœud précédent qui aurait remarqué le conflit. De plus, la place "verrou de sérialisation" garantit que si deux messages doivent traverser un point d'accueil, leur ordre de sortie sera le même que leur ordre d'entrée.

6.4.2 Critique de l'algorithme

Cet algorithme, qui peut s'exécuter en un seul tour d'anneau en cas de conditions favorables, a toutefois un défaut : il ne garantit pas qu'un point d'accueil pourra négocier le choix d'un identificateur en un temps fini. En effet, si ce point d'accueil n'est pas le plus prioritaire, il est possible que toutes ses tentatives se soldent par un échec si des points d'accueil plus prioritaire cherchent à allouer de nouveaux identificateurs de manière continue, créant ainsi une situation de famine.

Cependant, ce type de système réparti dans lequel de nouveaux nœuds sont en permanence en cours de création (correspondant à une saturation des connexions) ne correspond pas au modèle auquel nous nous intéressons. Nous avons donc choisi d'ignorer ce cas pathologique.

6.5 Gestion des unités

La gestion des paquetages catégorisés comme Remote_Call_Interface est rendue plus complexe par l'utilisation d'un ensemble de points d'accueil. En effet, deux points sont à considérer :

1. Chaque paquetage Remote_Call_Interface doit être enregistré auprès de tous les points d'accueil.
2. Un paquetage Remote_Call_Interface ne doit être offert que par une seule partition.

Les sections suivantes décrivent comment ces mécanismes ont été mis en œuvre : tout d'abord, chaque paquetage est préenregistré, puis cet enregistrement est confirmé par la suite.

6.5.1 Préenregistrement d'un paquetage

Le modèle adopté est le suivant : lorsqu'une partition P_i essaye d'enregistrer un paquetage v appelable à distance auprès de son point d'accueil P_j, celui-ci consulte tout d'abord sa table de paquetages et vérifie qu'un paquetage du même nom n'a pas encore été enregistré. Si un autre paquetage du même nom existe, la demande d'enregistrement est rejetée et la partition P_i est déconnectée du système.

Si le paquetage est nouveau, un jeton portant le nom du paquetage est envoyé sur l'anneau des points d'accueil. Tout point d'accueil qui voit passer un jeton nommant un paquetage de P_i déjà déclaré sur une partition P_k ne conserve que les informations relatives à la partition de plus petit identificateur. Celle des partitions P_i et P_k ayant le plus grand identificateur est considérée comme déconnectée du système, et tous les paquetages qu'elle a enregistrés précédemment sont considérés comme inatteignables. Cette situation peut survenir lorsque deux paquetages identiques sont déclarés par deux partitions différentes P_i et P_k utilisant deux points d'accueil différents.

Lorsqu'une partition voit passer un jeton portant le nom d'un paquetage déjà enregistré mais avec un identificateur supérieur à celui déjà présent dans la table, elle arrête la propagation du jeton. En effet, le point d'accueil ayant proposé ce paquetage verra ou a déjà vu passer le message correspondant au point d'accueil prioritaire, grâce aux propriétés de non déséquencement des protocoles de communication de GLADE, et sait que sa proposition de paquetage a déjà été refusée par certains nœuds.

La figure 6.10 page suivante présente un exemple d'un tel échange. Sur cette figure, l'anneau des points d'accueils (représenté par un cercle haché et grisé) est composé de P_1, P_2, P_3 et P_4, qui se succèdent dans cet ordre. La partition P_6 annonce à son point d'accueil P_3 qu'elle possède le paquetage RCI. P_5 annonce le même paquetage à son point d'accueil P_1 au même moment.

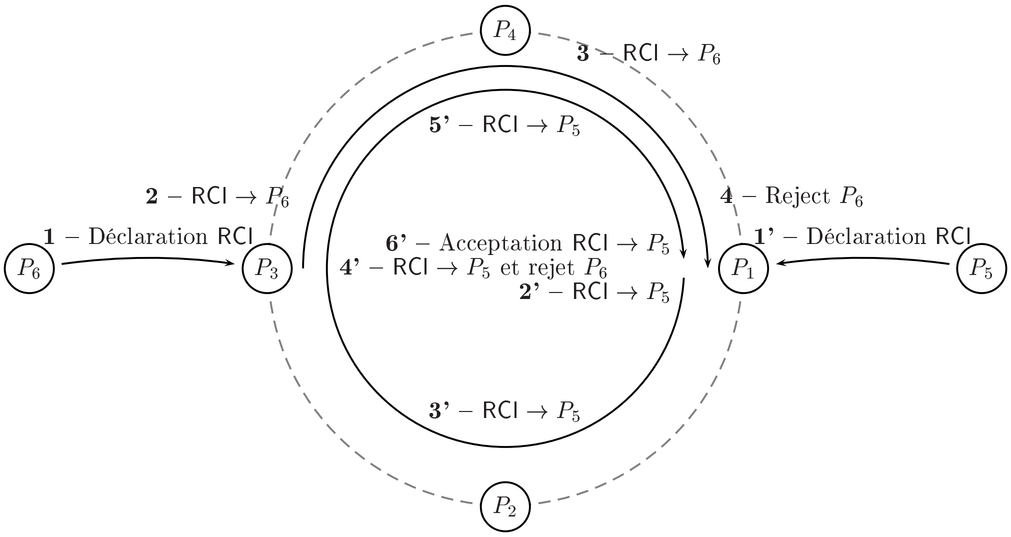


FIG. 6.10 – Résolution de conflit d'enregistrement de paquetages

Six étapes ont alors lieu :

1. P_6 signale à P_3 qu'elle possède le paquetage RCI, et P_5 fait de même avec P_1.
2. P_3 accepte la déclaration de P_6, et P_1 celle de P_5. P_3 et P_1 propagent toutes deux un jeton contenant les déclarations respectives.
3. Le jeton envoyé par P_3 parvient à P_4, qui accepte l'annonce de P_6. Celui de P_1 arrive sur P_2, qui fait de même avec l'annonce de P_5.
4. L'annonce de P_6 parvient à P_1, qui rejette cette annonce et ne retransmet pas le jeton. En effet, P_1 sait que P_5 est prioritaire par rapport à P_6, car elle possède un identificateur de partition plus petit. Au même moment, P_3 reçoit l'annonce de P_5, et sait qu'elle doit abandonner celle de P_6, pour la même raison. Elle envoie donc une exception Program_Error à P_6 lui indiquant que le paquetage RCI existe déjà dans l'application répartie.
5. P_4 reçoit l'annonce de P_5, et écrase celle qu'il avait reçue de P_6, P_5 étant plus prioritaire.
6. P_1 reçoit l'annonce de P_5, qu'elle a elle-même envoyée à l'origine. L'annonce a déjà fait un premier tour d'anneau, et va en faire un second, comme expliqué dans la section suivante.

6.5.2 Validation de l'enregistrement d'un paquetage

Un point d'accueil recevant un jeton qu'il a lui-même envoyé sait qu'un consensus a été dégagé et que le paquetage qu'il propose a été accepté. Il envoie donc un second jeton afin de signaler l'existence du consensus ; le paquetage est définitivement enregistré dans les tables gérant les unités de tous les points d'accueil.

Cette deuxième phase est rendue nécessaire par le fait qu'une partition dont un paquetage est temporairement accepté (c'est-à-dire une partition annonçant un paquetage sans que le jeton d'annonce ait fini son tour d'anneau) ne doit pas accepter de requêtes entrantes pour ce paquetage avant d'être assurée qu'il n'y aura aucun conflit. Au premier tour d'anneau, l'état de cette unité est marqué comme Declared. Au second, il est validé et est marqué comme Defined et peut répondre aux requêtes.

Notons que cette deuxième phase n'est pas nécessaire lors de l'allocation d'un identificateur de partition. En effet, le modèle d'identificateurs choisi étant totalement dynamique, il n'est pas possible qu'une partition soit référencée avant d'avoir enregistré des unités ou avant d'avoir elle-même fait un appel de sous-programme distant. Dans ces deux cas, le consensus sur l'identificateur de partition a déjà été accepté par l'ensemble des points d'accueil.

6.5.3 Cycle de vie d'une unité

Les états Declared et Defined ne sont pas les seuls états possibles d'une unité dans la table des unités. La figure 6.11 représente ces différents états ainsi que les transitions permettant de passer de l'un à l'autre.

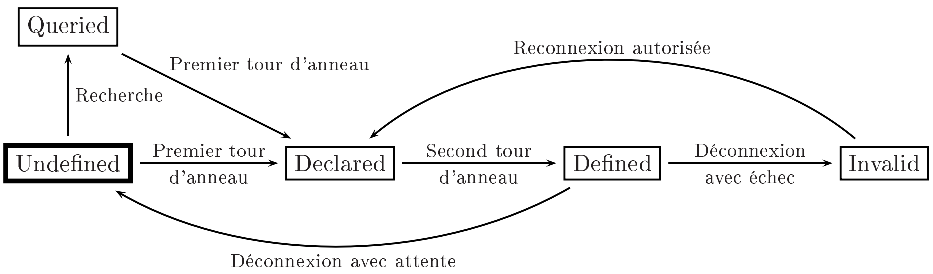


FIG. 6.11 – Cycle de vie d'une unité

Au départ, toutes les unités se trouvent dans l'état Undefined. Lorsqu'une partition liée à un point d'accueil P_i demande des informations sur une unité dans l'état Undefined (c'est le cas des unités inconnues de la table des unités), le statut de ce paquetage passe localement (sur P_i) à l'état Queried, indiquant qu'au moins une partition cherche à joindre cette unité.

Les deux tours d'anneau décrits ci-dessus font passer l'état du paquetage à Declared, dès que l'on sait qu'une partition au moins le propose, puis à Defined, lorsqu'un consensus a été dégagé sur la partition autorisée à exporter cette unité. Dans une application répartie sans panne, toutes les unités resteront dans cet état une fois qu'il est atteint.

En cas de déconnexion d'une partition, le comportement change selon la politique de reconnexion choisie. Une application rigide de l'annexe des systèmes répartis impose que l'ensemble des unités exportées par cette partition devienne indisponible, ce qui correspond sur le schéma à l'état Invalid. Toutefois, les politiques de reconnexion que nous avons ajoutées dans GLADE et décrites dans la section 9.3 page 133 permettent de décrire plus finement le comportement à adopter.

Lorsque la politique Fail_Until_Restart a été choisie, chaque unité se trouvant dans l'étant Invalid y reste jusqu'à ce qu'une nouvelle partition déclare exporter cette unité. À ce moment là, l'unité revient dans l'état Declared, et les nouvelles requêtes seront satisfaites lorsqu'un consensus est de nouveau atteint (matérialisé par l'état Defined).

Si la politique Wait_Until_Restart a été choisie, alors l'état de l'unité ne passe plus à Invalid mais à Undefined, ce qui fait que les requêtes vers ce paquetage seront bloquées jusqu'à la nouvelle obtention de l'état Defined.

6.6 Conclusion

Le passage, entre les premières versions de GLADE et la version actuelle, d'une politique centralisée à une politique répartie nous a permis de supprimer le point de panne unique qui rendait toute résistance aux défaillances irréalisable. L'utilisation d'une politique de reconnexion des unités permet de redémarrer à froid[2] des paquetages mourant en cours d'exécution de l'application répartie.

[2]Le redémarrage à froid est un redémarrage dans lequel l'état des unités remplacées n'est pas reconstitué automatiquement.

7

Appel de sous-programme à distance

Les mécanismes présentés ici considèrent que les étapes de localisation d'un sous-programme distant et de connexion à la partition sur laquelle ce sous-programme a été élaboré ont déjà été effectuées. Ces étapes sont présentées en détail dans le chapitre 6 page 95.

7.1 Invocation de sous-programme distant

7.1.1 Requête bidirectionnelle

L'envoi d'une requête représentant un appel de procédure à distance bidirectionnel[1] nécessite l'utilisation d'un identificateur permettant de marquer de manière non ambiguë la réponse correspondant à cette requête parmi l'ensemble des réponses reçues. En effet, dans le cadre d'une exécution concurrente, il est possible pour une partition d'effectuer simultanément plusieurs requêtes vers une autre partition, et les résultats peuvent parvenir dans un ordre différent de celui des requêtes.

La figure 7.1 page suivante modélise, sous la forme d'un réseau de Petri coloré, l'enchaînement des opérations se déroulant lors d'un tel appel, représentation qui sera raffinée dans la section suivante. Ce schéma correspond à l'implémentation que nous avions réalisée lorsque le compilateur GNAT ne disposait pas encore des fonctions d'avortement, mais reste utile pour poser les bases de ce mécanisme.

Les notations utilisées sont :
- t : tâche qui lance la requête ;
- q : contenu de la requête ;
- r : réponse ;
- id : identificateur d'une requête.

Dans le cadre d'une invocation à distance qui parvient à son terme, la succession des

[1] C'est-à-dire une procédure sur laquelle la directive de compilation Asynchronous n'a pas été appliquée, voir section 2.2.5 page 47.

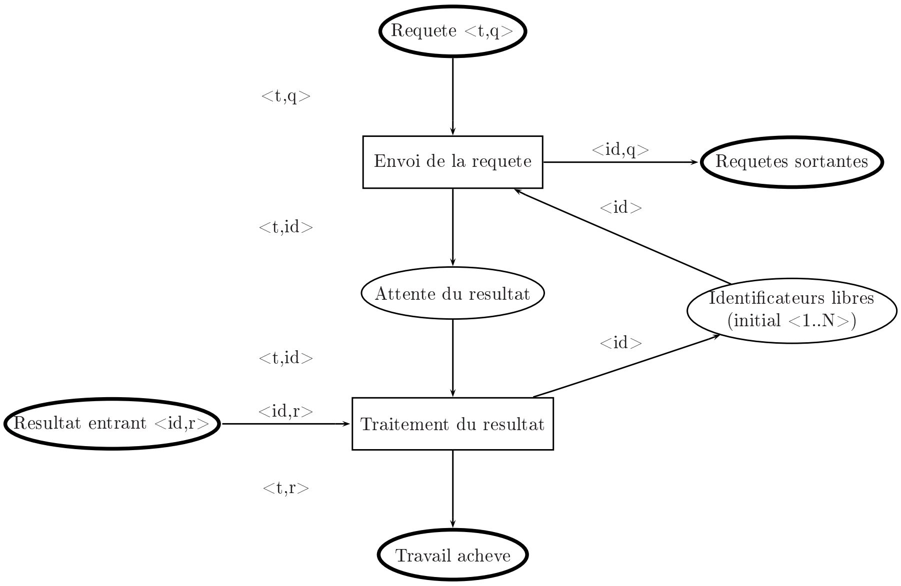


FIG. 7.1 – Émission d'une requête distante sans avortement

opérations se résume à :

1. l'envoi de la requête et d'un identificateur associé choisi parmi les identificateurs libres (transition "*Envoi de la requête*") ;

2. l'attente du résultat correspondant à cet identificateur (place "*Attente du résultat*") ;

3. le recyclage de l'identificateur pour un usage ultérieur et retour du résultat (transition "*Traitement du résultat*" et place "*Travail achevé*").

Le premier point d'entrée de ce réseau est la place "*Requete*", permettant de déposer la requête et son identificateur. La place "*Résultat entrant*" est utilisée par l'exécutif de GLADE pour faire parvenir les réponses. Chaque réponse est ensuite appariée avec la requête de même identificateur, identificateur qui se trouvera ensuite recyclé dans la place "*Identificateurs libres*".

La fin de l'appel de sous-programme à distance est représenté par la place "*Travail achevé*", où l'on trouve le résultat de la requête. Rappelons que, par le terme générique "résultat", on désigne à la fois le résultat d'une fonction, les paramètres de sortie d'une procédure ou une éventuelle exception ; cette exception peut par ailleurs avoir été générée soit par le sous-programme distant dans le cadre de son exécution, soit par la couche de communication pour signaler une faute sous la forme de l'exception Communication_Error.

La place "*Requêtes sortantes*" représente un tampon de communication du réseau : tout ce qui est placé ici sera envoyé en un temps fini au destinataire.

7.1.2 Avortement d'une requête

À l'exception des situations décrites dans [RM, 1995, 9.8] (par exemple lors de l'exécution d'un sous-programme d'un objet protégé), il est possible d'avorter une tâche Ada, c'est-à-dire de placer cette tâche dans l'état `abnormal` et d'arrêter son exécution. Si une tâche est avortée pendant un appel de sous-programme à distance, le paragraphe [RM, 1995, E.5(22)] précise qu'un message demandant l'avortement du sous-programme distant doit être envoyé, tandis que le paragraphe [RM, 1995, E.4(13)] indique que l'avortement effectif n'est pas obligatoire. Pour des raisons de symétrie avec les programmes monolithiques, nous avons choisi de traiter ce cas dans sa totalité, honorant ainsi les deux clauses citées.

Le réseau de Petri de la figure 7.2 présente une vue d'ensemble de la situation. Les transitions grisées représentent des transitions hiérarchiques, qui sont explicitées figure 7.3 page suivante. Trois processus apparaissent clairement : celui de gauche correspond à une demande d'appel distant. Il se divise en deux, l'appel lui-même et l'attente du résultat. Le réseau du milieu correspond au processus de traitement du résultat. Celui de droite à l'avortement d'une requête distante. À cet effet, une place d'entrée "*abort job*" a été ajoutée à usage exclusif de l'exécutif Ada ; cette place devient marquée lorsqu'une demande d'avortement a lieu sur une tâche qui est en train d'effectuer un appel de sous-programme à distance bidirectionnel.

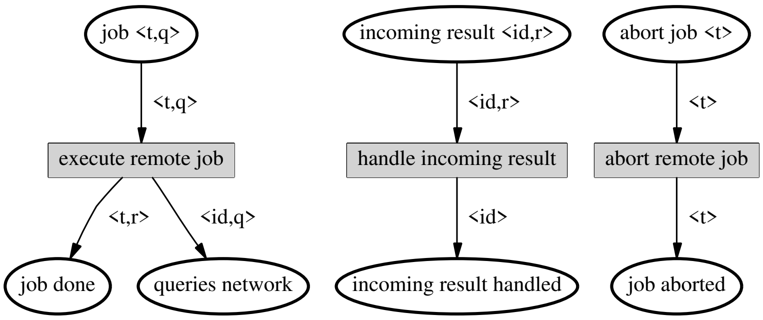


FIG. 7.2 – Représentation hiérarchisée d'une requête distante

On notera sur la figure 7.3 page suivante la présence d'un groupe nommé "*abortable part*" ; ce groupe dénote l'état pendant lequel la place "*abort job*" peut être rendue active par l'exécutif Ada. En effet, que ce soit après que la transition "*handle result*" a été franchie ou avant que la transition "*send job request*" le soit, la tâche gérant l'appel de sous-programme à distance est placée dans un état non avortable par l'utilisation de la procédure de finalisation d'un objet contrôlé. En effet, lorsqu'un objet contrôlé est détruit, sa procédure de finalisation est exécutée quelle que soit la façon dont la destruction intervient [RM, 1995, 9.8(10)], donc y compris en cas d'avortement de la tâche qui le définit.

Trois nouvelles transitions et deux places de sortie ont été ajoutées par rapport au réseau de Petri de la figure 7.1 page ci-contre. Ces ajouts correspondent au traitement des cas suivants :

1. Lorsqu'une requête en attente de résultat est avortée, un message d'avortement

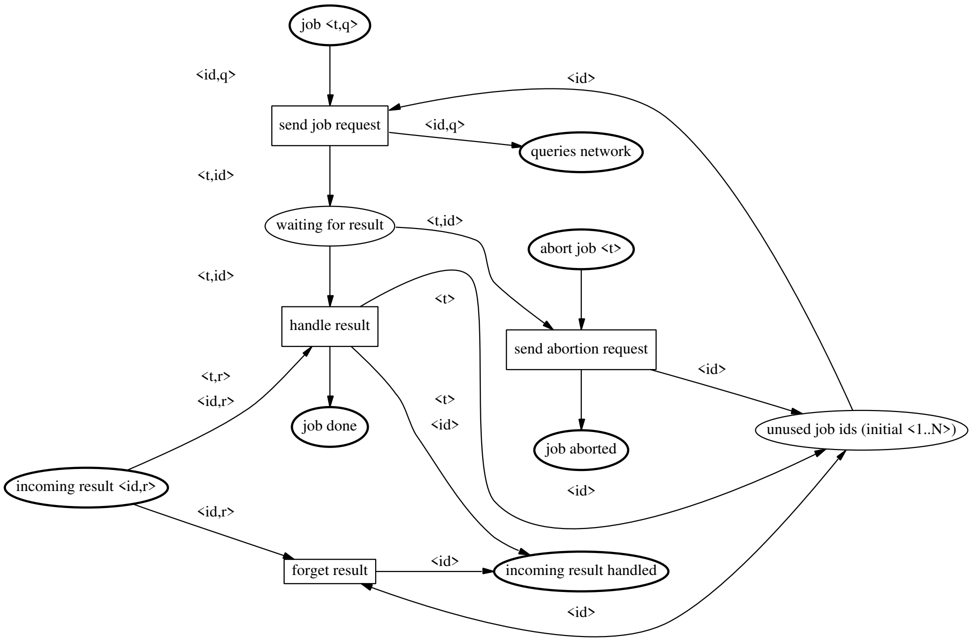


FIG. 7.3 – Émission d'une requête distante (décomposition)

est envoyé à la partition distante par le biais de la transition "*send abortion request*". L'identificateur est alors rendu de nouveau disponible pour une utilisation future.

2. Lorsqu'un résultat arrive et ne correspond à aucune requête en attente, ce résultat est ignoré. Ce cas de figure survient lorsqu'une demande d'avortement arrive trop tard sur la partition distante et croise l'envoi du résultat.

7.1.3 Analyse de la requête

Le réseau de la figure 7.3 permet d'identifier certains problèmes liés au modèle choisi.

Pénurie d'identificateurs

La liste des identicateurs disponibles est de taille finie et représentée par la place "*unused job ids*". De plus, les places "*unused job ids*" et "*waiting for result*" constituent un invariant [Jensen, 1981] [Memmi et Vautherin, 1991], en cela qu'à elles deux elles contiennent l'intégralité des N identificateurs.

Si p_{unused} et $p_{waiting}$ représentent respectivement les places "*unused job ids*" et "*waiting for result*", on a :

$$\forall M \in Acc(R, M_0), M(p_{unused}) \cup M(p_{waiting}) \quad = \quad \{\langle 1 \rangle, \ldots, \langle N \rangle\} \qquad (7.1)$$

Il est donc possible de faire N requêtes simultanément d'une partition P vers d'autres partitions du système. Lorsque N requêtes sont déjà en cours d'exécution, toute nouvelle tentative de requête se verra différée jusqu'à ce qu'un identificateur soit de nouveau disponible.

Conflit d'identification

Supposons que N requêtes soient lancées simultanément depuis une partition P et que la requête 1 soit avortée. Supposons maintenant que le message d'avortement parvienne trop tard à la partition qui traitait cette requête, et que le résultat soit tout de même envoyé. Supposons pour finir qu'une nouvelle requête ait lieu depuis la partition P (qui prendra donc le seul identificateur libre, à savoir 1), et que le résultat non désiré arrive à ce moment. Un simple examen du réseau de la figure 7.3 page ci-contre montre que cette réponse sera, de manière incorrecte, associée à la nouvelle requête, provoquant le passage de la partition P dans un état erroné.

Pour éviter ce problème et celui décrit précédemment, nous avons choisi de prendre N suffisamment grand (256 requêtes semble être une limite raisonnable comme nombre maximal de requêtes simultanées entre deux partitions du système). De même, les identificateurs libres sont stockés dans un tampon circulaire, ce qui permet de réutiliser chaque identificateur après un temps d'attente maximal. Toutes les applications que nous avons examinées se satisfaisaient très largement de ce choix (voir aussi section 12.5 page 168).

Défaillance silencieuse

La figure 7.3 page ci-contre montre qu'il existe deux moyens pour débloquer un appel de sous-programme à distance :

1. la réception de la réponse associée à la requête ;
2. l'avortement effectué par l'exécutif Ada.

Cependant, il est possible qu'une partition distante disparaisse ou devienne inatteignable sans qu'une faute puisse être détectée ; en effet, les connexions TCP ne sont signalées comme fermées qu'à l'initiative d'un des participants [Feit, 1993]. Si un nœud physique disparaît brutalement, tous les nœuds logiques qu'il porte seront silencieusement inatteignables. De plus, le temps d'exécution d'un sous-programme distant est impossible à connaître *a priori* : il n'est donc pas possible pour l'exécutif de GLADE d'utiliser un chien de garde qui détecterait une expiration du temps maximum accordé. Pour cette raison, tout programme robuste devrait contenir des chiens de garde adaptés autour de chaque appel de sous-programme distant, afin de déclencher l'utilisation de la place "*abort job*" lorsqu'un délai spécifique, choisi par le développeur, arrive à expiration. Un exemple est donné dans l'exemple 7.1 page suivante, dans lequel l'appel de procédure distant est avorté s'il excède une minute.

Nous envisageons également de mettre en place un autre mécanisme dans l'exécutif de GLADE : l'utilisateur pourra, au moyen d'une directive de compilation dans le fichier de configuration de son application (voir section 10.4.2 page 146), spécifier un temps maximum T qu'il estime ne jamais devoir être atteint pour faire un aller-retour entre deux partitions. À chaque fois qu'une partition P_1 aura des requêtes en cours dirigées vers une partition P_2, des messages de vérification réguliers seront envoyés de P_1 vers P_2 demandant une réponse immédiate. Si aucune réponse n'est reçue dans un temps

Exemple 7.1 Utilisation d'un chien de garde au niveau utilisateur

```
Cancelled : Boolean := False;
[...]
select
   delay 60.0;
   Cancelled := True;
then abort
   Remote_Procedure_Call;
end select;
if Cancelled then
   --  L'appel à distance ne s'est pas
   --  terminé dans le délai spécifié.
   [...];
end if;
```

T, la communication avec la partition P_2 sera considérée par la partition P_1 comme défectueuse et l'exception Communication_Error [RM, 1995, E.5(5)] sera renvoyée à l'utilisateur afin de débloquer les appels en attente. Bien entendu, ce mécanisme sera adapté afin d'échanger un minimum de messages entre deux partitions (notamment dans le cas où P_2 effectue également des requêtes vers P_1).

7.1.4 Requête unidirectionnelle

Une requête unidirectionnelle, c'est-à-dire un appel à une procédure distante sur laquelle la directive de compilation Asynchronous est appliquée, s'effectue de manière plus simple qu'une requête bidirectionnelle. En effet, aucun résultat n'étant attendu, il n'est pas nécessaire d'allouer un identificateur permettant d'apparier la requête et sa réponse. Une valeur spécifique mais invalide d'identificateur (c'est-à-dire en dehors de l'intervalle $[1, N]$) est utilisée pour indiquer au serveur que l'appel est unidirectionel. Comme spécifié dans [RM, 1995, E.4(15)], toute exception différente de Communication_Error est donc perdue.

L'appelant continue alors son exécution sans attendre la fin de l'exécution de la procédure distante, comme représenté sur la figure 7.4 page suivante. On a donc dans ce cas une génération implicite de parallélisme, car le code de l'appelant et le code de l'appelé s'exécutent en parallèle.

7.2 Traitement d'une requête entrante

La partie permettant de recevoir des requêtes entrantes est plus complexe que celle chargée d'émettre des requêtes sortantes. Le réseau de Petri coloré de la figure 7.5 page 114 représente une vision hiérarchisée du traitement d'une requête entrante, tandis que celui des figures 7.6 page 115 et 7.7 page 116 montre le fonctionnement interne du réservoir de tâches décrit ci-dessous et du flux d'exécution.

Les multiples occurrences des places grisées "*task pool*" et "*task count*" représentent deux places communes, mais ont été séparées en plusieurs copies afin de permettre une présentation plus aérée du réseau. La place "*task pool*" représente le réservoir de tâches,

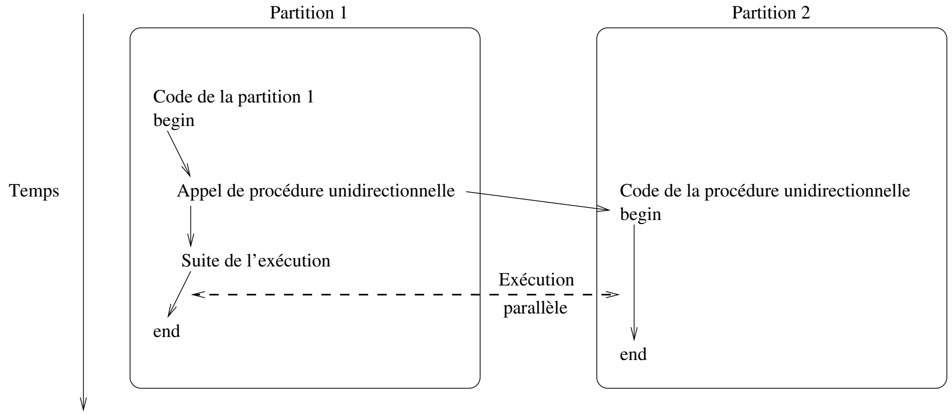


FIG. 7.4 – Génération implicite de parallélisme

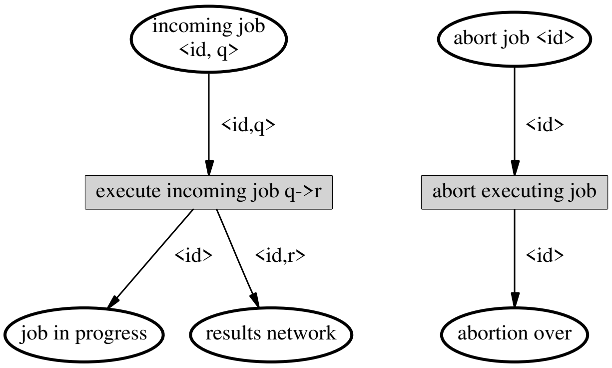


FIG. 7.5 – Traitement hiérarchique d'une requête entrante

tandis que la place "*task count*" contient un unique couple $(Total, Free)$ représentant le nombre de tâches anonymes[2] existant dans la partition ainsi que le nombre de tâches anonymes disponibles.

On notera que, dans l'implémentation réelle, les identificateurs de requêtes sont composés de l'identificateur entrant ainsi que du numéro de partition. Cela signifie que le résultat de deux requêtes de même identificateur provenant de deux partitions différentes ne verront pas leurs résultats mélangés.

7.2.1 Réservoir de tâches

Dans le cadre d'un appel à un sous-programme local, le code du sous-programme est exécuté dans le contexte de la tâche appelante. Pour un appel de sous-programme distant, cela est impossible et un tel sous-programme doit être exécuté dans le contexte d'une tâche locale à la partition cible.

L'utilisation de la tâche réceptionnant la requête sur le canal de communication simplifierait grandement le traitement des requêtes entrantes ; mais cela limiterait les appels concurrents d'une partition P_1 vers une partition P_2 à un seul. Pour cette raison, chaque requête entrante se voit liée dynamiquement à une tâche anonyme, dans le contexte de laquelle le code du sous-programme va se dérouler. Cette tâche anonyme aura également la charge de renvoyer le résultat de l'appel de sous-programme distant vers l'émetteur.

La première version de GLADE allouait les tâches au rythme des requêtes entrantes. Chaque nouvelle tâche traitait une et une seule requête, puis envoyait le résultat et se terminait. Cependant, des mesures de performance ont indiqué que cette méthode est coûteuse en terme de ressources et offre des temps de service bien supérieurs à ce qui peut être espéré dans le cadre d'appels à de petits sous-programmes distants. Pour cette raison, nous avons décidé de créer un réservoir de tâches, contenant un ensemble de tâches anonymes prêtes à être utilisées par les requêtes entrantes.

[2]Une **tâche anonyme** est une tâche dont le rôle est d'exécuter une requête entrante. Elle est anonyme car elle peut être chargée d'exécuter n'importe quel sous-programme.

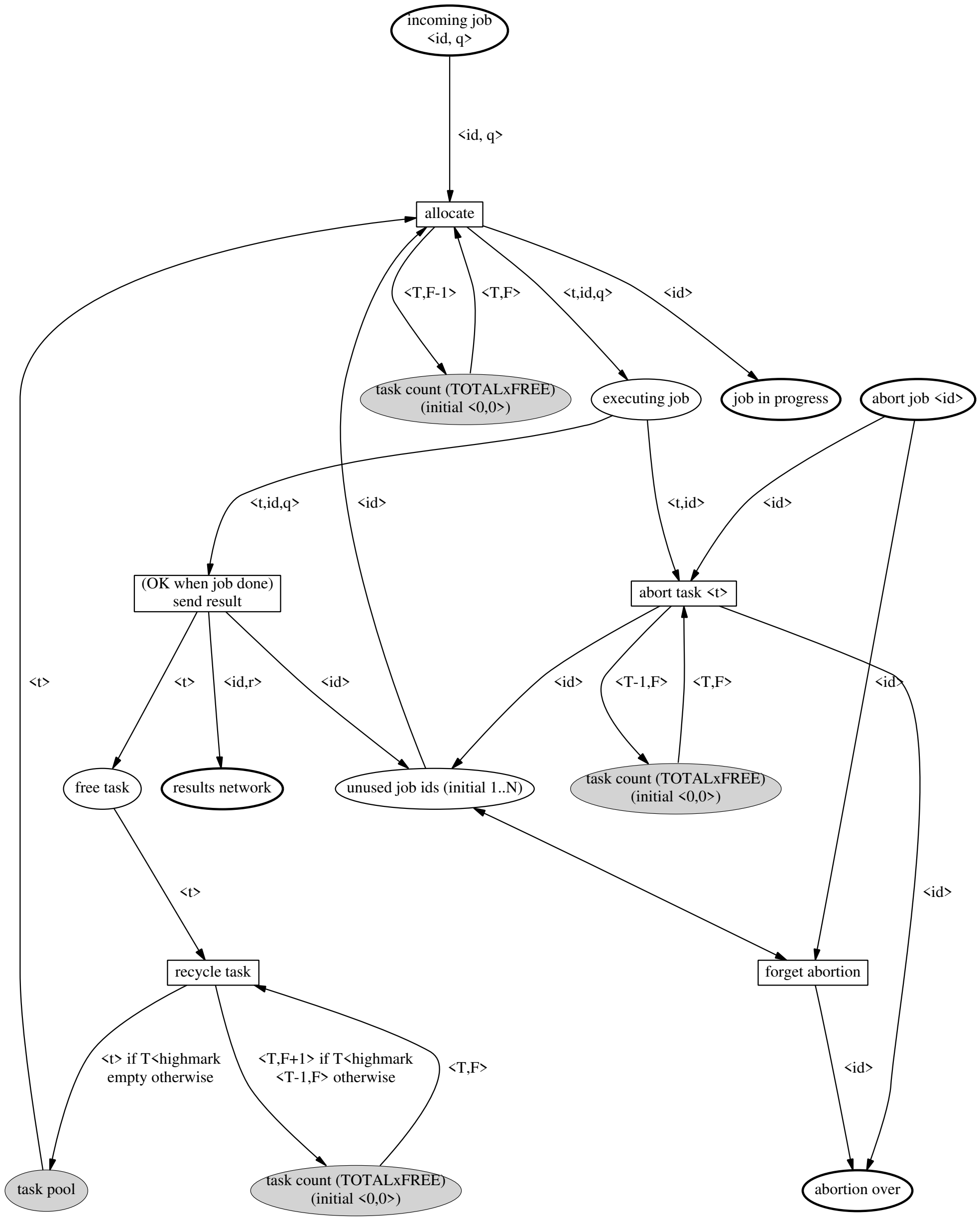


FIG. 7.6 – Traitement d'une requête entrante (partie 1)

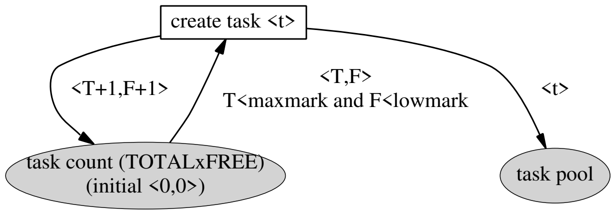


FIG. 7.7 – Traitement d'une requête entrante (partie 2)

Ce réservoir est paramétré par trois quantités numériques :

lowmark représente le nombre minimal de tâches dont on souhaite la disponibilité pour traiter les requêtes entrantes à un moment donné.

highmark représente le nombre de tâches libres à partir duquel on ne souhaite pas réutiliser les tâches en fin de cycle. Les tâches anonymes en excédent seront détruites.

maxmark représente le nombre de tâches anonymes qu'on ne souhaite pas dépasser dans le système.

Un bon dimensionnement de ces quantités améliore considérablement les performances d'un serveur. Si $lowmark$ est trop faible, une requête entrante risquera d'attendre plus longtemps la disponibilité d'une tâche anonyme ; à l'inverse, une valeur trop grande augmentera le nombre de tâches existant à un moment donné dans la partition, causant à l'ordonnanceur[3] une plus grosse charge de travail. Si $maxmark$ est positionnée à la valeur n, alors seules n requêtes entrantes pourront être servies simultanément, les autres devant attendre la fin de l'exécution des requêtes précédentes pour avoir une chance d'être exécutées.

Une tâche s'exécutant à la priorité la plus basse disponible se charge de maintenir le réservoir à un niveau convenable. Cette tâche correspond à la figure 7.7.

Un examen de la place "*task pool*" permet de faire les remarques suivantes :

1. Le réservoir est rempli par la tâche de plus basse de priorité de manière asynchrone. Cela signifie que si le réservoir est vide et que d'autres tâches effectuent des calculs intensifs, il est possible que la tâche de création ne soit jamais élue, créant ainsi une pénurie temporaire de tâche anonymes. Cependant, il ne serait pas raisonnable d'augmenter la priorité de la tâche de création : celle-ci a pour but d'augmenter l'efficacité de l'application répartie et ne doit en aucun cas bloquer une tâche appartenant à l'application elle-même.
2. Une tâche ayant fini de traiter sa requête et ayant envoyé le résultat va se remettre d'elle-même dans le réservoir (transition "*recycle task*").
3. Étant donné que seule la transition "*create task*" peut créer de nouvelles tâches et qu'elle n'est traversable que lorsque le nombre total de tâches est inférieur à $maxmark$, il est clair qu'il n'y aura jamais plus de $maxmark$ tâches anonymes en service à un moment donné.

[3] L'**ordonnanceur** est le module logiciel qui gère l'accès aux processeurs des différentes tâches d'une application Ada.

7.2.2 Chemin d'une requête ordinaire

Chaque requête entrante est déposée avec son identificateur dans la place "*incoming job*" par l'exécutif GLADE. Dès qu'une tâche anonyme t est disponible pour traiter la requête, l'exécutif peut continuer son exécution depuis la place "*job in progress*". La tâche anonyme se trouve dans l'état modélisé par la place "*executing job*", puis franchit la transition "*send result*"provoquant l'envoi du résultat lorsque la requête entrante a terminé son exécution. La tâche se rend alors disponible en atteignant la place "*task pool*" à nouveau, de même que l'identificateur que l'on retrouve dans la place "*unused job ids*".

7.2.3 Avortement commandé à distance

Dans certains cas, l'avortement d'un appel de sous-programme à distance peut générer un message de contrôle à destination de la partition distante (voir section 7.1.2 page 109). Dans ce cas, le message d'avortement est ajouté dans la place "*abort job*". Selon que le sous-programme est encore en cours d'exécution ou non, les deux cas de figures suivants peuvent se présenter.

Exécution non terminée

Si le sous-programme entrant est en cours d'exécution, la transition "*abort task*" est alors franchissable. Ceci provoque l'avortement de la tâche anonyme correspondant à la requête dont l'avortement est demandé. Le compteur totalisant le nombre de tâches dans le système est alors décrémenté.

Exécution déjà terminée

Si le sous-programme entrant a déjà achevé son exécution au moment où la demande d'avortement parvient dans la place "*abort job*", la transition "*forget abortion*" peut être franchie car l'identificateur de requête a été recyclé dans la place "*unused jobs*". Dans ce cas, la demande d'avortement est considérée comme obsolète et ne produit aucun effet.

De même qu'un conflit d'identification peut se produire au niveau de l'appelant (voir section 7.1.3 page 111), une requête d'avortement arrivant trop tard sur une partition cible peut être, de manière incorrecte, associée à une requête autre que la requête visée. Cependant, le message d'avortement étant traité de manière synchrone par la partition cible (la place "*abortion over*" n'est atteinte que lorsque l'avortement a été soit effectué soit abandonné), aucune nouvelle requête ne sera traitée avant la fin de la procédure d'avortement. Dans l'hypothèse d'une connexion unique et sans déséquencement des paquets entre deux partitions (c'est le cas avec TCP), un tel conflit ne peut donc pas se produire.

7.2.4 Procédures unidirectionnelles

Aucun traitement relatif aux procédures unidirectionnelles n'apparaît dans le réseau de Petri coloré de la figure 7.6 page 115. Ce cas est traité par la transition "*send result*", qui se contente de ne pas envoyer de résultat dans la place "*results network*".

Notons que par "résultat" on entend ici uniquement "exception éventuelle". En effet, les règles sémantiques permettant de rendre une procédure unidirectionnelle implique notamment qu'elle n'ait aucun paramètre de retour.

7.3 Conclusion

En nous appuyant sur l'interface normalisée de l'annexe des systèmes répartis, nous avons conçu et développé un modèle de gestion des appels de sous-programmes distants permettant aussi bien les appels bidirectionnels qu'unidirectionnels. Les conseils du manuel de référence demandant de traiter l'avortement d'une requête distante ont été intégrés : un tel avortement stoppera la tâche concrétisant un appel distant, afin d'éviter un gaspillage de temps processeur.

Nous avons rendu le système plus efficace en concevant et en réalisant un réservoir de tâches. Ce réservoir permet de disposer de tâches "prêtes à l'emploi" qui seront utilisées pour les nouvelles requêtes entrantes, économisant ainsi le temps nécessaire pour créer une nouvelle tâche en Ada.

8

Terminaison d'une application répartie

8.1 Exposé du problème

Le problème de la terminaison existe dans tout programme, réparti ou non. Il consiste en la question : *"si on n'arrêtait pas le programme maintenant, continuerait-il à effectuer les opérations pour lesquelles il a été prévu ?"*. Dans le cas où la réponse est négative, le programme peut être terminé sans risque, puisque le fait qu'il termine ou non ne sera pas perceptible. Ce problème, devenu un grand classique dans le domaine des systèmes répartis, a été abordé dès 1980 dans [Francez, 1980a] et [Dijkstra et Scholten, 1980]. Il n'en est toutefois pas fait mention dans le manuel de référence Ada [RM, 1995, Annexe E].

Dans un programme Ada monolithique, cette question est résolue de la manière suivante : lorsque le nombre de tâches actives d'un programme (une tâche active est une tâche dont le flot d'exécution n'est ni parvenu à son terme ni bloqué sur une construction `select` ne comportant pas l'alternative `terminate`) est égal à zéro, alors le programme peut terminer. Dans ce cas, toutes les tâches bloquées sur une construction `select` avec alternative `terminate` sont aiguillées sur cette alternative et le programme principal (constitué de la tâche d'environnement, qui est l'ancêtre de toutes les autres tâches) rend la main au système d'exploitation ou au moniteur. Bien entendu, toutes les tâches implicites qui pourraient être créées par l'exécutif Ada lui-même mais qui n'apparaissent pas dans le code utilisateur ne sont pas prises en compte dans le calcul et n'empêchent jamais la terminaison d'un programme.

Dans un programme réparti, il n'existe pas *a priori* d'état global permettant de connaître l'état de l'ensemble des tâches s'exécutant sur les diverses partitions à un moment donné. Il est donc nécessaire d'obtenir un consensus de la part de toutes les partitions participantes avant de prendre la décision de terminer effectivement les différents nœuds du système réparti. Ce problème a été étudié par de nombreux auteurs, dans le cadre de configuration particulières.

Nous avons utilisé successivement pour GLADE deux algorithmes originaux pour déterminer l'état de terminaison d'un programme réparti. Le premier, basé sur une architecture en étoile, garantit la détection de manière efficace et fiable uniquement en l'absence d'appels de sous-programmes unidirectionnels. Ce défaut lui a valu d'être remplacé par un second algorithme, présent dans l'implémentation actuelle de GLADE, qui s'affranchit de cette contrainte. Il fonctionne dès que le système réparti est connexe (ce qui est toujours le cas dans les systèmes répartis en Ada).

8.2 Conditions de déclenchement

Les deux algorithmes présentés ici nécessitent une opération active de la part d'une des partitions afin d'initier leur déroulement. Dans le cas où le point d'accueil est unique, celui-ci est utilisé pour déclencher chaque phase lorsqu'il est lui-même localement prêt à terminer. Comme expliqué dans la section suivante, cela implique qu'il vérifie régulièrement son état afin de savoir quelles sont les tâches actives sur sa partition. Cet état est vérifié par une tâche s'exécutant à la priorité la plus faible possible. Si une autre tâche monopolise le processeur, alors la terminaison n'a aucune raison d'être ; de fait, aucune puissance de calcul n'est utilisée pour le calcul de la terminaison globale lorsque la partition est active.

Lorsque plusieurs points d'accueil sont disponibles, celui ayant le plus petit identificateur de partition se voit attribuer le rôle d'initiateur. Lorsque l'initiateur disparaît, suite à une défaillance, un nouvel initiateur prend sa place en utilisant les mêmes critères de choix.

Dans l'hypothèse de la disparition de l'ensemble des points d'accueil d'un système réparti, l'application est considérée erronées et se termine automatiquement. En effet, il est impossible d'accéder aux informations de nommage ou de connecter de nouvelles partitions à l'application répartie en l'absence de point d'accueil.

8.3 Hypothèses

Nous avons choisi ces deux algorithmes de terminaison en raison des propriétés suivantes d'un système réparti en Ada utilisant GNAT et GLADE :

1. Il n'existe aucun moyen standard en Ada de connaître le nombre de tâches en cours d'exécution et non bloquées sur une clause select avec une alternative terminate. L'exécutif de GNAT nous fournit cependant le moyen d'obtenir cette information de manière active. Cela signifie que l'exécutif de GLADE n'est pas averti spontanément lors du changement d'état d'une tâche mais peut l'obtenir à une date précise.
2. Les protocoles de communication utilisés par GLADE sont tous fiables et ne déséquencent pas les messages. Si deux messages M et M' sont envoyés successivement d'une partition P_i à une partition P_j, alors le message M arrivera sur P_j avant M' (voir section 5.4 page 90).
3. Aucune création spontanée d'activité ne peut avoir lieu. Une partition dont l'activité est localement terminée restera dans cet état à moins de recevoir une requête entrante lui demandant d'exécuter un sous-programme appelable à distance.

Le premier algorithme utilise uniquement les propriétés 1 et 3. Le second pour sa part les utilise toutes.

8.4 Algorithme centralisé

L'algorithme centralisé, présent dans les premières versions de GLADE, vérifiait qu'il existait une date t comprise entre le début de l'application répartie et l'instant courant à laquelle toutes les partitions étaient prêtes à terminer localement. Cet algorithme utilisait des paires de vagues successives : la première vague d'une paire provoquait la mise sous surveillance de l'état de chaque partition, tandis que la seconde générait le retour d'observation de chaque partition.

Cet algorithme était adapté au cas des appels de sous-programmes distants bidirectionnels : en effet, si un appel de sous-programme distant a été envoyé sur le réseau mais pas encore reçu, l'émetteur, bloqué en attente de la réponse, n'est pas prêt à terminer localement. En revanche, dans le cas d'appels de sous-programmes unidirectionnels, l'émetteur est débloqué dès que sa requête est envoyée sur le réseau et peut donc être prêt à terminer localement. On peut donc avoir une situation dans laquelle toutes les partitions sont prêtes à terminer localement mais où la terminaison globale ne peut néanmoins être décidée, car un message en transit peut réveiller une des partitions après la phase d'observation. Nous avions donc le risque d'un "faux positif", situation dans laquelle la terminaison globale était décidée à tort.

8.5 Arbre de terminaison

Cependant, il est incorrect et dangereux de terminer une application répartie avant la fin de son exécution légitime. Nous avons donc conçu un nouvel algorithme, basé sur les propriétés des arbres couvrants, permettant de garantir la correction de l'état de terminaison.

Le principe de l'algorithme est le suivant : il construit tout d'abord un arbre. Une vague de messages de contrôle est ensuite envoyée depuis la racine à tous ses fils, et chaque fils l'envoie à ses propres fils. Un nœud de l'arbre est considéré comme prêt à terminer si les trois conditions suivantes sont vérifiées simultanément :

1. Le nœud n'a aucune activité.
2. Le nœud est une feuille de l'arbre, ou bien tous ses fils sont également prêts à terminer.
3. Le nœud n'a envoyé aucun message relatif aux appels de sous-programme à distance depuis qu'il a répondu à son père lors de la vague précédente (les raisons de la prise en compte des seuls messages sortants sont expliqués dans la section 8.5.4 page 124.

Cet algorithme a été inspiré par l'algorithme de terminaison statique décrit dans [Brzezinski et al., 1993] et par celui présenté dans [Chandy et Lamport, 1985]. Les restrictions sur la forme du système réparti (une étoile autour de la partition initiant l'algorithme de terminaison) ont cependant été levées grâce à la construction d'un arbre.

8.5.1 Construction de l'arbre

Le **graphe des partitions** est défini comme un arbre couvrant, dont les partitions du système réparti sont les nœuds et les connexions ouvertes entre les partitions les arcs. À partir de ce graphe, nous générons de manière dynamique un arbre comportant l'ensemble des partitions. Chaque partition n'a qu'une connaissance très incomplète de l'arbre ; elle ne connaît avec certitude que son père.

Chaque nouvelle vague est accompagnée d'une nouvelle estampille ("*timestamp*" en anglais) permettant de la distinguer des autres. L'estampille de la vague n est notée Φ_n. La suite Φ est monotone et strictement croissante, ce qui signifie que l'estampille d'une vague sera strictement supérieure à celles des vagues précédentes. Lorsqu'une partition P_i reçoit, de la partition P_j, un message de contrôle marqué de l'estampille Φ_n, trois cas peuvent se présenter :

1. Si Φ_n est supérieure à la dernière estampille reçue, alors une nouvelle vague est en cours de propagation. Dans ce cas, P_j devient le père de P_i et l'estampille courante de P_i devient Φ_n.
2. Si Φ_n est égale à son estampille courante, P_i a déjà un père associé à cette vague et ne change donc rien.
3. Si Φ_n est inférieure à l'estampille courante, on est en présence d'un message de contrôle obsolète. Ce message est ignoré.

Ce principe est visible sur la figure 8.1.

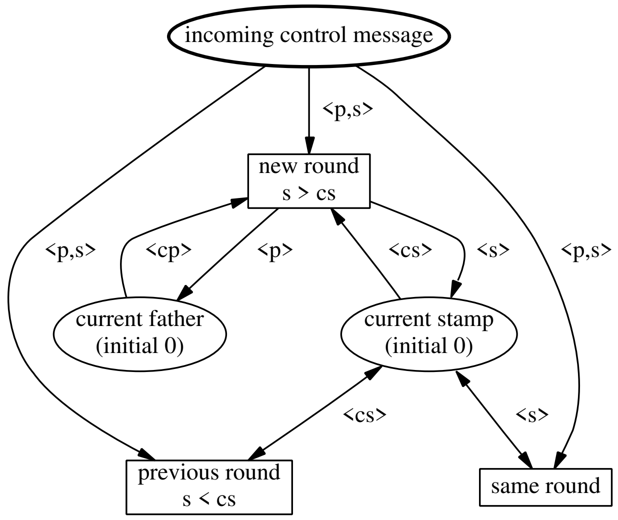


FIG. 8.1 – Réception d'un message de contrôle

De plus, chaque partition recevant un message de contrôle avec une nouvelle estampille en envoie un avec la même estampille à tous ses voisins (c'est-à-dire l'ensemble des partitions avec qui une connexion a été ouverte), excepté son nouveau père. Cela conduit, pour chaque vague, à la construction d'un arbre couvrant sur lequel l'algorithme de terminaison va pouvoir être appliqué. Notons que l'arbre associé à une vague particulière n'est pas nécessairement le même que celui associé à la vague suivante. En effet, les temps de propagation étant généralement variables, aucune prédiction de la structure de l'arbre n'est possible. La figure représente un graphe de partitions et un des arbres pouvant être construits à partir de ce graphe, en supposant que la partition P_1 initie la demande de terminaison.

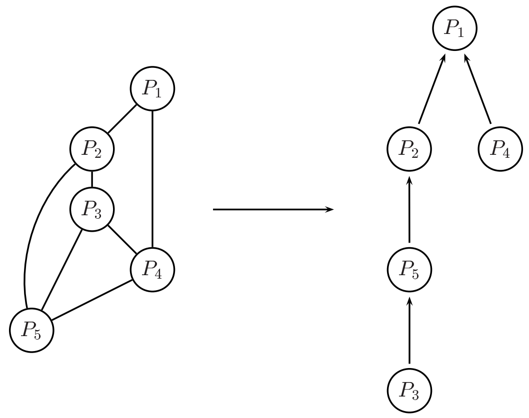


FIG. 8.2 – Transformation d'un graphe en arbre

On pourra également noter qu'aucun nouveau canal de communication n'est ouvert par cet algorithme. En effet, tous les messages émis depuis une partition sont destinés à un de ses voisins, et la définition d'un voisin est une partition avec laquelle on a déjà communiqué dans le passé.

8.5.2 Remontée des résultats

Le but de la $n^{\text{ème}}$ vague de messages de contrôle est de déterminer si, lorsqu'elles ont reçu la vague précédente $n-1$, l'ensemble des partitions étaient prêtes à terminer et si, depuis, elles n'ont envoyé aucun message susceptible de générer de l'activité. Pour cela, chaque partition répond à son père une fois qu'elle a elle-même obtenu la réponse de ses fils.

Chaque partition recevant un message de contrôle avec une nouvelle estampille correspondant à la vague n envoie à tous ses voisins excepté son père un message de contrôle similaire. La partition ne répondra à son père qu'après avoir obtenu la réponse de tous ses fils.

Cependant, nous avons montré plus haut qu'une partition ne connaît pas ses fils, ni même ne sait si elle en a. Les seules informations dont elle dispose sont la liste de ses voisins et son père pour la vague courante. Or, si une partition a des fils, ceux-ci se trouvent nécessairement parmi ses voisins, son père excepté. Une partition renvoie donc une réponse à son père uniquement lorsqu'elle a obtenu une réponse de l'ensemble de ses voisins. Si un de ses voisins a envoyé une réponse négative, ou si la partition n'est pas prête à terminer localement ou a envoyé un message relatif aux appels de sous-programme à distance depuis sa réponse au père lors de la vague précédente, alors une réponse négative est envoyée au père ; dans le cas contraire, une réponse positive est envoyée.

Lorsqu'une partition possédant déjà un père reçoit une demande d'un de ses voisins, elle lui renvoie immédiatement une réponse positive quel que soit son état vis à vis de la terminaison. Cela ne fausse aucunement le résultat, puisque chaque partition n'est censée renvoyer son état véritable et celui de ses fils qu'à son père ; son état n'intervient donc pas dans le calcul de la réponse de partitions ne se trouvant pas dans son ascendance.

8.5.3 Détection de la terminaison en temps fini

Quand l'application répartie est prête à se terminer les deux propriétés suivantes sont vérifiées :

1. toutes les partitions sont prêtes à terminer localement ;
2. aucune partition n'émet de message.

Supposons que cet état de terminaison globale survienne avant l'émission de la vague n. Dans ce cas, il est possible que des messages aient été émis entre les vagues $n - 1$ et n, la terminaison ne sera donc pas détectée à cette vague. Lors de la vague $n + 1$, aucun message n'aura été émis depuis la vague n et toutes les partitions seront encore prêtes à terminer localement. La terminaison sera donc décidée lors de cette seconde vague.

8.5.4 Messages en transit

Dans cette section, nous démontrons que si, lors d'une vague n, toutes les partitions sont prêtes à terminer localement et n'ont envoyé aucun message depuis la vague $n-1$, alors la terminaison globale peut être décidée, et que dans le cas contraire une réponse négative sera propagée jusqu'à l'initiateur de la vague.

Examinons d'abord la situation triviale dans laquelle au moins une partition refuse la terminaison parce qu'au moment d'envoyer sa réponse à son père, elle n'est pas prête à terminer localement. Dans ce cas, la terminaison globale sera rejetée.

Supposons ensuite qu'au moment de répondre à leur père respectif toutes les partitions sont prêtes à terminer localement. Nous allons montrer que si un le programme réparti n'est pas terminé globalement, alors au moins une partition rejettera la terminaison car elle aura envoyé un message après avoir répondu à la vague $n - 1$ et avant de répondre à la vague n.

Le schéma figure 8.3 page ci-contre présente un exemple : à la vague n, toutes les partitions sont prêtes à terminer localement (la terminaison locale est représentée par des pointillés sur la schéma). Les partitions 1, 4 et 5 n'auront plus jamais d'activité, mais les partitions 2 et 3 seront en fait réveillées par un ou plusieurs messages en transit et leur état de terminaison n'est pas permanent.

On note T_i^n la date à laquelle la partition P_i répond à son père à l'occasion de la vague n. Les T_i^n sont différents de partition en partition, étant donné l'absence d'horloge globale et les délais introduits par la structure de l'arbre (tous les nœuds n'ont pas le même nombre de fils *a priori*).

Soit $T_\alpha(P_i)$ la date de reprise d'activité d'une partition P_i après l'envoi de sa réponse à son père pour la vague n. On note T_α la date de la première activité après la vague n :

$$T_\alpha = \min_i T_\alpha(P_i)$$

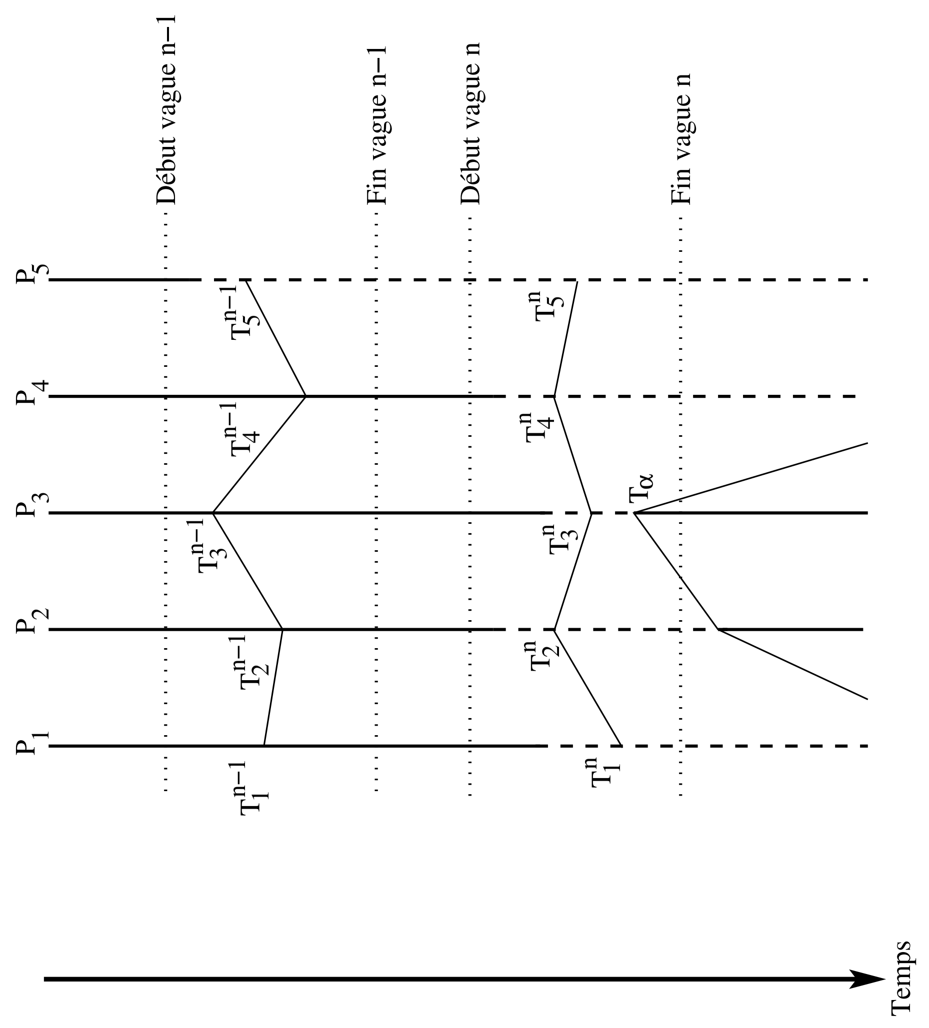


FIG. 8.3 – Détection de messages en transit

Dans notre exemple, cela correspond à la reprise d'activité de la partition 3. Étant donné qu'il n'y a pas de génération spontanée d'activité, cette fin de l'état de terminaison locale a obligatoirement été provoqué par un message. Appelons P_j la partition ayant envoyé ce message.

Deux possibilités sont envisageables :

1. P_j a envoyé le message après T_j^n. Dans ce cas-là, P_j aurait eu une activité entre T_j^n et T_α, ce qui est en contradiction avec la définition de T_α. Ce cas de figure est donc impossible.
2. P_j a envoyé le message avant T_j^n. Dans ce cas, étant donné que la présence de la vague $n-1$ garantit que les messages envoyés entre deux partitions u et v sont arrivés avant T_v^{n-1} grâce à la conservation de l'ordre des paquets dans les protocoles de communication de GLADE, le message a obligatoirement été émis entre T_j^{n-1} et T_j^n. P_j a donc envoyé une réponse négative à son père lors de la vague n, et la terminaison globale a été rejetée.

On a donc montré qu'en cas d'état de terminaison locale sur toutes les partitions au moment de la vague n, si une partition doit recommencer ses traitements dans le futur, alors la vague n obtiendra une réponse négative. La contraposée de ce théorème prouve qu'en cas d'état de terminaison locale sur toutes les partitions au moment de la vague n, une réponse positive à la vague n garantit qu'aucune activité n'aura plus jamais lieu dans le système réparti. La seule exception pourrait être l'arrivée d'une nouvelle partition dans le système réparti, mais nous avons choisi d'ignorer délibérément ce cas qui n'est pas évoqué dans le manuel de référence Ada ; nous considérons comme erroné le fait d'essayer de connecter une partition à un système réparti dont la terminaison globale a été décidée.

8.5.5 Résistance aux défaillances

Une précaution supplémentaire a été mise en œuvre pour résister à la défaillance de partitions au cours de l'algorithme. La partition qui initie la demande de terminaison globale accorde à ses voisins un temps maximum au-delà duquel elle n'attend plus de réponse pour la vague courante. Si certains voisins manquent à l'appel à l'échéance du délai imparti, la terminaison globale est considérée non acquise.

Comme mentionné dans la section 5.4 page 90, la détection de la défaillance d'un nœud se fait en un temps fini. Lorsque les voisins d'une partition défaillante s'aperçoivent que celle-ci est morte, ils l'enlèvent de leur liste de voisins. La réponse de cette partition ne sera donc plus attendue lors du calcul de la terminaison globale.

8.5.6 Notes sur l'algorithme

Cet algorithme présente une originalité par rapport à ceux trouvés habituellement dans la littérature portant sur la terminaison : il n'utilise aucun compteur des messages reçus afin de vérifier l'absence de messages en transit. Il est uniquement basé sur l'utilisation de variables binaires indiquant si une partition a envoyé ou non des messages entre deux vagues telles qu'elle les perçoit.

De plus, cet algorithme est très efficace : il peut, dans certains cas, décider de la terminaison globale de l'application répartie dès la vague suivant cette terminaison effective. Les algorithmes habituellement utilisés nécessitent deux phases, comme celui exposé dans la section 8.4 page 121, une phase de mise en observation suivie d'une phase de rapport d'observation.

8.6 Conclusion

Nous avons implémenté un mécanisme de terminaison réparti dont les propriétés sont comparables au mécanisme de terminaison d'une partition isolée utilisé par un programme Ada monolithique. L'algorithme utilisé tient compte des messages en transit dans tous les canaux de communication utilisés par l'application répartie, et n'ouvre aucun nouveau canal lors de son déroulement. Il supporte aussi un nombre quelconque de fautes survenant pendant la prise de décision ; ces fautes ne peuvent pas causer de déclenchement intempestif de la terminaison globale.

Toutefois, ce mécanisme n'est pas adapté à un modèle d'application du type client-serveur, dans laquelle le serveur a typiquement une durée de vie supérieure à celle du client. Nous avons pour cela étendu les possibilités de terminaison ; ces extensions sont présentées dans la section 9.2 page 131.

QUATRIÈME PARTIE

Outils et extensions

Nous présentons dans cette partie les outils et les extensions fournies avec GLADE ne faisant pas partie *stricto sensu* de l'annexe des systèmes répartis. Le chapitre 9 présente un descriptif des extensions que nous avons implémentées. Le chapitre 10 est consacré à GNATDIST, l'outil de partitionnement de GLADE. Le chapitre 11 présente les services externes supplémentaires que nous avons jugé indispensable de fournir pour une utilisation optimale de GLADE.

9 Extensions

9.1 Présentation

L'annexe des systèmes répartis du manuel de référence Ada spécifie un ensemble de règles qui doivent être respectées pour déclarer qu'une implémentation est conforme à la norme. Toutefois, il est possible de fournir dans une implémentation des extensions pour peu qu'elles ne modifient pas la sémantique des applications réparties.

Nous avons choisi d'étendre GLADE en y incorporant certaines extensions qui nous paraissaient indispensables dans un système réparti moderne. Tout d'abord, nous avons développé un modèle de terminaison permettant d'écrire des applications client-serveur de manière intuitive, ainsi qu'une gestion fine des politiques de résistance aux défaillances dues aux déconnexions intempestives de partitions. Nous avons également ajouté des possibilités de filtrage afin de rendre l'utilisation de GLADE compatible avec les politiques de sécurité imposées par certains sites, ou pour permettre d'utiliser des applications réparties en Ada au dessus de liens à bas débit grâce à la compression transparente "au vol". Il est également possible de déverminer un programme réparti simplement en rejouant certaines partitions. Pour finir, un exécutif léger permet, dans certains cas, de s'accommoder de conditions contraignantes concernant la mémoire ou la puissance de calcul en réduisant les ressource utilisées par GLADE, et un contrôle fin des priorités des requêtes entrantes permet d'aider au choix de la politique d'ordonnancement d'un tel système.

9.2 Extension du modèle de terminaison

9.2.1 Le modèle client-serveur

Le manuel de référence Ada autorise (mais n'oblige pas) un utilisateur à définir un programme principal pour chaque partition [RM, 1995, 10.2(7)]. Cela signifie qu'il est facile de créer une ou plusieurs partitions (appelées aussi clients) ne comportant aucun paquetage appelable à distance, dont le programme principal utilisera les services d'une ou plusieurs partitions tierces (appelées serveurs). Alors que chaque paquetage appelable à distance ne peut être présent que sur une seule partition du système réparti [RM, 1995, E.1(9)] (en effet, son attribut Partition_ID doit renvoyer un résultat similaire à chaque évaluation), rien n'empêche d'exécuter successivement ou simultanément plusieurs partitions clientes utilisant les mêmes serveurs.

Notons que le fait de ne pas posséder de paquetages appelables à distance ne signifie pas que ces clients ne peuvent offrir aucun service aux autres partitions du système : l'utilisation d'objets répartis permet à ces partitions d'être utilisées à distance, à travers les méthodes d'objets dont les références ont été préalablement transmises aux serveurs.

Cependant, les algorithmes de terminaison présentés dans le chapitre 8 page 119 et dérivés du modèle Ada monolithique ne s'appliquent pas au modèle basé sur des clients et des serveurs. En effet, un serveur a traditionnellement une durée de vie plus longue que celle de ses clients ; un client doit pouvoir terminer son exécution lorsqu'il n'a plus besoin du serveur, et le serveur ne doit pas obligatoirement être stoppé lorsqu'il n'a plus de client, étant donné que de nouveaux clients peuvent apparaître dans le futur.

9.2.2 Les différents modèles de terminaison

Nous avons choisi trois modèles de terminaison qui nous paraissent couvrir la majorité des cas d'applications réparties ; ces choix s'appliquent individuellement à chaque partition par le biais d'une clause de représentation dans le fichier de configuration de l'application (voir section 10.4.9 page 151) :

Global_Termination (valeur par défaut) : ce choix de terminaison représente le fonctionnement décrit dans les algorithmes du chapitre 8. Une telle partition participe au calcul de la terminaison globale de l'application, et n'est stoppée que lorsque cette terminaison globale a été décidée.

Local_Termination : une partition utilisant ce choix de terminaison termine son exécution lorsqu'elle n'a plus de tâche active. Elle participe au calcul de la terminaison globale au sens où tant qu'il existe au moins une partition avec ce choix de terminaison dans le système réparti, celui-ci considère qu'aucune terminaison globale n'est possible. C'est le type de terminaison le plus adapté à un client.

Deferred_Termination : une telle partition participe au processus de terminaison globale, mais ne se termine pas même si les autres partitions sont arrêtées. Elle sera donc prête à accepter de nouvelles requêtes entrantes par la suite ; c'est le type de terminaison le plus adapté à un serveur.

L'utilisation conjointe de Local_Termination et Deferred_Termination permet de créer des programmes basés sur le modèle client-serveur de manière intuitive. Chaque client terminera lorsque son programme principal aura terminé, alors que les serveurs tourneront indéfiniment.

Un autre modèle qu'il est possible de créer est celui induit par l'utilisation conjointe de Local_Termination et Global_Termination. Les serveurs restent actifs tant qu'ils ont des clients vivants, et se terminent dès qu'ils n'ont plus de clients.

9.3 Politique de reconnexion

L'annexe des systèmes répartis du manuel de référence Ada ne contient pas la notion de déconnexion d'une application répartie. En effet, un programme réparti écrit en Ada est censé se comporter comme une application monolithique, dans laquelle il est inconcevable qu'une partie du code disparaisse et devienne inatteignable.

Nous avons donc choisi d'offrir au programmeur la possibilité de définir la conduite à tenir en cas de disparition prématurée d'une partition. Trois politiques ont été implémentées, et peuvent être sélectionnées pour chaque partition par le biais d'une clause de représentation (voir section 10.4.10 page 151). Toutefois, certaines précautions doivent être prises lors du remplacement d'une partition déconnectée par une autre ; notamment, l'état interne de la partition défectueuse devra avoir été, si nécessaire, sauvegardé puis restauré sur la nouvelle partition à l'aide d'un mécanisme externe à l'annexe des systèmes répartis. L'utilisation de paquetages Shared_Passive peut être envisagé pour effectuer cette sauvegarde (voir section 4.9 page 81).

9.3.1 Erreur temporaire

L'option Fail_Until_Restart provoque la levée de l'exception Communication_Error dès qu'une partition essaye d'utiliser un service d'une partition P_i qui a été déconnectée entre-temps. Ce comportement est autorisée par le manuel de référence, car l'erreur peut être assimilée à un problème de communication [RM, 1995, E.4(16)].

Toutefois, si une partition P_j se connecte à l'application répartie et qu'elle possède certains paquetages Remote_Call_Interface qui étaient définis sur P_i auparavant, alors les appels à ces paquetages seront dès lors dirigés vers P_j. Aucune exception ne sera plus levée lors d'appels de sous-programme distants appartenant à ces paquetages.

9.3.2 Erreur définitive

L'option Rejected_On_Restart interdit à une partition P_j d'offrir les paquetages Remote_Call_Interface offerts précédemment par une partition P_i maintenant déconnectée du système. C'est le comportement par défaut du système, qui interdit à une partition considérée comme morte de ressusciter.

9.3.3 Erreur masquée

Afin d'offrir une résistance aux pannes par indisponibilité de manière transparente pour l'utilisateur (excepté le surcroît de temps nécessaire pour mettre en place un service de remplacement), une troisième option, Wait_Until_Restart, permet de bloquer les appels vers les paquetages Remote_Call_Interface d'une partition P_i déconnectée du système jusqu'à ce qu'une solution de remplacement soit disponible. Dès qu'une partition P_j affirme posséder les paquetages en question, les appels en attente sont débloqués et s'exécutent comme si aucune erreur n'avait eu lieu.

9.4 Filtres

Comme évoqué dans la section 5.4 page 90, nous avons ajouté la possibilité de définir des filtres entre deux partitions d'une application répartie. Ces filtres peuvent être utilisés pour :

1. la compression des données échangées ;
2. l'authentification des partitions ;
3. le chiffrement des paquets.

Un filtre, s'il est exempt d'erreurs, ne change rien à la sémantique d'une application répartie. Des données filtrées puis défiltrées seront inchangées, même si ces deux opérations ont lieu sur des partitions différentes (voir figure 9.1). L'opération de défiltrage est aussi appelée **filtrage inverse**.

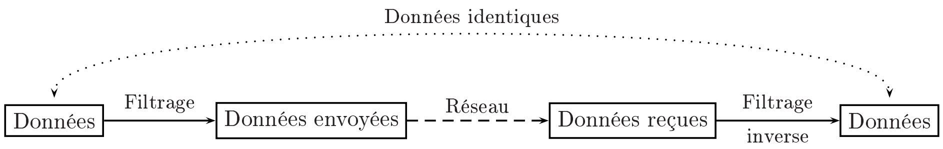


FIG. 9.1 – Filtrage des données entre partitions

Un filtre peut être appliqué à un canal de communication ou à tous les canaux impliquant une partition ; la syntaxe des directives à utiliser dans le fichier de configuration sont décrites dans la section 10.4.5 page 149.

9.4.1 Déclaration d'un filtre

La déclaration d'un filtre "Xyz" par un utilisateur de GLADE se fait en deux étapes :

1. Écriture (ou réutilisation) d'un paquetage System.Garlic.Filters.Xyz, définissant un nouveau type dérivé de System.Garlic.Filters.Filter_Type ; le nom de ce paquetage est imposé par GLADE. Ce paquetage sera élaboré sur toutes les partitions susceptibles d'utiliser ce filtre.
2. Définition des canaux de communication dans le fichier de configuration de GNATDIST.

L'extrait de code 9.1 page suivante présente les méthodes associées à un type de filtrage. Ces méthodes peuvent être regroupées en deux catégories : les méthodes d'administration, qui prennent en charge l'enregistrement du filtre auprès de l'exécutif et la négociation des clés de filtrage à utiliser (dans le cas d'un algorithme d'authentification ou de chiffrement par exemple), et les méthodes de filtrage, qui prennent en charge la transformation des paquets qui leur sont transmis.

La méthode Register_Filter doit être appelée pendant l'élaboration du paquetage contenant le filtre. Elle associe une instance du type filtre concerné avec le nom tel qu'il est utilisé dans le fichier de configuration de GNATDIST.

La méthode Generate_Params est utilisée pour négocier les paramètres à utiliser pour les filtres paramétrables. Cette procédure supporte à la fois les filtres asymétriques,

Exemple 9.1 Définition du type de filtrage

```
[...]
package System.Garlic.Filters is
[...]
private

   type Filter_Type   is abstract tagged limited null record;
   type Filter_Access is access all Filter_Type'Class;

   procedure Register_Filter
     (Filter : in Filter_Access;
      Name   : in String);

   type Filter_Params_Type is abstract tagged null record;
   type Filter_Params_Access is access all Filter_Params_Type'Class;

   procedure Generate_Params
      (Filter          : in  Filter_Type;
       Public_Params   : out Filter_Params_Access;
       Private_Params  : out Filter_Params_Access;
       Exchange_Params : out Boolean) is abstract;

   function Filter_Params_Read
      (Filter : Filter_Type;
       Stream : Ada.Streams.Stream_Element_Array)
      return Filter_Params_Access is abstract;

   function Filter_Params_Write
      (Filter : Filter_Type;
       Params : Filter_Params_Access)
      return Streams.Stream_Element_Access is abstract;

   function Filter_Incoming
      (Filter : in Filter_Type;
       Params : in Filter_Params_Access;
       Stream : in Streams.Stream_Element_Access;
       Offset : in Ada.Streams.Stream_Element_Offset)
      return Streams.Stream_Element_Access is abstract;

   function Filter_Outgoing
      (Filter : in     Filter_Type;
       Params : in     Filter_Params_Access;
       Stream : access Streams.Params_Stream_Type)
      return Streams.Stream_Element_Access is abstract;
[...]
end System.Garlic.Filters;
```

utilisant des paramètres publics pour coder les données et privés pour les décoder, et les filtres symétriques, dont les paramètres doivent être échangés entre les partitions participant à l'échange. L'échange est détaillé dans la section 9.4.2.

La méthode Filter_Outgoing est appelée pour filtrer les données brutes qui seront envoyées à une partition distante. Filter_Incoming est utilisée réciproquement pour défiltrer des données entrantes.

9.4.2 Échange de clés

Lorsque les possibilités de filtrage transparent de GLADE sont utilisées à des fins de confidentialité ou d'authentification, des clés doivent être échangées. Cela peut se faire de deux méthodes différentes :

1. Utiliser une clé secrète, qui est une convention partagée par les deux partitions. Il existe des algorithmes basés sur les clés secrètes très efficaces, tels que DES [National Bureau of Standards, 1980] ou RC5 [Rivest, 1994].

2. Utiliser un couple (clé publique, clé privée) pour chaque partition. Un message envoyé à une partition sera chiffré en utilisant la clé publique du destinataire, qui pourra le déchiffrer en utilisant sa clé privée. Notons qu'il est impossible de déduire la clé privée de la clé publique. Cette méthode est plus sûre que la première, car aucune partition autre que la partition destinataire ne pourra déchiffrer le message, y compris la partition l'ayant envoyé. L'algorithme à clé publique le plus utilisé est RSA [Adleman et al., 1978].

L'utilisation d'une clé secrète nécessite soit un accord *a priori* sur la valeur de cette clé, soit une négociation au moment de l'exécution. La deuxième méthode est plus sûre, car si une clé secrète utilisée lors d'une session précédente est compromise, la présence d'une nouvelle clé secrète lors de la session suivante garantit une confidentialité renouvelée.

Toutefois, la négociation dynamique d'une clé secrète nécessite l'utilisation d'un algorithme à clé publique pour échanger la clé partagée de manière confidentielle. C'est le rôle du "filtre d'enregistrement" associé à un canal de communication. La figure 9.2 page suivante montre un tel échange : P_i demande tout d'abord à P_j de lui envoyer sa clé publique (sur un canal non sécurisé), et lui transmet ensuite la clé (secrète) de session K que P_i et P_j utiliseront par la suite pour communiquer. Ce modèle est partiellement inspiré de [Kienzle et al., 1996].

On notera qu'une personne S espionnant le réseau de communication ne pourra déchiffrer aucun paquet de données. Le seule échange non chiffré qu'il pourra observer consiste en la demande de la clé publique de P_j par P_i[1].

[1] Cependant, rien n'empêche S de répondre à la place de P_j et de se faire passer pour elle. La protection contre ce genre d'attaque ne peut se faire qu'en stockant au préalable les clés publiques de toutes les partitions dans le code exécutable, mais cela signifie également qu'il faut y stocker les clés privées correspondantes, ce qui suppose que l'accès aux machines sur lesquelles tournent les partitions est sécurisé.

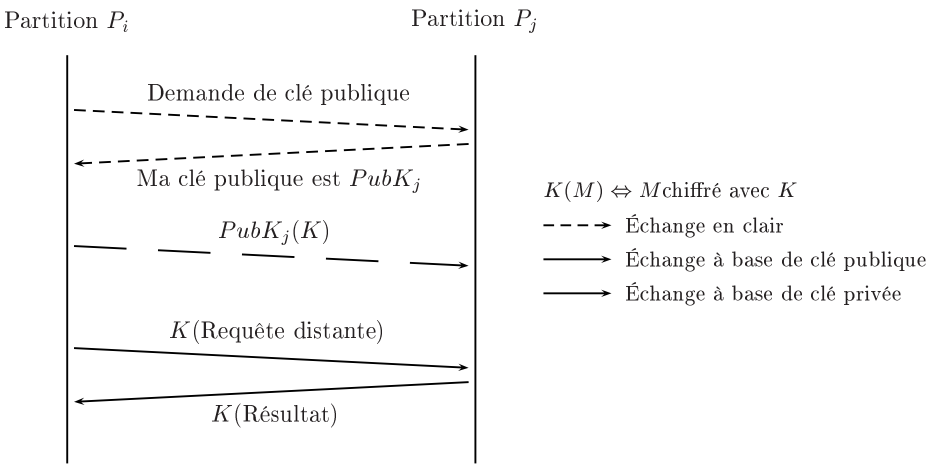


FIG. 9.2 – Négociation d'une clé privée

9.5 Rejeu d'application répartie

9.5.1 Exposé du problème

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. En effet, les changements de contexte se font à l'initiative de l'exécutif du compilateur et de celui du système hôte. Un même programme peut, selon les conditions de charge de travail sur la machine sur laquelle il est exécuté, donner des résultats différents. L'exemple 9.2 illustre cette situation : sur une machine peu chargée, il est possible que Calcul_Prenant_Environ_5_Secondes s'exécute en moins de 5 secondes, et dans ce cas la fonction Non_Déterministe renverra False. Sur une machine chargée, il est probable que cette fonction renverra True.

Exemple 9.2 Exemple de programme non déterministe

```
function Non_Déterministe return Boolean is
begin
   select
      delay 5.0;
      return True;
   then abort
      Calcul_Prenant_Environ_5_Secondes;
      return False;
   end select;
end Non_Déterministe;
```

Dans le cadre d'une application répartie, les communications entre partitions ajoutent également à l'entropie générale du système. L'utilisation de messages unidirectionnels peut rendre le comportement non déterministe. Dans l'exemple de la figure 9.3 page suivante, la partition P_1 envoie des messages unidirectionnels successivement à P_2 et

P_3. Dès réception du message de P_1, P_2 envoie à son tour un message unidirectionnel à P_3. Il est impossible de prévoir, sans connaître les temps de transfert de chaque paquet entre les partitions, quel message P_3 recevra en premier (correspondant aux phases 2 et 2' sur la figure).

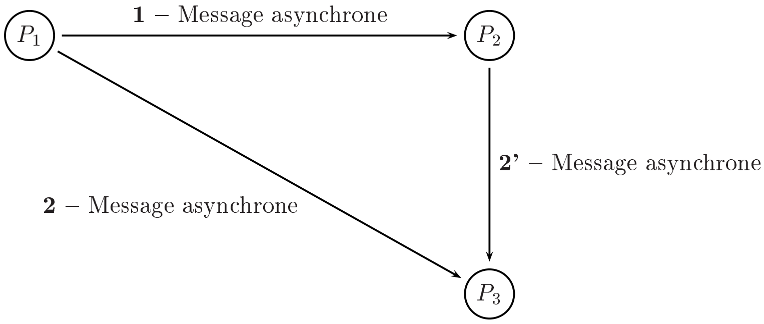


FIG. 9.3 – Messages unidirectionnels et non-déterminisme

Cependant, lors du développement d'une application, on souhaite souvent pouvoir rejouer une exécution de manière similaire à l'exécution initiale ; cela revient à ajouter du déterminisme dans une application non déterministe.

9.5.2 Enregistrement et rejeu

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 de traces sera utilisé lors du rejeu pour réordonnancer ensuite les événements dans l'ordre initial. Cette technique, décrite en détail dans [LeBlanc et Mellor-Crummey, 1986] et [McDowell et Helmbold, 1989], 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 d'implémenter dans GLADE le second type de rejeu, plus puissant que le précédent. [Levrouw et Audenaert, 1993] soulignent que la place disque nécessaire est beaucoup plus importante que dans le premier type de rejeu, mais cela est largement compensé par un avantage non négligeable : il est possible de rejouer l'application répartie du point de vue d'une unique partition P_i. Les échanges entre deux autres partitions P_j et P_k ne seront pas rejoués, mais leurs effets lors des communications avec P_i seront ressentis grâce au contenu des paquets stockés précédemment. De la même manière, les conséquences des échanges des autres partitions avec le monde extérieur seront visibles tels que perçus par P_i.

9.5.3 Détails de l'implémentation

Pour chaque partition, des paquets de taille variable, au format décrit figure 9.4, sont stockés dans un fichier de traces. Le fait d'enregistrer le délai entre deux messages entrants plutôt qu'uniquement l'ordre des messages permet d'effectuer une simulation plus proche de l'exécution initiale. De plus, nous n'utilisons pas de dates absolues, car un rejeu sur une machine dont la charge de travail est élevée donnerait un résultat trop différent de l'exécution initiale.

Délai depuis le paquet précédent
Taille du paquet de données
Premier octet ⋮ *(données)* Dernier octet

FIG. 9.4 – Format des paquets

Les informations enregistrées dans le fichier de traces sont uniquement celles concernant les appels de sous-programme à distance. Les échanges liés à la structure interne de GLADE ne sont pas reproduits, et sont aisément simulés au moment du rejeu. La seule information ne pouvant pas être simulée, car dépendant des conditions initiales, concerne l'attribution de l'identificateur de partition. Pour cela, un enregistrement spécial ne respectant pas le format décrit dessus est inséré au début du fichier de traces et contient le numéro de partition attribué lors de l'exécution initiale.

L'enregistrement et le rejeu sont asymétriques : pour enregistrer l'exécution d'une partition, il suffit de donner l'argument `--trace` sur sa ligne de commande lors de l'exécution initiale. Cela déclenche l'enregistrement du numéro de partition dès son attribution, ainsi que des messages entrants en provenance d'autres partitions (notons que le numéro de partition arrive obligatoirement avant la première requête représentant un appel de sous-programme à distance). Des dérivations placées à des points stratégiques de l'exécutif de GLADE permettent de sauvegarder ces échanges.

Le rejeu se fait par l'intermédiaire de l'option `--replay` sur la ligne de commande de la partition dont on souhaite rejouer l'exécution. Le rejeu utilise l'architecture modulaire de GLADE, et demande à utiliser un protocole spécial, appelé Replay, à la place du protocole utilisé en première instance. Ce protocole lit les événements dans le fichier de traces, et les envoie au cœur de GLADE comme s'ils provenaient réellement d'une autre partition.

9.6 Exécutif léger

Dans certains cas particuliers, il n'est pas nécessaire de disposer sur une partition de toutes les possibilités offertes par GLADE. Par exemple, dans un système réparti constitué de stations de travail et d'un robot mobile, il est peu probable que le robot soit lui-même un point d'accueil. De plus, le robot peut être uniquement client du reste de l'application répartie et n'offrir aucun service. Dans ce cas, il est souhaitable de pouvoir réduire l'encombrement mémoire de l'exécutif de GLADE ; nous avons donc créé un exécutif léger permettant de s'accommoder de contraintes sévères.

Les conditions suivantes sont nécessaires et suffisantes pour déclencher l'utilisation automatique de l'exécutif léger pour une partition :

1. La partition ne possède aucun paquetage catégorisé Remote_Call_Interface.
2. La partition a un mode de terminaison local.
3. Aucun des paquetages présents sur la partition n'a de visibilité sur la définition d'un type pointeur sur objet distant.

La troisième condition garantit que la partition n'exporte aucun objet pouvant être appelé par la suite depuis une autre partition. Notons que l'absence de paquetages Remote_Call_Interface interdit de prendre un pointeur sur sous-programme distant pointant sur un sous-programme de la partition, car ces sous-programmes cibles doivent être déclarés dans un paquetage Remote_Call_Interface [RM, 1995, E.2.2(12)].

À l'heure actuelle, l'exécutif est allégé de la manière suivante :

1. Comme la partition n'exporte aucun service, aucune tâche n'est créée pour traiter les requêtes entrantes. Toutes les demandes, administratives ou relatives aux appels de sous-programme distants, sont traitées sous la forme de requête-réponse. La partition ne recevra donc aucune requête sans l'avoir demandée au préalable.
2. La partition ne dispose pas du réservoir de tâche ni de la tâche de remplissage de ce réservoir.
3. La partition ne contient pas de code relatif au calcul de la terminaison globale.

Il est possible d'aller plus loin dans l'allégement de la puissance de calcul et de l'occupation mémoire nécessaires : une version future de l'exécutif de GLADE n'utilisera aucune tâche et aucun objet protégé.

9.7 Échange de priorités

Il est possible pour deux partitions d'une application répartie d'utiliser des politiques d'ordonnancement différentes [RM, 1995, D.2.2(6)], c'est-à-dire d'utiliser des algorithmes différents pour affecter les tâches du programme aux processeurs disponibles. De plus, si deux partitions se trouvent sur des nœuds physiques d'architectures différentes, le type System.Priority n'a pas obligatoirement la même définition [RM, 1995, D.1(10)].

Pour ces deux raisons, GLADE ne transmet pas, dans son comportement par défaut, la priorité de la tâche appelante à la tâche traitant la requête entrante. Cependant, nous avons ajouté une directive de compilation dans le fichier de configuration de GNATDIST (voir section 10.4.7 page 150) permettant de faire correspondre les priorités de ces deux tâches. Dans ce cas, la priorité de l'appelant est ramenée à un nombre normalisé compris entre 0 et 255. La priorité de l'appelé est calculée en dénormalisant cette valeur. La figure 9.5 page ci-contre représente un tel échange.

Cette option de GLADE doit être utilisée avec circonspection par l'architecte de l'application répartie. En effet, si elle permet de s'affranchir des différences de définitions du type System.Priority, elle ne prend pas en compte les différentes politiques d'ordonnancement.

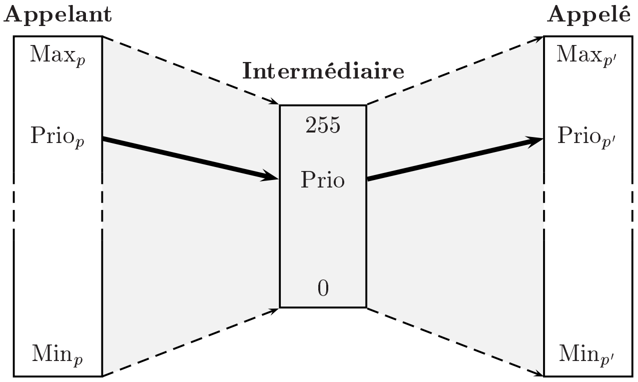


FIG. 9.5 – Transmission de priorités entre partitions

9.8 Conclusion

Le manuel de référence ne spécifie pas le fonctionnement interne de l'exécutif des systèmes répartis. Nous en avons tiré avantage pour concevoir et implémenter un certain nombre d'extensions dont nous estimions qu'elles rendraient l'utilisation de GLADE plus aisée.

Ces extensions, qui portent aussi bien sur la résistance aux défaillances que sur la politique de sécurité ou l'espace mémoire disponible, permettent de s'affranchir de contraintes imposées par l'architecture ou l'environnement utilisés par l'application répartie. Nous continuerons à étendre les possibilités de GLADE au fur et à mesure de l'apparition des besoins.

10

L'outil de partitionnement

10.1 Introduction

Un programme réparti est composé de plusieurs partitions. Le fait de répartir les différents paquetages sur les partitions est appelé "processus de partitionnement" ou "configuration des partitions" [RM, 1995, E(4-5)]. Ce processus n'est pas décrit dans le manuel de référence Ada ; seuls ses effets sont spécifiés.

Notre outil Gnatdist permet de décrire l'ensemble du programme réparti par le biais d'un fichier de configuration dont la syntaxe et la sémantique sont proches de celles du langage Ada. Ce fichier de configuration permet la description des différentes partitions et de leurs paquetages, ainsi que l'utilisation des fonctionnalités avancées de Glade, comme le filtrage ou la réplication des points d'accueil.

10.2 Schéma global

La figure 10.1 page suivante présente une vue globale de l'utilisation de Gnatdist. L'utilisateur, après avoir écrit le fichier de configuration décrivant l'application répartie, invoque Gnatdist avec ce fichier comme argument. Gnatdist vérifie alors la disponibilité de tous les fichiers sources nécessaires pour compiler l'application en mode non réparti, vérifie ces fichiers en utilisant la passe d'analyse sémantique de GNAT, puis génère les souches et les squelettes pour tous les paquetages ayant la catégorisation Remote_Call_Interface.

Ensuite, pour chaque partition, Gnatdist génère un fichier contenant les options associées à cette partition, par exemple le fait d'être ou non un point d'accueil. Le programme principal de chaque partition est ensuite généré par Gnatdist, programme principal qui se chargera d'initialiser l'exécutif de Glade et qui, après avoir éventuellement exécuté le programme principal spécifié par l'utilisateur, se chargera de l'initiation du processus de terminaison (voir chapitre 8 page 119).

Pour remplir son rôle, Gnatdist doit invoquer plusieurs outils de la chaîne de développement GNAT (voir section 3.1 page 61). Tout d'abord, les fichiers contenant

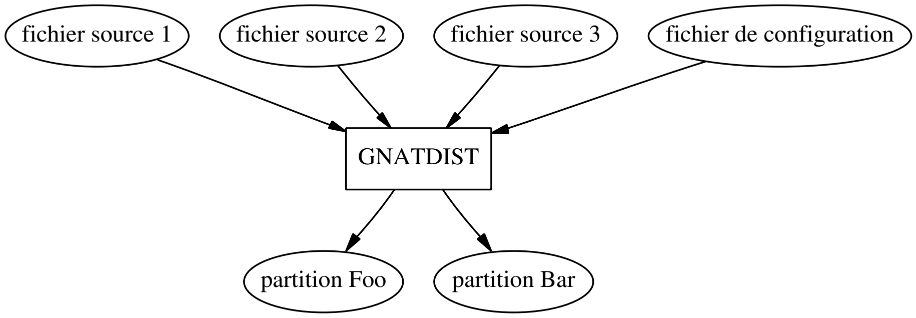


FIG. 10.1 – Schéma global du fonctionnement de GNATDIST

le code source de l'application répartie doivent être analysés, afin de vérifier qu'ils ne contiennent pas d'erreurs et qu'il n'y a pas de fichier manquant. L'analyse permet également d'extraire la liste des paquetages contenant la directive de compilation Remote_Call_Interface, ces paquetages jouant un rôle privilégié dans l'application répartie.

Afin d'extraire cette information des fichiers sources, GNATDIST utilise une partie des modules de l'utilitaire GNATMAKE livré avec GNAT. Ceci permet à GNATDIST de demander la compilation et l'analyse récursive de toutes les unités dont dépendent les paquetages mentionnés dans son fichier de configuration, et d'obtenir les informations de catégorisation extraites par le compilateur. À partir de ces informations, GNATDIST peut vérifier que chaque paquetage Remote_Call_Interface est placé sur une et une seule partition du système.

La liste de ces paquetages catégorisés est ensuite utilisée afin de générer les souches et les squelettes de chacun d'eux. Un mécanisme d'optimisation permet, en cas d'utilisations successives de GNATDIST, de ne régénérer que les souches et squelettes rendus obsolètes par une modification du paquetage source.

Ensuite, pour chaque partition déclarée dans le fichier de configuration, GNATDIST génère deux fichiers Ada contenant respectivement la liste des options à appliquer à cette partition et le programme principal qui sera utilisé par cette partition. Certaines propriétés de l'exécutif, comme la faculté de pouvoir traiter un appel entrant, ne doivent être déclenchées qu'après l'élaboration complète de la partition [RM, 1995, E.5(21)] ; c'est donc dans le programme principal que sont démarrées ces opérations. Si l'utilisateur a spécifié une procédure à utiliser comme programme principal de la partition, celle-ci est appelée par le programme principal construit par GNATDIST. L'utilitaire GNATMAKE est ensuite utilisé pour construire les exécutables correspondant à chaque partition.

10.3 Construction partielle d'application répartie

Dans certains cas, un développeur ne souhaite pas distribuer les corps de paquetages correspondant à un service qu'il fournit. En effet, les spécifications suffisent normalement à décrire le service au niveau sémantique et doivent masquer les détails d'implémentation.

Pour cette raison, GNATDIST supporte la construction partielle d'une application répartie. Lorsque l'utilisateur spécifie un ou plusieurs noms de partitions sur la ligne de commande, en plus du nom du fichier de configuration et des options de compilation, GNATDIST ne considère pas l'absence de certaines unités comme une erreur dans la mesure où l'outil parvient à construire les partitions demandées.

10.4 Syntaxe du fichier de configuration

Cette section décrit la syntaxe et les effets sémantiques du fichier de configuration de GNATDIST. Nous utilisons une grammaire BNF[1] [Koster, 1971] modifiée similaire à celle présente dans le manuel de référence Ada [RM, 1995, 1.1.4].

10.4.1 Configuration initiale

Dans cette section, nous présentons la syntaxe utilisée pour décrire une application répartie basique. Les sections suivantes décriront comment utiliser les extensions que nous avons ajouté dans GLADE.

Déclaration de configuration

```
CONFIGURATION ::=
     configuration IDENTIFIER is DECLARATIVE_PART
     [begin SEQUENCE_OF_STATEMENTS]
     end [IDENTIFIER];
DECLARATIVE_PART ::=
     { DECLARATIVE_ITEM }
DECLARATIVE_ITEM ::=
     PARTITION_DECLARATION
  |  CHANNEL_DECLARATION
  |  REPRESENTATION_CLAUSE
  |  SUBPROGRAM_DECLARATION
  |  IMPORTED_FUNCTION_DECLARATION
  |  MAIN_SUBPROGRAM_DECLARATION
  |  PRAGMA
SEQUENCE_OF_STATEMENTS ::=
     STATEMENT { STATEMENT }
STATEMENT ::=
     PARTITION_IDENTIFIER := UNIT_LIST;
UNIT_LIST ::=
     ([UNIT_IDENTIFIER {, UNIT_IDENTIFIER}])
```

Un fichier de configuration pour GNATDIST commence par le mot-clé configuration, suivi d'un identificateur nommant le programme réparti. Si le mot-clé end terminant le fichier est suivi d'un identificateur, alors cet identificateur doit être similaire à celui suivant le configuration. L'identificateur doit correspondre au nom du fichier de configuration, une fois le suffixe .cfg supprimé. Cet identificateur est utilisé par GNATDIST

[1] Backus Naur Form

pour créer un sous-répertoire contenant les divers fichiers annexes utilisés pour créer les partitions.

La description de la configuration est divisée en deux parties : la première, la partie déclarative, est obligatoire et comprend notamment la déclaration des différentes partition. La seconde, optionnelle et comportant uniquement des affectations, permet d'affecter des paquetages à des partitions au cas où cela n'aie pas été fait dans la première partie.

Déclaration de partition

PARTITION_DECLARATION ::=
    PARTITION_IDENTIFIER : **partition** [:= UNIT_LIST] ;

Une partition peut être entièrement déclarée dans la partie déclarative, à l'aide du type prédéfini partition. On remarquera que la valeur initiale peut consister en une paire de parenthèses, signifiant que cette partition ne possède aucun paquetage catégorisé comme Remote_Call_Interface ou Shared_Passive.

Une partition est déclarée comme un ensemble d'unités de bibliothèque. Ces unités peuvent être catégorisés Remote_Call_Interface ou Shared_Passive, auquel cas elles ne doivent être présentes que sur une seule partition. Elle peuvent également ne pas porter ces catégorisations ; dans ce cas, il est possible de les inclure dans plusieurs partitions simultanément.

Si une partition est déclarée mais non initialisée, cette initialisation devra être faite explicitement dans la seconde partie du fichier de configuration. Cette possibilité, maintenant obsolète, est cependant conservée pour des raisons de compatibilité avec les premières versions de GNATDIST.

Déclaration de sous-programme principal

MAIN_SUBPROGRAM_DECLARATION ::=
    procedure SUBPROGRAM_IDENTIFIER **is in** PARTITION_IDENTIFIER ;

Cette construction permet de désigner à la fois la partition principale et le sous-programme principal à exécuter sur cette partition. Ce sous-programme est utilisé comme la procédure principale de la version non-répartie dans les phases préliminaires de vérification sémantique.

10.4.2 Extensions

Cette section présente la syntaxe permettant d'utiliser les extensions au modèle de répartition de l'annexe des systèmes répartis décrits dans le chapitre 9 page 131. L'utilisation de ces extensions se fait à travers des directives de compilation, lorsque ces extensions s'appliquent à l'ensemble du programme réparti, ou à travers des clauses de représentation, lorsque ces extensions s'appliquent à une ou plusieurs partitions du système.

Les termes "directive de compilation" et "clause de représentation" sont issues du manuel de référence Ada. Leur usage ici est impropre, mais permet d'unifier le vocabulaire utilisé pour GNATDIST avec le vocabulaire traditionnel Ada.

Directives de compilation
PRAGMA ::=   **pragma** PRAGMA_NAME [(PRAGMA_ARGUMENT {,PRAGMA_ARGUMENT})]   PRAGMA_ARGUMENT ::=   [PARAMETER_NAME =>] PARAMETER_ARGUMENT

Les directives de compilation, introduites par le mot-clé pragma, permettent de spécifier des propriétés globales de l'application répartie.

Clauses de représentation
REPRESENTATION_CLAUSE ::=   **for** PARTITION_IDENTIFIER'ATTRIBUTE_NAME **use** ATTRIBUTE_ARGUMENT ; | **for partition**'ATTRIBUTE_NAME **use** ATTRIBUTE_ARGUMENT ;

Une clause de représentation de partition s'applique soit à une partition spécifique soit à l'ensemble des partitions décrites dans le fichier de configuration. Le choix de l'ensemble des partitions est indiqué par l'utilisation du mot-clé partition à la place de l'identificateur de partition.

Fonction importée
IMPORTED_FUNCTION_DECLARATION ::=   **function** FUNCTION_IDENTIFIER (PARAMETER_IDENTIFIER : **[in]** String) **return** String ;   **pragma** Import ([Convention =>] CONVENTION_LITERAL,   [Entity =>] FUNCTION_IDENTIFIER, [External_Name =>] STRING_LITERAL) ;

L'importation de fonction autorise une certaine dynamicité dans la gestion de l'application répartie. Une fonction, importée depuis Ada ou depuis une commande externe, peut être utilisée dans une clause de représentation afin de spécifier, par exemple, sur quelle machine lancer une partition donnée. Un programme externe renvoyant la machine la moins chargée (avec le plus de temps processeur disponible) permettra à l'application répartie de s'exécuter plus rapidement.

On notera que seule les fonctions à un paramètre de type chaîne de caractères renvoyant une chaîne de caractères sont importables. Cela suffit en effet à couvrir l'ensemble des besoins des extensions de GNATDIST.

10.4.3 Localisation du point d'accueil principal

Localisation du point d'accueil
pragma Boot_Server ([Protocol_Name =>] STRING_LITERAL, [Protocol_Data =>] STRING_LITERAL) ;

Lorsqu'une partition souhaite participer à une application répartie, il est nécessaire qu'elle sache où et comment contacter un des points d'accueil de l'application (voir section 6.3 page 99). La directive de compilation Boot_Server a un double effet :

1. elle indique à la partition principale avec quel protocole et quels paramètres elle

doit se mettre en attente de requêtes des autres partitions ;

2. elle indique aux partitions secondaires comment et où contacter la partition principale.

Cette directive ne permet pas d'exploiter directement les points d'accueil multiples, mais permet de démarrer des partitions automatiquement lorsque la partition principale est encore vivante.

10.4.4 Lancement des partitions secondaires

Lancement des partitions secondaires
pragma Starter ([Convention =>] CONVENTION_LITERAL) ; CONVENTION_LITERAL ::= Ada | None | Shell

Un programme réparti étant composé de plusieurs partitions, il peut être souhaitable de lancer, au démarrage de la partition principale, l'ensemble des exécutables représentant les différentes partitions. La directive de compilation Starter permet de choisir parmi trois méthodes de lancement :

Ada : la partition principale, après s'être mise en écoute sur un port libre du système, lancera, par le biais d'un interpréteur de commande, les autres partitions sur les machines appropriées en leur donnant les bons paramètres sur leur ligne de commande respective.

None : la partition principale ne lancera automatiquement aucune autre partition. Ce mode est particulièrement adapté au fonctionnement en client-serveur (voir section 9.2.1 page 131).

Shell : un script sera créé, dont l'exécution lancera l'ensemble des partitions du programme réparti.

Si une partition n'a pas été associée à un ordinateur par le biais d'une clause de représentation (voir section 10.4.2 page 146), l'utilisateur se verra demander un nom de machine à utiliser.

Localisation des partitions
for PARTITION_IDENTIFIER'Host **use** STRING_LITERAL ; | **for** PARTITION_IDENTIFIER'Host **use** IMPORTED_FUNCTION_IDENTIFIER ; | **for partition**'Host **use** STRING_LITERAL ; | **for partition**'Host **use** IMPORTED_FUNCTION_IDENTIFIER ;

Il est possible de spécifier dans le fichier de configuration sur quel ordinateur une partition sera élaborée. La désignation de cet ordinateur est soit une chaîne de caractères statique, soit le résultat d'un appel de fonction importée, que ce soit une fonction définie en Ada ou un programme externe.

Dans le cas d'une fonction importée, l'argument passé à la fonction est le nom de la partition tel qu'il est défini dans le fichier de configuration. Le résultat doit être un nom de machine utilisable avec la commande rsh.

Paramètres de la ligne de commande
for PARTITION_IDENTIFIER'Command_Line **use** STRING_LITERAL ; | **for partition**'Command_Line **use** STRING_LITERAL ;

Lorsqu'une partition est lancée automatiquement, il est parfois souhaitable de lui transmettre des arguments supplémentaires sur la ligne de commande, qui n'est plus accessible de manière traditionnelle à l'utilisateur. Cette clause de représentation permet de spécifier une chaîne de caractères qui sera automatiquement ajoutée à la ligne de commande de la partition fraîchement lancée.

Cet attribut ne s'applique pas à la partition principale dans le cas où la directive de compilation Starter est utilisée avec un paramètre égal à Ada. En effet, la partition principale est chargée de lancer les autres partitions, il est trop tard dans son cas pour ajouter des paramètres sur sa ligne de commande.

Déclaration de sous-programme principal secondaire
for PARTITION_IDENTIFIER'Main **use** SUBPROGRAM_IDENTIFIER ; | **for partition**'Main **use** SUBPROGRAM_IDENTIFIER ; SUBPROGRAM_DECLARATION ::= **procedure** SUBPROGRAM_IDENTIFIER ;

Il est possible de spécifier un sous-programme principal par partition, comme indiqué dans [RM, 1995, 10.2(7,21)]. Par défaut, toutes les partitions secondaires ont un sous-programme principal implicite correspondant à l'instruction null. En utilisant cette clause de représentation, l'utilisateur peut choisir un sous-programme principal différent sur chaque partition.

10.4.5 Gestion des filtres

Déclaration de canal
CHANNEL_DECLARATION ::= CHANNEL_IDENTIFIER : **channel** [:= PARTITION_PEER] ; PARTITION_PEER ::= PARTITION$_1$_IDENTIFIER, PARTITION$_2$_IDENTIFIER

Les canaux de communication considérés ici sont l'ensemble des paires de partitions distinctes. Ils représentent un support idéal pour les filtres (voir section 9.4 page 134) dans la mesure ou un filtre s'applique nécessairement à la communication entre deux partitions.

Déclaration de filtre
for CHANNEL_IDENTIFIER'Filter **use** STRING_LITERAL ; | **for channel**'Filter **use** STRING_LITERAL ; | **for** PARTITION_IDENTIFIER'Filter **use** STRING_LITERAL ; | **for partition**'Filter **use** STRING_LITERAL ;

Un filtre peut s'appliquer à un canal, à l'ensemble des canaux ou à une ou toutes les partitions (dans ce cas, tous les canaux auxquels appartiennent les partitions en question sont concernés). Le nom du filtre, qui doit être une chaîne de caractères, correspond au nom d'une unité enfant de System.Garlic.Filters. Par exemple, le nom "Zip" utilisera le type de filtre défini dans le paquetage System.Garlic.Filters.Zip.

Filtre utilisé pour l'enregistrement
pragma Registration_Filter ([Filter =>] STRING_LITERAL) ;

Un filtre particulier peut être utilisé pour la négociation d'une clé de session (voir section 9.4.2 page 136), et déclaré grâce à cette directive.

10.4.6 Vérification des versions

Vérification de cohérence des versions
pragma Version ([Check =>] BOOLEAN_LITERAL) ; BOOLEAN_LITERAL ::= False | True

Un des points forts de la répartition en Ada est la vérification automatique de la cohérence des versions des paquetages catégorisés comme Remote_Call_Interface ou Shared_Passive et des paquetages qui en dépendent sémantiquement (voir section 1.4.4 page 31). Dans certains cas, il peut être souhaitable de désactiver ce contrôle de version ; cette démarche dangereuse ne doit effectuée qu'en en comprenant toutes les conséquences en cas de problème.

10.4.7 Transmission des priorités

Propagation des priorités entre partitions
pragma Priority ([Propagate =>] BOOLEAN_LITERAL) ;

Dans la section 9.7 page 140, nous identifions les problèmes dûs au passage de la priorité de la tâche appelante à la tâche exécutant une requête distante. C'est pourquoi, par défaut, GNATDIST ne la transmet pas, sauf si l'utilisateur le demande explicitement, à l'aide de la directive de compilation Priority. Cette directive ne doit être utilisée que lorsque les politiques d'ordonnancement et les définitions des priorités sont les mêmes sur toutes les partitions du système réparti.

10.4.8 Répertoire de stockage des exécutables

Répertoire de stockage
for PARTITION_IDENTIFIER'Storage_Dir **use** STRING_LITERAL ; | **for partition**'Storage_Dir **use** STRING_LITERAL ;

Lorsqu'une partition est lancée automatiquement par effet de la présence, dans le fichier de configuration, de la directive de compilation Starter avec un argument autre que None, l'exécutable correspondant à cette partition sera recherché dans le répertoire indiqué par la clause de représentation Storage_Dir. Par défaut, cet attribut est positionné au répertoire depuis lequel GNATDIST a été lancé.

Cet attribut est également utilisé par GNATDIST lui-même. En effet, les exécutables générés pour chaque partition le seront dans le répertoire demandé. Cela permet de désigner, de manière non ambiguë, un chemin logique unique visible depuis l'ensemble du réseau.

10.4.9 Terminaison d'une partition

Terminaison d'une partition

```
    for PARTITION_IDENTIFIER'Termination use TERMINATION_LITERAL;
  | for partition'Termination use TERMINATION_LITERAL;
TERMINATION_LITERAL ::=
    Deferred_Termination | Global_Termination | Local_Termination
```

Dans la section 9.2 page 131, nous présentons les trois politiques de terminaison que nous avons adoptées. Cette clause de représentation permet d'appliquer à une seule ou à l'ensemble des partitions une politique particulière. La valeur par défaut en l'absence d'un tel attribut est Global_Termination.

10.4.10 Politique de reconnexion

Politique de reconnexion

```
    for PARTITION_IDENTIFIER'Reconnection use RECONNECTION_LITERAL;
  | for partition'Reconnection use RECONNECTION_LITERAL;
RECONNECTION_LITERAL ::=
    Fail_Until_Restart | Rejected_On_Restart | Wait_Until_Restart
```

La section 9.3 page 133 présente la signification des différentes politiques de reconnexion ainsi que leur fonctionnement, nous ne les détaillons donc pas ici.

10.4.11 Nombre de tâches anonymes d'une partition

Contrôle du nombre de tâches actives

```
    for PARTITION_IDENTIFIER'Task_Pool use TASK_POOL_SIZE_ARRAY;
  | for partition'Task_Pool use TASK_POOL_SIZE_ARRAY;
TASK_POOL_SIZE_ARRAY ::=
    (NATURAL1_LITERAL, NATURAL2_LITERAL, NATURAL3_LITERAL)
```

Comme décrit dans la section 7.2.1 page 114, le réservoir de tâches anonymes présent sur chaque partition active et permettant de traiter les requêtes entrantes se configure avec trois quantités. Cette clause de représentation permet de donner, partition par partition, un triplet correspondant respectivement aux paramètres lowmark, highmark et maxmark.

10.5 Conclusion

Nous avons conçu et réalisé un outil de partitionnement permettant de décrire la structure d'une application répartie. La syntaxe purement déclarative du langage de description a été choisie pour être proche de celle du langage Ada, langage supposé connu du développeur. Cet outil de partitionnement permet non seulement d'utiliser toutes les possibilités de l'annexe des systèmes répartis décrites dans le manuel de référence, comme la définition d'un programme principal sur chaque partition, mais également de tirer avantage des nombreuses extensions dont nous avons enrichi GLADE.

À l'aide de cet outil, il est également possible de préciser la localisation physique des différentes partitions de l'application répartie. Au besoin, un programme shell ou une portion de code Ada insérée dans la partition principale effectue le lancement de l'ensemble des partitions participant au programme.

De plus, cet outil permet la construction partielle d'une application répartie. Il est possible de ne construire que certaines partitions, sans disposer obligatoirement de l'implémentation des paquetages Remote_Call_Interface référencés dans ces partitions.

11

Services supplémentaires

11.1 Présentation

Comme nous l'avons évoqué dans la section 1.5.2 page 35, une des caractéristiques intéressantes de CORBA est sa modularité. Le cœur du système est réduit, et des services additionnels viennent enrichir ses fonctionnalités de manière incrémentale. Cela permet non seulement de ne pas encombrer tous les systèmes CORBA avec des services qui ne seront pas utilisés, mais offre également à l'architecte de l'application répartie la possibilité d'obtenir différents services de différentes sources. Lorsqu'un service n'offre pas les performances ou la robustesse attendue, il peut être remplacé par une autre implémentation fournissant les mêmes fonctionnalités.

En tant qu'utilisateurs de GLADE nous-même, nous avons été amené à écrire certains services que nous avons souhaité offrir à la communauté. Ce chapitre détaille les principaux d'entre eux.

11.2 Nommage de ressources

Dans CORBA, le service de nommage est un service externe indispensable pour permettre aux différentes parties d'une application répartie de dialoguer. Tous les vendeurs de logiciels CORBA fournissent leur implémentation de ce service, dont l'interface est, bien entendu, normalisée par l'OMG.

Ce service permet de représenter une hiérarchie de nommage par un graphe orienté similaire à un système de fichier Unix, comme représenté sur la figure 11.1 page suivante. Chaque objet est soit une feuille de l'arbre, comme l'objet `hosts`, soit un nœud (ou **contexte**) pouvant être accroché à un parent (indiqué par une flèche) et pouvant avoir des enfants.

Le nom d'un objet est toujours relatif à un contexte. Deux objets peuvent avoir le même nom s'ils sont référencés dans des contextes différents. À partir d'une racine fixée, il est possible de parcourir l'arbre afin de trouver le contexte ou l'objet final représenté par un nom. Par exemple, en supposant que le séparateur de noms soit le caractère \,

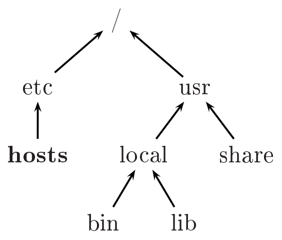


FIG. 11.1 – Hiérarchie du service de nommage

l'association {USR, "local\bin"} représente un objet (ici un contexte) de manière non ambiguë, si USR représente l'objet noté "usr" sur la figure. Un objet spécial, dont la référence est échangée par un mécanisme extérieur au service de nommage, est appelé **racine** de la hiérarchie de nommage (dans notre cas l'objet / du schéma).

Nous offrons dans GLADE un service de nommage fortement inspiré de celui normalisé par l'OMG, dans le but de faciliter la traduction de programmes utilisant CORBA en programmes utilisant l'annexe des systèmes répartis.

11.3 Exclusion mutuelle

Il existe dans le standard CORBA un service de gestion de concurrence très complet, dans lequel des primitives aussi complexes que l'intention de réserver une ressource existent. Pour GLADE, nous avons choisi d'offrir un service d'exclusion mutuelle plus simple, basé sur l'algorithme décrit par [Li et Hudak, 1986]. Cet algorithme est basé sur le fait que tout nœud pouvant être impliqué dans la négociation d'un verrou connaît un propriétaire probable pour ce verrou. Lorsqu'un nœud cède ou promet le verrou à un autre nœud, alors ce dernier devient le propriétaire probable pour le premier.

11.3.1 Présentation de l'algorithme

La figure 11.2 page suivante présente une instance de cet algorithme mettant en jeu cinq partitions P_1 à P_5 potentiellement intéressées par le verrou. Au tout début, P_1 possède le verrou et toutes les partitions le considèrent comme propriétaire probable de la ressource (étape 1). Les flèches représentent cette notion de probable possession de la ressource.

À l'étape 2, P_5 souhaite acquérir le verrou. Elle envoie donc une demande à P_1 (flèche en pointillés) et considère qu'elle-même est maintenant le propriétaire le plus probable du verrou. En effet, elle sait que P_1 le lui transmettra à un moment donné. Il est donc plus logique qu'une requête entrante doive être traitée par P_5 que par P_1.

Juste avant l'étape 3, P_1 a transmis le verrou à P_5. P_5 est devenu le propriétaire probable pour P_1, étant donné que c'est la dernière partition dont P_1 sait qu'elle a eu le verrou. P_2, P_3 et P_4, n'étant pas au courant de cet échange, considèrent toujours P_1 comme propriétaire probable de la ressource.

Lors de l'étape 4, P_4 envoie une demande à son propriétaire probable P_1 afin d'acquérir le verrou, et devient à ce moment son propre propriétaire probable, comme P_5 lors de

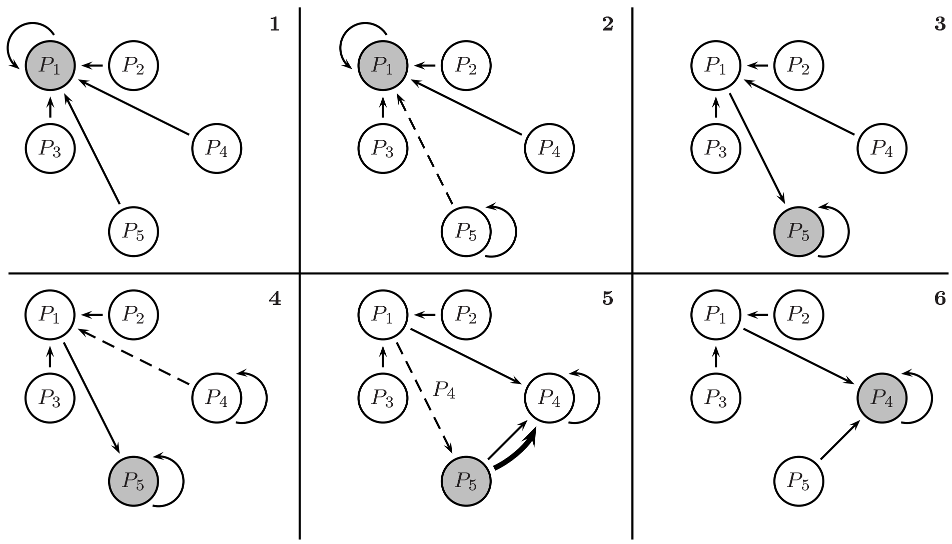


FIG. 11.2 – Algorithme de Li et Hudak

l'étape 2. P_1 n'étant pas, de son propre point de vue, le propriétaire du verrou, elle va transmettre la requête à P_5.

L'étape 5 représente l'état des partitions après que P_5 a reçu la demande de P_4 transmise par P_1 (flèche en pointillés surmontée du symbole P_4). À ce moment-là, P_5 a promis le verrou à P_4, situation modélisée par une flèche grasse. P_4 est également devenue le propriétaire probable de P_5 et P_1, car c'est la dernière partition dont on sait qu'elle obtiendra la ressource.

Pour finir, l'étape 6 montre la configuration du système lorsque P_4 a obtenu le verrou. Encore une fois, P_2 et P_3, n'ayant jamais participé à un échange, sont toujours dans leur configuration initiale.

On voit facilement, et [Li et Hudak, 1986] prouvent, que toutes les partitions pointent, directement ou indirectement, sur une partition dont on sait qu'elle possède ou qu'elle a demandé le verrou. Notamment, dans le cas de l'étape 5, une partition souhaitant acquérir le verrou le demandera directement à P_4, bien qu'elle ne l'ait pas encore acquis, plutôt qu'au propriétaire véritable P_5, qui de toute façon le cédera à P_4.

11.3.2 Implémentation comme service de GLADE

À chacun des nœuds représentés ci-dessus (P_1 à P_5) correspond un objet réparti Ada. La création d'un verrou utilise le service de nommage présenté précédemment ; le premier nœud qui crée un sémaphore d'un nom donné devient automatiquement son propriétaire probable pour toutes les partitions déclarant par la suite souhaiter utiliser ce verrou.

L'exemple 11.1 page suivante contient les principaux points de la déclaration du paquetage GLADE.Mutexes. L'utilisation d'un verrou réparti est extrêmement simple : il suffit, sur une partition, de créer un objet de type Mutex_Type et de l'initialiser avec un nom convenu *a priori*. Les opérations habituelles Acquire et Release, permettant respectivement de prendre et de relâcher le verrou sont également fournies.

Exemple 11.1 Exclusion mutuelle utilisant des objets répartis

```
[...clauses de dépendances...]
package GLADE.Mutexes is
   pragma Remote_Types;

   type Mutex_Type is tagged limited private;

   procedure Initialize
     (Mutex : access Mutex_Type;
      Name      : in String);
   --  Initialize a mutex given its name. The same name must be used
   --  to have several virtual instances of the mutex, which is quite
   --  useful.

   procedure Acquire (Mutex : access Mutex_Type);
   --  Get the mutex. This is a blocking operation since the mutex
   --  may be used by someone else at this time. Program_Error is raised if
   --  the mutex has not been initialized.

   procedure Release (Mutex : access Mutex_Type);
   --  Release the mutex. Program_Error is raised if no P operation
   --  has been performed on the mutex.

   function Acquired (Mutex : access Mutex_Type) return Boolean;
   pragma Inline (Acquired);
   --  Return True if the mutex has been acquired locally at the time of the
   --  call.

   type Mutex_Access is access all Mutex_Type'Class;

[...]
private
[...]

  type Mutex_Type is tagged limited record
      Is_Used        : Boolean := False;
      Probable_Owner : Mutex_Access;
      Promised_To    : Mutex_Access;
      Barrier        : Positive;
      Name           : String_Access;
      Index          : Positive;
   end record;

end GLADE.Mutexes;
```

Ce service a de plus un but didactique, en cela qu'il présente les multiples possibilités d'utilisation des objets répartis en Ada. Dans le cas présent, le type Mutex_Type possède lui-même dans sa structure interne deux pointeurs sur objets répartis correspondant au propriétaire probable et à l'instance de Mutex_Type a qui le verrou a été éventuellement promis.

11.4 Transmission de flux continu

Avec le développement des réseaux haut débit, il devient de plus en plus courant d'échanger des flux multimédia dans des applications réparties interactives. Malheureusement, la plupart des systèmes répartis actuels n'intègrent pas, dans leurs concepts, la notion de flux continu d'information. Les architectures basées sur les appels de sous-programmes ou de méthodes à distance ne sont pas adaptées au besoin d'applications souhaitant ouvrir un canal de communication permanent afin d'échanger des données à un rythme soutenu. De plus, les solutions basés sur des canaux unidirectionnels sont souvent limitées ; par exemple, il est possible d'utiliser des sockets BSD, mais celles-ci n'assurent pas la transformation des données nécessaire dans un système hétérogène.

Heureusement, une solution efficace et élégante existe, consistant à combiner trois caractéristiques d'Ada :

1. Les flux, présentés dans la section 2.2.3 page 44, peuvent être considérés comme des tampons avec conversion automatique des données en un format compréhensible par toutes les partitions dans un système hétérogène.
2. Les objets répartis permettent d'échanger des références entre partitions. Par chance, les types de flux Ada sont étiquetés, limités et privés, ce qui leur permet d'être utilisé comme cible d'un pointeur sur objet réparti.
3. La directive de compilation Asynchronous appliquée sur un pointeur sur objet réparti rend unidirectionnels tous les appels de méthodes qui peuvent l'être (c'est-à-dire les appels à des procédures n'ayant que des paramètres en entrée et aucun en sortie).

Nous avons donc créé un type de flux spécial dont l'écriture remplit un tampon et dont la lecture est bloquante jusqu'à ce que les données demandées soient disponibles. Ce type de flux, appelé Tunnel, est destiné à être créé sur la partition souhaitant recevoir des données à haut débit. De simples lectures, en utilisant les attributs dédiés Read ou Input, permettent d'obtenir des données qui ont été préalablement placées dans le flux par une partition distante, qui s'est contenté d'utiliser les attributs Write ou Output. L'appel à ces attributs utilise la procédure Write définie pour tous les types de flux, et est rendu unidirectionnel par l'application de la directive de compilation Asynchronous appliqué sur le type de pointeur distant utilisé pour désigner le flux sur la partition émettrice.

Les exemples 11.2 et 11.3 montrent la simplicité d'utilisation de cette technique.

Exemple 11.2 Écriture dans un flux continu

```
with GLADE.Remote_Streams; use GLADE.Remote_Streams;

procedure Sender is
   X : Remote_Stream := [...obtention de la référence...];
begin
   --  Envoie successivement tous les entiers non négatifs
   for I in Natural'Range loop
      Natural'Write (X, I);
   end loop;
end Sender;
```

Exemple 11.3 Lecture dans un flux continu

```
with Ada.Text_IO;   use Ada.Text_IO;
with GLADE.Tunnels; use GLADE.Tunnels;

package body Receiver is

   X : aliased Tunnel := [...création de l'objet tunnel...];

   procedure Receive is
   begin
      --  Affichage de tous les entiers reçus par le tunnel
      loop
         Put_Line (Natural'Image (Natural'Read (X'Access)));
      end loop;
   end Receive;

end Receiver;
```

L'émetteur récupère une référence sur le tunnel préalablement créé chez le récepteur, puis utilise cette référence pour écrire dans le flux l'ensemble des entiers non négatifs. De son côté, le récepteur, après avoir créé le tunnel et l'avoir enregistré auprès d'un serveur de noms (non représenté ici), se contente de lire les données au fur et à mesure de leur arrivée. On pourra noter que l'émetteur ne sait pas qu'il écrit dans un type de flux distant : l'aiguillage se fait à l'intérieur de l'implémentation des attributs Write ou Output, implémentation générée par le compilateur.

La création de ce service nous a amené à demander au comité de normalisation un correctif immédiat au manuel de référence Ada. En effet, sous sa forme actuelle, [RM, 1995, E.4.1(9)] ne permet pas à la directive de compilation Asynchronous de transformer les appels à la méthode Write du type Tunnel en appels unidirectionnels. En effet, les restrictions évoquées dans ce paragraphe de la norme s'appliquent à tort aux paramètres de contrôle de l'appel de méthode, ce qui n'est pas nécessaire, le contenu des objets répartis ne transitant jamais implicitement entre deux partitions.

11.5 Conclusion

La création de services réutilisables par les architectes d'application réparties permet d'augmenter la classe d'applications visées par l'annexe. Que ce soit des services classiques comme le nommage ou la réservation de ressource, ou des services émergents comme la distribution de flux multimédia, leur disponibilité immédiate peut conduire à choisir l'annexe des systèmes répartis plutôt qu'une solution extérieure au langage.

CINQUIÈME PARTIE

Analyse critique

Cette partie contient tout d'abord, dans le chapitre 12, une critique de l'annexe des systèmes répartis ainsi que des propositions d'amélioration. Le chapitre 13 présente une comparaison des systèmes répartis en Ada avec CORBA, le modèle le plus populaire d'objets répartis.

12

Critique du manuel de référence

Ce chapitre expose les reproches que nous avons été amené à formuler contre la définition actuelle de l'annexe des systèmes répartis d'Ada. Que ce soit en tant qu'architecte de GLADE ou en tant qu'utilisateur, nous avons parfois eu l'impression d'être bridé par des règles qui, selon nous, n'ont pas de raison d'être. Lorsque cela nous semble aider à la compréhension d'un argument ou d'une solution, nous décrivons la manière dont d'autres systèmes répartis ont résolu les difficultés que nous avons rencontrées.

La majorité de ces critiques porte sur le paquetage d'interface System.RPC. La déclaration de ce paquetage est reproduite dans l'annexe B page 191.

12.1 Découpage en couches

Dans cette section, nous découpons l'annexe des systèmes répartis en trois couches conceptuelles, présentées sur la figure 12.1. La couche de haut niveau concerne les aspects sémantiques de la répartition en Ada. La couche intermédiaire décrit l'interface entre le code généré par le compilateur et l'exécutif des systèmes répartis. Enfin, la couche de bas niveau représente les protocoles de communication utilisés entre les différentes partitions d'une application répartie.

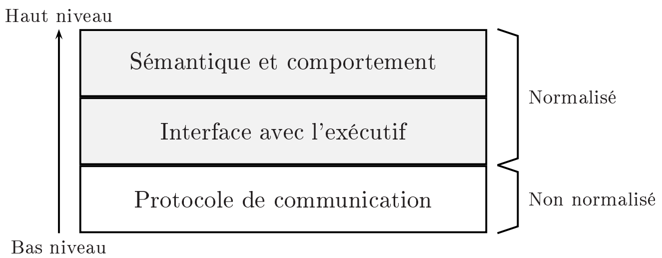


FIG. 12.1 – Couches d'une application répartie Ada

La couche de haut niveau doit nécessairement être normalisée. En effet, il est difficilement envisageable de développer une infrastructure pour applications réparties sans décrire le comportement au niveau sémantique des différentes constructions manipulables. Cette normalisation correspond, dans CORBA, au langage IDL, qui est très proche du modèle d'objets sous-jacent. En Ada, cette normalisation correspond au large sous-ensemble du langage utilisable en mode réparti.

La couche de niveau intermédiaire correspond à la définition dans le manuel de référence du paquetage System.RPC. Comme indiqué sur la figure 12.2, cette interface n'est pas normalisée dans CORBA. Cela signifie qu'une implémentation CORBA est libre d'utiliser l'interface qu'elle souhaite entre ses souches et squelettes et son exécutif, à condition qu'elle respecte la sémantique de l'IDL et qu'elle utilise le format de paquets normalisés par l'ISO (couche de bas niveau).

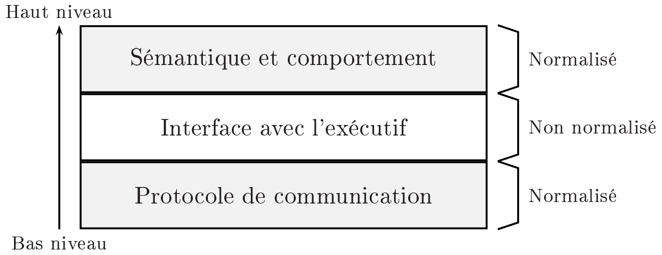


FIG. 12.2 – Couches d'une application répartie CORBA

Comme nous l'avons écrit dans la section 1.4.3 page 29, cette normalisation est partie d'une volonté d'interopérabilité : les concepteurs de l'annexe avaient pour but de permettre l'intégration de n'importe quel exécutif des systèmes répartis avec n'importe quel compilateur Ada implémentant la partie de la norme correspondant à la génération de souches et de squelettes. Malheureusement, cette vision utopique s'accommode mal des contraintes d'un système réparti moderne pour plusieurs raisons :

1. Comme évoqué dans la section 4.3 page 68, aucun mécanisme n'est présent dans le paquetage System.RPC pour obtenir l'identificateur de partition d'un service distant à partir de son nom. Il y a donc nécessairement un retour d'information de la part de l'outil de partitionnement, qui travaille dans ce cas avec une description statique de la structure de l'application répartie.
2. L'interface entre le compilateur et l'exécutif utilise des objets de type flux[1] qui sont équivalents à des tableaux d'octets non typés. Cela signifie que les différentes partitions d'un système réparti doivent utiliser des compilateurs qui supportent un même format de données, puisque le contenu du flux n'est pas normalisé. De plus, le codage des informations dans les flux n'est pas normalisé ; par exemple, un entier peut être stocké sur un nombre d'octets différents selon le compilateur et la plate-forme.

Le premier problème mentionné ci-dessus paraît *a priori* surmontable, mais occasionne en fait une rigidité nuisible au déploiement des systèmes répartis écrits en Ada : dans le cadre d'applications client-serveur, on peut souhaiter distribuer l'interface d'un serveur (la déclaration de ses paquetages Remote_Call_Interface et de leur fermeture sémantique) et laisser d'autres équipes écrire leurs propres clients de manière indépendante.

[1]Les flux sont décrits dans la section 2.2.3 page 44.

Cela suppose que, quelle que soit la manière dont ils décrivent leur application répartie (leurs clients peuvent en effet utiliser d'autres paquetages Remote_Call_Interface en plus de ceux fournis par la première équipe), l'identificateur des paquetages communs est identique. Cette contrainte est un frein considérable au développement de programmes répartis coopératifs.

Afin de nous affranchir de cet obstacle, nous avons ajouté, entre l'exécutif de GLADE et le code généré par GNAT, un nouveau paquetage d'interface. Ce paquetage, System.Partition_Interface, définit à un niveau sémantique élevé des opérations permettant de retrouver des identificateurs de partition à partir de noms de paquetages.

En ce qui concerne la couche de bas niveau, son absence de normalisation dans le manuel de référence Ada est regrettable sous deux aspects :

1. Aucune garantie n'est fournie concernant le fait que deux exécutifs différents seront capables de dialoguer.
2. Il n'est pas possible d'écrire, de manière portable, une interface avec d'autres systèmes répartis, car le format de données est *a priori* inconnu.

Cette démarche est à l'opposé de l'attitude adoptée par les autres systèmes répartis comme CORBA ou RMI, dans lesquels le protocole de communication et le format des données échangées sont normalisés. Cela garantit non seulement l'interopérabilité des diverses implémentations, mais aussi une compatibilité ascendante lors de changement de version d'un logiciel. En effet, le format de paquet étant normalisé, il ne peut pas être modifié d'une version à l'autre.

Il serait donc profitable d'apporter une amélioration à l'annexe des systèmes répartis à deux niveaux distincts :

1. La normalisation des protocoles de communication garantirait l'interopérabilité des différentes implémentations et fournirait la possibilité d'interfacer des services écrits en Ada depuis le monde extérieur.
2. La suppression de la normalisation de la couche intermédiaire permettrait aux architectes de l'exécutif des systèmes répartis et à ceux du compilateur de communiquer plus efficacement.

Le premier point peut être traité immédiatement, en proposant une définition des protocoles de communication à utiliser. Par exemple, l'utilisation de IIOP, le protocole de communication utilisé par CORBA au-dessus des protocoles TCP/IP, permettrait de faire coopérer des applications réparties écrites en Ada et des services utilisant CORBA.

Le second point en revanche nécessite une révision de la norme Ada. Son adoption condamnerait la possibilité d'utiliser n'importe quel exécutif des systèmes répartis sur n'importe quel compilateur, mais augmenterait probablement l'attrait des architectes de compilateurs Ada pour cette annexe, notamment si le premier point est adopté.

12.2 Synchronisation répartie

Une des formes de synchronisation explicite les plus utilisées en Ada 83 est la demande de rendez-vous. Dans un tel appel, une tâche demande à une autre d'accepter un rendez-vous afin qu'elles se synchronisent, avec ou sans échange de données.

Les paquetages Shared_Passive permettent, comme décrit dans la section 4.9 page 81, de disposer d'objets protégés[2] simples utilisables simultanément depuis plusieurs partitions d'une application répartie. Toutefois, il n'est pas possible d'effectuer de vrais appels de rendez-vous (synchronisation active) entre partitions.

L'introduction de rendez-vous distants dans une future révision de la norme permettrait d'étendre le modèle de programmation : en effet, il est possible de demander un rendez-vous à plusieurs tâches, et seul l'un d'entre eux sera accepté. Ceci a déjà été mise en œuvre dans des systèmes tels que celui décrit dans [Gammage et al., 1987] et [Kamel et Lo, 1990] ; cela nécessite l'utilisation de techniques de négociation répartie, techniques déjà utilisées lors du calcul de la terminaison globale d'une application Ada répartie (voir chapitre 8 page 119). Cela pose des problèmes théoriques liés aux différentes politiques d'ordonnancement des partitions d'une application répartie. Par exemple, comment ordonnancer les entrées dans une file d'attente lorsque les deux appelants et l'appelé ont trois politiques d'ordonnancement différentes ?

12.3 Gestion de l'identificateur de partition

En Ada, les partitions sont identifiées à l'aide d'un identificateur numérique unique. Cet identificateur, de type System.RPC.Partition_ID, peut être manipulé par le programmeur afin d'obtenir des renseignements sur une partition d'une application répartie [RM, 1995, E.1(16)]. Malheureusement, cette demande de renseignement, disponible à travers des sous-programmes fournis par l'exécutif, n'est pas portable par défaut de normalisation.

Le choix d'un type numérique non privé pour désigner une partition d'un système réparti est critiquable. En effet, l'utilisateur n'est pas sensé effectuer la moindre opération arithmétique sur un objet de ce type. Cependant, ce choix résulte du fait que l'attribut Partition_ID renvoie un nombre du type universal_integer [RM, 1995, E.1(9)].

Pour leur part, CORBA et RMI ne possèdent pas d'identificateur de partition à proprement parler. Ces systèmes ne manipulent que des références sur des objets distants, appelées IOR[3]. Dans ces références se trouvent codés à la fois le moyen de contacter un objet distant et le moyen de retrouver cet objet sur la partition distante, dans le cas où une partition gère plusieurs objets. Cette approche pourrait être adoptée dans le cadre de l'annexe des systèmes répartis, en utilisant un type d'identificateur normalisé pour désigner une entité distante, que ce soit un sous-programme appelable à distance ou un objet distant.

De plus, à partir d'une référence CORBA, il est possible d'obtenir des informations sur l'objet à partir d'un programme externe. Avec un identificateur de partition Ada, il n'est possible d'obtenir des informations que depuis l'intérieur de l'application répartie : cela interdit de procéder à une analyse *post mortem* d'une application erronée.

[2]Les objets protégés sont décrits dans la section 2.1.2 page 41.
[3]Interoperable Object Reference

12.4 Types de flux

Le code généré par le compilateur Ada utilise des flux pour transmettre à l'exécutif des systèmes répartis les données à destination d'une partition distante. Ces données, opaques pour l'exécutif, sont transmises par le compilateur par l'intermédiaire des procédures Do_RPC et Do_APC, et lui reviennent par l'intermédiaire de la procédure désignée par le pointeur enregistré à l'aide de Establish_RPC_Receiver.

12.4.1 Préparation des requêtes

Lorsque le compilateur souhaite générer une requête vers une partition distante, il emballe en premier lieu les paramètres à destination de cette partition dans un type de flux dédié, Params_Stream_Type. Ces flux possèdent un discriminant de type numérique, dont le but est de prédire lorsque c'est possible la taille des données qui seront ajoutées dans le flux par la suite, afin de pouvoir préallouer une zone en mémoire.

Le schéma traditionnel pour effectuer un appel de sous-programme distant est résumé dans l'exemple 12.1 (le traitement des exceptions est ignoré). Dans cet exemple, un flux contenant *a priori* 40 octets est créé pour la requête, et un de 20 octets pour la réponse.

Exemple 12.1 Simulation simplifiée d'un appel de sous-programme distant

```
[...]
   Requête : aliased Params_Stream_Type (40);
   Réponse : aliased Params_Stream_Type (20);
[...]
   Integer'Write (Requête'Access, X);
   Float'Write (Requête'Access, Y);
   Do_RPC (Partition, Requête'Access, Réponse'Access);
   Natural'Read (Réponse'Access, Z);
   return Z;
[...]
```

La présence de ce discriminant semble indiquer que les concepteurs de l'annexe des systèmes répartis ont choisi d'éviter, probablement pour des raisons d'efficacité, des copies de zones en mémoire inutiles qui seraient dues, en l'absence de ce paramètre, aux appels successifs à la procédure Write du flux. Cependant, cela ne prévient pas le fait que le compilateur doit construire la totalité de la requête avant de la transmettre à l'exécutif des systèmes répartis.

Supposons qu'à ce discriminant indiquant la taille du flux soit ajouté ou substitué un discriminant de type System.RPC.Partition_ID désignant la partition distante à laquelle cette requête est destinée. Cela permettrait à l'exécutif des systèmes répartis de transmettre les paramètres au destinataire au fur et à mesure de leur emballage, ou bien dès qu'un certain nombre d'octets sont présents dans le tampon d'envoi. En effet, les routines d'emballage ayant accès au flux ont également accès à ses discriminants ; dans le cas qui nous intéresse, cela signifie que ces routines savent, au moment de leur exécution, quelle partition est le destinataire final de la requête (voir exemple 12.2 page suivante). De plus, certains systèmes de communication comme ATM utilisent de petites cellules (53 octets), comme décrit dans [Labelle, 1999] ; l'envoi de données au fur et à mesure de leur emballage permettrait d'utiliser au mieux les capacités du réseau.

Exemple 12.2 Nouveau discriminant pour le type Params_Stream_Type

```
--  Dans la déclaration de System.RPC
[...]
   type Params_Stream_Type (Recipient : Partition_ID) is
     new Ada.Streams.Root_Stream_Type with private;
[...]

--  Dans le corps de System.RPC
[...]
   procedure Write
     (Stream : in out Params_Stream_Type;
      Data   : in     Ada.Streams.Stream_Element_Array)
   is
   begin
      --  Envoie les données directement au destinataire au lieu de
      --  les stocker dans un tampon.

      Envoi_Immédiat_Par_TCP (Destinataire => Stream.Recipient,
                              Données      => Data);
   end Write;
[...]
```

Cette idée de transmission à la volée est déjà utilisée dans les routeurs modernes, dans lesquels le début d'un paquet entrant est traité et aiguillé avant la réception de la totalité du paquet [Kamrad, 1998]. L'application de cette technique lors de la réception de requêtes entrantes permettrait également d'accélérer le traitement d'appels de sous-programmes dont les paramètres sont volumineux.

Ces affirmations sont corroborées par les mesures expérimentales présentées sur la figure 12.3 page suivante. Ce test, réalisé sur un Pentium II 400 sous Linux, consiste à faire effectuer un aller-retour à un tableau d'entiers de taille variable entre deux partitions. Tant que la taille de ce tableau est inférieure à 960 entiers, ce tableau, ainsi que les informations de contrôle, peuvent être contenus dans un tampon de la couche TCP, et sont donc envoyés d'un seul bloc. Lorsque le paquet atteint 960 entiers, le tampon d'émission est totalement rempli, et le second paquet ne peut être envoyé que progressivement au fur et à mesure que de l'espace noyau se libère. Nous observons donc une croissance linéaire jusqu'à environ 1200 entiers. À partir de 1200 entiers, la courbe se stabilise : le noyau a terminé d'envoyer le premier paquet, et le reste du second paquet est accepté rapidement.

L'adoption d'un modèle de communication “au vol” permettrait à l'exécutif de Glade d'envoyer le premier paquet TCP dès que des données suffisantes ont été placées dans le flux dédié à l'envoi. De plus, cela permettrait, du côté du récepteur, de réveiller plus tôt une tâche anonyme qui absorberait les paquets entrants au fur et à mesure. On peut noter une certaine similitude entre cette proposition et le service de propagation de flux continu présenté dans la section 11.4 page 157.

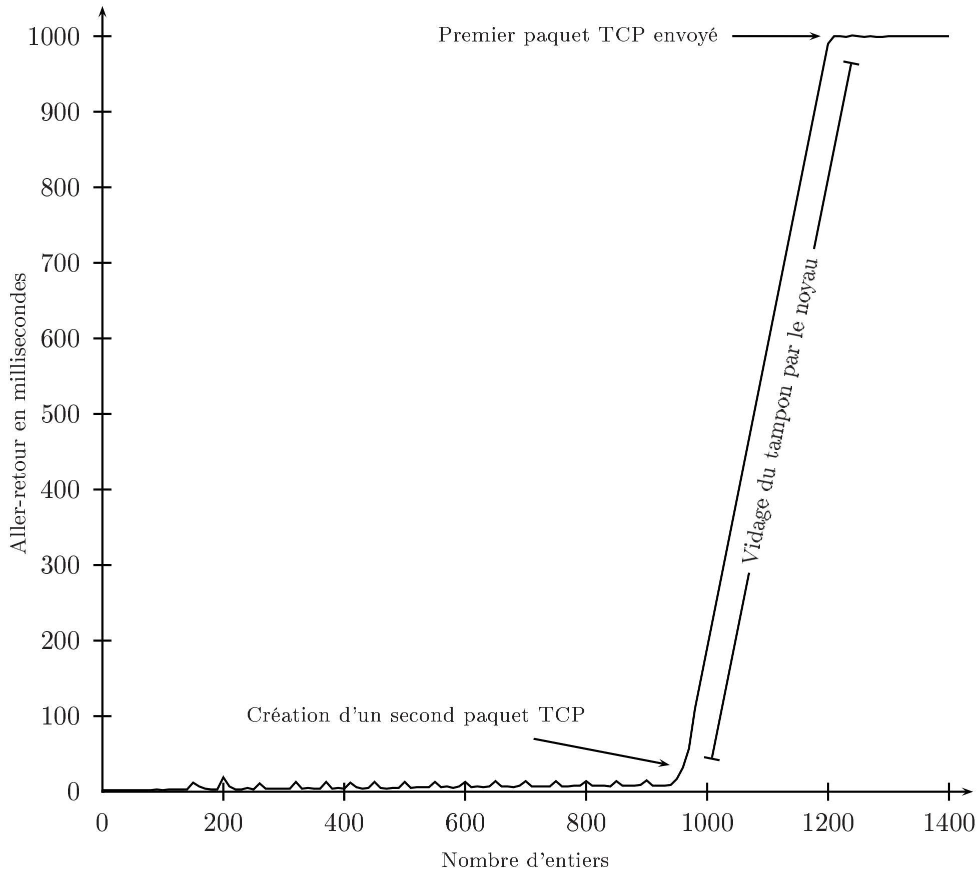


FIG. 12.3 – Engorgement des tampons de communication

12.4.2 Traitement de l'hétérogénéité

Une autre limitation liée aux flux provient de leur manque de normalisation. En effet, le format d'un flux et notamment son comportement vis-à-vis de systèmes différents n'étant pas défini, une implémentation de l'annexe des systèmes répartis en Ada peut ne pas supporter des architectures hétérogènes [RM, 1995, E.1(6)].

Bien que l'annexe définisse un type de flux dédié à la répartition, celui-ci ne peut pas bénéficier d'un encodage particulier des données qui lui sont passées. En effet, l'écriture d'une valeur dans un flux se fait par l'intermédiaire des attributs Write ou Output ; dans le traitement de ces attributs, il n'est pas prévu de cas spéciaux selon le type de flux utilisé. De plus, la recommandation de [RM, 1995, 13.13.2(17)] suggère que la représentation des données à l'intérieur d'un flux soit la même que la représentation en mémoire, ce qui va à l'encontre du souhait naturel de support de l'hétérogénéité présent dans toute architecture répartie.

Il serait donc bénéfique, si la transmission des paramètres se fait toujours par le biais de flux dans la nouvelle révision du langage, que la représentation des données à l'intérieur d'un flux soit normalisée pour l'ensemble des plates-formes. Dans le cas contraire, la présence d'une directive de compilation ou d'une clause de représentation permettant de choisir le format des données utilisées pour chaque type de flux serait appréciable.

12.5 Caractère dynamique d'une application répartie

Au fur et à mesure que les systèmes de communication et les réseaux à haut débit se développent, nous voyons apparaître des applications réparties impliquant un grand nombre de nœuds. Le problème de l'évolutivité de tels systèmes fonctionnant parfois en continu devient de plus en plus sensible. Il est difficile, lorsqu'une application répartie déployée à grande échelle ne doit être stoppée que pendant des laps de temps minimaux, de faire évoluer son code de manière transparente pour les autres partitions.

Les objets répartis aident à gérer ce genre de situations : il est possible d'ajouter dynamiquement à un système de nouveaux objets, plus spécialisés ou plus robustes que ceux déjà en place, et d'utiliser un mécanisme permettant aux nouveaux clients de l'application d'être dirigés automatiquement vers ces nouveaux objets. Toutefois, cela nécessite une connaissance *a priori* de l'interface de ces objets, c'est-à-dire de leur liste de méthodes et des types des arguments de ces méthodes.

CORBA permet de passer outre cette contrainte grâce à la notion d'appel dynamique. Il est possible, en interrogeant un dépôt d'interface (IR, "*Interface Repository*" en anglais), de récupérer la liste des méthodes associées à un nom de classe ainsi que leur signature. Ensuite, en utilisant un service d'appel dynamique d'interface (DII, "*Dynamic Interface Invocation*" en anglais), il est possible de construire une requête vers un objet de ce type en fournissant un par un les paramètres de la méthode appelée. Les paramètres de retour sont ensuite extraits de la même manière.

Un tel service pourrait être ajouté à l'annexe des systèmes répartis. Certains diront que cela va à l'encontre du typage fort et de la sécurité offerts par Ada, mais cela n'est en fait pas plus risqué que de s'interfacer avec un sous-programme écrit dans un langage externe. La difficulté en Ada proviendrait probablement du fait que certains types de paramètres, les pointeurs par exemple, provoquent parfois le passage de paramètres implicites supplémentaires, représentant par exemple leur niveau d'accessibilité[4] [RM, 1995, 3.10.2(1)]. Ceci peut être résolu en fournissant des attributs spéciaux permettant de lire et d'écrire ces types dans des requêtes dynamiques, similaires aux attributs Write et Read déjà existants.

12.6 Formulation de l'annexe des systèmes répartis

Comme le fait remarquer [Kermarrec, 1999], à la différence de la majeure partie des chapitres du manuel de référence Ada, la formulation de l'annexe est majoritairement négative. Où il pourrait être écrit "il est autorisé de faire X et Y", on trouve "il est interdit de faire Z". Cet aspect restrictif rebute, d'après l'auteur, un certain nombre d'utilisateurs potentiels qui ne voient en l'annexe E qu'une liste d'interdictions et de restrictions. Ceci est d'autant plus regrettable que peu d'exemples concrets de l'utilisation des systèmes répartis en Ada sont disponibles librement.

De plus, [Patton, 1998] regrette que cette annexe se concentre principalement sur l'interface entre le code généré par le compilateur et l'exécutif gérant la répartition, et considère que cela nuit à la clarté de l'exposé. Comme nous l'écrivons dans la section 12.1, nous soutenons cette critique ; l'annexe devrait se contenter de présenter les effets sémantiques induits par l'utilisation de la répartition ainsi que le protocole de bas niveau à utiliser.

12.7 Conclusion

L'annexe des systèmes répartis a été introduite dans la dernière révision de la norme Ada. Sa jeunesse, ainsi que son absence d'implémentation de référence complète au moment de la normalisation, ont causé tout naturellement l'introduction de maladresses dans sa définition.

Toutefois, nous espérons par nos suggestions parvenir, lors de la prochaine révision du langage, à la conception d'une annexe robuste dont le champ d'application couvrira encore plus de besoins. Grâce à cette première tentative, qui est pour nous un succès malgré ses défauts, nous avons acquis une solide connaissance des problèmes potentiels liés à l'intégration de la répartition au sein d'un langage de programmation.

[4]Le **niveau d'accessibilité** d'un type pointeur en Ada correspond au nombre d'environnements dans lesquels la déclaration de ce type est imbriquée. Le but est d'empêcher la copie incontrôlée d'un pointeur dont la cible est une variable locale vers un pointeur de type "plus global", afin d'éviter de manipuler des références pointant sur un objet qui a été détruit. Cela revient à écrire que la durée de l'objet pointé doit être au moins égale à celle du type du pointeur.

13

Comparaison d'Ada 95 et CORBA

13.1 Introduction

Ce chapitre établit une comparaison entre les concepts mis en œuvre pour la définition de l'architecture CORBA (présentée brièvement dans la section 1.5.2 page 35), et ceux de l'annexe des systèmes répartis. À l'heure actuelle, il est quasiment impossible de parler de systèmes répartis sans que CORBA ne soit mentionné ; cette popularité se retrouve dans la plupart des journaux informatiques destinés au grand public et aux entreprises, dans lesquels Ada n'apparaît que rarement.

De même que nous sommes intimement convaincus que le langage Ada devrait être utilisé beaucoup plus largement dans l'industrie et pour la formation des étudiants en informatique, nous estimons que l'annexe des systèmes répartis mériterait d'être déployée à plus grande échelle, malgré les critiques formulées dans le chapitre 12. Il nous semble inutile d'utiliser CORBA dans des environnements Ada purs, *a fortiori* dans un parc de machines homogène.

13.2 Concepts généraux

CORBA est un système entièrement basé sur les objets répartis. La programmation par objets, bien que fort répandue de nos jours, reste plus difficile à appréhender que la programmation procédurale. De plus, pour certains types de problèmes, l'utilisation d'une méthodologie orientée objet s'avère plus coûteuse qu'une méthodologie traditionnelle.

L'annexe des systèmes répartis d'Ada permet, au contraire, d'utiliser le même style de programmation pour une application répartie que pour une application monolithique. L'architecte est libre d'utiliser des objets et de les répartir sur différents nœuds, mais il peut également se contenter d'invoquer des sous-programmes à distance lorsque cela s'applique mieux au problème qu'il cherche à résoudre.

Considérons le cas concret d'une application dont les capacités de calcul lourd sont déportés sur une machine différente de celle gérant l'interface utilisateur. Les fonctions mathématiques sont typiquement représentées par des fonctions ou des procédures dans le langage de programmation cible. L'introduction d'objets répartis dont l'invocation des méthodes permet d'effectuer ces opérations ajoute à l'architecture générale une couche objet totalement artificielle.

Le "tout objet" est un effet de mode déjà ressenti lors de l'introduction du langage Java. Si l'architecture d'un programme peut, dans certains cas, être simplifiée par l'utilisation d'objets et d'aiguillage dynamique, l'utilisation d'objets de manière systématique là où ils ne sont pas nécessaires nuit autant à la compréhension d'un système complexe qu'à son analyse statique. En effet, la présence de l'aiguillage dynamique ne permet pas toujours de prédire quel sous-programme sera appelé lors d'un appel de méthode. C'est pourquoi il est en général interdit lors de l'utilisation de sous-ensembles restreints d'Ada destinés à augmenter (encore) la sécurité du langage.

[Patton, 1998] a cherché à déterminer jusqu'à quel niveau une spécification de service pouvait descendre sans faire de choix définitif entre l'annexe des systèmes répartis et CORBA, dans le cadre d'une application écrite entièrement en Ada. Il montre que le fait de considérer CORBA comme une alternative possible force les architectes à organiser la conception autour des acteurs plutôt qu'autour des fonctionnalités de l'application. Tout au long de l'analyse, les choix sont dictés principalement par CORBA, même si au final la répartition au niveau du langage est utilisée.

13.3 Interopérabilité

Ce point de la comparaison est traité dans la section 12.1 page 161. Il est évident que l'annexe des systèmes répartis ne peut pas entrer en compétition avec CORBA sur le plan de l'interopérabilité, une implémentation de l'annexe des systèmes répartis n'ayant *a priori* aucune raison d'être compatible avec une autre.

13.4 Description d'interface

Le contrat entre un service implémenté à l'aide d'un objet CORBA (le fournisseur) et le nœud qui l'utilise (le client) est contenu dans une déclaration d'interface. Cette interface, écrite dans le langage IDL, est indépendante des langages utilisés par les protagonistes. Elle décrit les types utilisés et les méthodes applicables à chaque objet à l'aide d'une syntaxe proche de celle du C++.

Une des grandes forces de CORBA est sa capacité à faire s'interfacer différents langages de programmation à travers les objets répartis : une interface IDL peut être traduite dans tous les langages cibles supportant CORBA. Ceci a, comme conséquence directe, de limiter les constructions supportées par l'IDL à l'intersection des constructions supportées par les langages cible. Par exemple, il est impossible de décrire un type entier dont on spécifierait les bornes ; le seul choix possible se résume à un certain nombre de types entiers prédéfinis.

Inversement, l'annexe des systèmes répartis d'Ada permet d'utiliser une grande majorité des constructions disponibles dans un programme Ada traditionnel pour décrire l'interface d'un service distant. Si quelques restrictions empêchent l'utilisation sans précautions de types dont la sémantique est principalement locale (les types pointeurs par exemple), il est toujours possible pour le programmeur de décrire la manière dont ces types sont transportés (voir la section 4.7 page 79). Les sous-types, les types possédant un discriminant et les agrégats complexes peuvent être utilisés pleinement lors d'appels de services distants. Le contrat de typage Ada est respecté que l'appel soit local ou distant.

De plus, la répartition en Ada ne nécessite pas l'utilisation d'un autre langage pour décrire l'interface des services utilisables à distance : leur déclaration telle qu'utilisée dans des programmes monolithiques suffit à décrire pleinement la manière dont ils peuvent être invoqués. Comme le souligne [Riley, 1996], cela aplanit considérablement la courbe d'apprentissage nécessaire à la maîtrise de la répartition.

13.5 Lisibilité

Les types CORBA utilisables dans une interface IDL étant normalisés, leurs traductions dans un langage cible ne correspondent généralement pas à des types de base de ce langage. Par exemple, le développement d'une partie d'une application CORBA en Ada nécessite l'utilisation du type "chaîne de caractères" CORBA.String au lieu du type standard String. Cela entraîne souvent la génération d'interfaces peu lisibles comparés à leur équivalent utilisant l'annexe des systèmes répartis.

Considérons à titre d'exemple un service permettant d'imprimer une chaîne de caractères sur un terminal distant. Une interface IDL pour ce service est présentée dans l'extrait 13.1 page suivante. Un exemple de code Ada généré pour cette interface par le produit Broca est présenté dans l'extrait 13.2.

Tout d'abord, on peut noter l'introduction d'un type d'objet Ref étendant le type standard CORBA.Object.Ref. Ce type, dont la seule utilité est de permettre la localisation d'une instance du service, est accompagné de deux fonctions de conversion. La présence de ces entités est systématique et requise lors de la génération du code Ada correspondant à une interface IDL, et n'améliore en rien l'apparence du code source. On remarque ensuite la présence d'un paramètre supplémentaire de type Ref dans la signature de la procédure Put_String : il sert à dénoter une instance particulière du service. Ceci, en plus de l'utilisation du type CORBA.String au lieu du type String, indique qu'une répartition *a posteriori* d'une application Ada monolithique utilisant CORBA nécessitera la réécriture des appels de services, ou l'écriture d'une couche intermédiaire masquant l'utilisation de CORBA.

En comparaison, l'exemple 13.3 présente l'équivalent de ce service en utilisant l'annexe des systèmes répartis. On notera que ce service n'est présent qu'une seule fois dans l'application répartie, les paquetages catégorisés comme Remote_Call_Interface n'étant situés que sur une seule partition.

13.6 Efficacité

Nous ne comparons pas ici l'efficacité réelle des implémentations de l'annexe des systèmes répartis et de CORBA, mais l'efficacité théorique. En effet, les modèles adoptés par chaque système imposent certaines limitations dont il est impossible de s'affranchir.

Exemple 13.1 Définition IDL de l'interface hello

```
interface hello {
   void put_string (in string s);
};
```

Exemple 13.2 Code généré à partir de l'interface hello

```
with CORBA;
with CORBA.Object;
pragma Elaborate_All (CORBA);

package hello is

   type Ref is new CORBA.Object.Ref with null record;

   function Unchecked_To_Ref
     (Self : in CORBA.Object.Ref'Class)
      return Ref;

   function To_Ref
     (Self : in CORBA.Object.Ref'Class)
      return Ref;

   procedure put_string
     (Self : in Ref;
      s : in CORBA.String);

   Repository_Id : constant CORBA.String
      := CORBA.To_CORBA_String ("IDL:hello:1.0");

   function Is_A
     (Self   : Ref;
      Logical_Type_Id : CORBA.String)
      return CORBA.Boolean;

end hello;
```

Exemple 13.3 Équivalent du service hello en utilisant la répartition Ada

```
package Hello is

   pragma Remote_Call_Interface;

   procedure Put_String (S : in String);

end Hello;
```

13.6.1 Appel d'objet distant

Lors d'un appel d'objet distant, une implémentation CORBA est libre de construire la requête "au vol", c'est-à-dire de transmettre les paramètres d'un appel de méthode au fur et à mesure de leur conversion dans le format normalisé utilisé entre les nœuds. Lorsque les paramètres ont une taille conséquente, cela permet non seulement de pouvoir envoyer des paquets sur le réseau alors que les paramètres suivants ne sont pas encore prêts, mais cela permet également au nœud destinataire d'aiguiller les paramètres entrants vers les routines de décodage appropriées au fur et à mesure de leur arrivée.

L'annexe des systèmes répartis ne permet pas d'adopter ce mécanisme. Nous montrons dans la section 12.4.1 page 165 que cela pose des problèmes de dégradation de performances dès lors que les données devant être échangées entre deux partitions dépassent la taille du tampon alloué à la communication par le noyau du système d'exploitation sous-jacent. Il est donc clair qu'une implémentation tirant partie de la possibilité offerte par CORBA de transmettre les paramètres avant la construction totale de la requête sera plus efficace qu'une implémentation de l'annexe des systèmes répartis.

13.6.2 Appel d'objet local

Les performances potentielles sont inversés lors d'un appel d'objet local, c'est-à-dire d'un objet se trouvant sur la même partition. En effet, le modèle CORBA fait transiter l'ensemble des requêtes par l'exécutif CORBA (ORB), que l'objet soit local ou distant.

En revanche, comme il est mentionné dans la section 4.5 page 72, il est possible en Ada de traiter un objet local différemment d'un objet réparti. Cela signifie qu'aucun surcoût n'est introduit du fait de la définition d'un pointeur sur objet distant. Lors de l'utilisation d'applications manipulant un grand nombre d'objets pouvant être accédés aussi bien localement qu'à distance, l'annexe des systèmes répartis est donc potentiellement plus efficace.

13.7 Intégration des services

La plupart des fonctionnalités normalisées de CORBA le sont à travers des services additionnels qui viennent se greffer sur le cœur du système. Le service de nommage par exemple, évoqué dans la section 11.2 page 153, n'est pas obligatoirement utilisé dans une application CORBA. Dans ce cas, les références initiales d'objets doivent être échangés par des mécanismes externes à l'architecture de répartition.

L'annexe des systèmes répartis n'intègre pas de service de nommage à proprement parler, mais est capable de localiser un certain nombre de paquetages : ceux catégorisés à l'aide de la directive de compilation Remote_Call_Interface. Ces paquetages permettent aisément l'échange de références d'objets par un mécanisme d'enregistrement et de restitution.

Tous les autres services additionnels définis par la norme CORBA peuvent être également implémentés à l'aide de la répartition en Ada. En effet, ces services sont basés sur des objets répartis et n'impliquent pas de traitement hors norme.

13.8 Disponibilité

Un des points négatifs de l'annexe des systèmes répartis en Ada est son manque de variété. Seules deux implémentations sont disponibles à l'heure actuelle, toutes deux pour le compilateur GNAT :

Glade : notre implémentation, décrite dans la section 3.2 page 65, est complète et validée selon la suite de tests ACVC 2.1. Elle est considérée *de facto* comme implémentation de référence, et de nombreux travaux d'expérimentation l'utilisent comme base pour tester de nouvelles technologies ou de nouvelles extensions.

Adept : cette implémentation est basée sur une ancienne version de Glade et est maintenant entièrement gérée par l'université TAMU (Texas, USA). Elle est malheureusement incomplète, et n'est ni validée, ni validable dans l'état actuel. Elle a été modifiée afin de servir de passerelle entre l'annexe des systèmes répartis et le système de répartition RMI utilisé par Java.

CORBA, en revanche, est largement répandu. Plusieurs produits commerciaux supportent la génération de code pour Ada : Orbix, de IONA, OrbExpress, de OIS, OrbAda, de Top Graph'X, et Broca de l'Enst. Notons que ces deux derniers logiciels sont écrits entièrement en Ada.

13.9 Conclusion

L'annexe des systèmes répartis et CORBA ont tous deux leurs qualités et leurs défauts. Il est impossible de dire que l'un est meilleur que l'autre dans l'absolu sans entrer dans les détails de l'application cible.

Lorsque l'interopérabilité entre différents langages de programmation est primordiale, le choix de CORBA s'impose naturellement. En revanche, pour le développement d'applications réparties en Ada pour lesquelles il est important de préserver la sécurité inhérente au langage, le choix de l'annexe des systèmes répartis l'emporte.

Conclusion et perspectives

Conclusion

Cette thèse a présenté les différentes étapes nécessaires pour passer d'un code source représentant un programme réparti Ada en l'application répartie elle-même. Les trois phases consistent en la transformation du code à l'intérieur du compilateur, l'interfaçage avec l'exécutif des systèmes répartis et l'utilisation des outils et services adéquats pour construire l'application finale.

Que la répartition soit désirée pour des raisons de performances ou de tolérance aux pannes, il est possible d'utiliser plusieurs techniques différentes pour la réalisation concrète. L'envoi de message, technologie la plus spartiate, est en général délaissée au profit des appels de sous-programmes distants. Ces appels peuvent être enrichis sémantiquement en étendant leur comportement aux constructions avancées du langage, comme par exemple les exceptions ou les types *a priori* non transportables. Des techniques plus avancées comme les objets répartis permettent d'intégrer plus facilement la répartition dans une modélisation centrée sur les acteurs plutôt que sur les fonctionnalités.

Le langage Ada 95 est tout à fait adapté à la répartition d'applications de par son annexe des systèmes répartis. À la différence de systèmes tels que CORBA ou RMI dédiés spécifiquement à la répartition, la norme Ada intègre ce concept dans le cœur même du langage. Il n'est pas nécessaire pour l'utilisateur d'apprendre de nouvelles constructions syntaxiques et sémantiques, car l'adjonction de directives de compilation suffit à morceler une application monolithique en un ensemble d'entités distribuables sur des nœuds virtuels, appelés partitions.

Ces directives de compilation permettent d'organiser les paquetages selon les fonctions qu'ils auront à accomplir. Les paquetages Remote_Types contiennent les définitions des types qui seront échangés entre les partitions, les paquetages Remote_Call_Interface servent de contenant pour les sous-programmes appelables à distance et les paquetages Shared_Passive représentent une mémoire virtuelle partagée et répartie.

Au commencement de cette thèse, le compilateur libre GNAT générait déjà les souches et les squelettes des services répartis sous forme de code Ada. Cette méthode ne permettait pas d'utiliser certaines constructions nécessaires pour la génération d'objets répartis notamment à cause des conflits de nommage entre le code généré et le code de l'utilisateur. Nous avons entièrement réécrit les modules gérant l'expansion des constructions relatives à la répartition. Ces modules travaillent maintenant au niveau de l'arbre sémantique décoré interne au compilateur et ne produisent plus de fichiers

Ada intermédiaires.

Lors de cette réécriture, nous avons également repensé le modèle des objets répartis afin d'éviter des problèmes de construction liés au modèle précédent. Nous avons de plus introduit un nouveau paquetage d'interfaçage, System.Partition_Interface, permettant de faire communiquer de manière bien spécifiée la partie relative à la localisation et au nommage de l'exécutif des systèmes répartis avec le code généré par le compilateur. Cela nous a permis d'ajouter une grande dynamicité dans la gestion de la localisation des différents services en ne se limitant pas au paquetage prédéfini System.RPC.

Par la suite, nous avons implémenté un modèle de pointeurs sur sous-programmes distants tel que défini dans l'annexe des systèmes répartis. Cette implémentation fonctionne parfaitement mais ne nous satisfait pas totalement au niveau des performances. Nous avons donc proposé un nouveau modèle permettant de représenter de telles références par le biais de pointeurs sur objets distants, modèle qui devra être implémenté par la suite.

Nous avons choisi de traiter le cas de la directive All_Calls_Remote de manière directe, en mélangeant dans le code généré les souches et les squelettes d'un paquetage Remote_Call_Interface et en adaptant les appels de sous-programmes distants pour tenir compte de ce type de constructions. Même si nous doutons de l'utilité de cette fonctionnalité, cela nous était indispensable afin d'obtenir une implémentation conforme au manuel de référence et passant tous les tests de validation.

En ce qui concerne la directive de catégorisation Shared_Passive, une solution originale a été choisie utilisant le système de fichiers répartis sous-jacent, par exemple NFS ou CODA. Cette solution a l'avantage d'offrir des possibilités de persistance à toute application utilisant de tels paquetages, qu'elle soit monolithique ou répartie.

L'ensemble de ce code généré s'interface, à travers les deux paquetages décrits plus haut System.RPC et System.Partition_Interface, avec l'exécutif des systèmes répartis. Cet exécutif, construit de façon très modulaire, gère à la fois les protocoles de bas-niveau permettant la communication entre partitions et les fonctionnalités de haut-niveau telles que le service de nommage. Les partitions communiquent en général deux à deux dans le cadre d'appel de sous-programmes et méthodes distants, mais certaines partitions particulières, appelées "points d'accueil", appartiennent à un anneau virtuel permettant de répliquer les informations de localisation afin de résister à une panne d'un des nœuds.

Cette localisation des services a donné lieu à deux versions successives. Tout d'abord, une architecture centralisée permettait d'enregistrer et de localiser des partitions ainsi que des paquetages Remote_Call_Interface et Shared_Passive. Cependant, cela présentait l'inconvenient de créer un point sensible dans l'application répartie : si la partition gérant le service de nommage subissait une panne, l'ensemble du programme réparti devenait aveugle et ne pouvait plus créer de liens entre deux partitions, faute de pouvoir les localiser.

La seconde phase a consisté en l'ajout d'un anneau virtuel permettant de répliquer, sur des points d'accueil multiples, toutes les données de contrôle. Des protocoles de communication de groupe ont été étudiés et adaptés à la configuration d'une application répartie écrite en Ada afin d'obtenir un service qui soit à la fois efficace et cohérent. C'est ainsi que la négociation d'un identificateur de partition s'obtient en un tour d'anneau tandis que l'enregistrement d'une unité nécessite deux tours afin d'éviter des conflits potentiels en cas de déclarations quasi-simultanées du même paquetage par plusieurs partitions.

Les appels de sous-programmes et méthodes distants interviennent après la localisation des services. Pour ce faire, nous avons développé un protocole de communication masquant les différences entre les différents nœuds, possiblement hétérogènes, de l'application répartie. Afin d'augmenter l'efficacité de ces appels, nous avons conçu et implémenté un réservoir de tâches permettant d'éviter le surcoût lié à la création dynamique de tâches lors du traitement des appels entrants. Ce réservoir, dont les caractéristiques sont entièrement paramétrables par l'utilisateur, permet de réutiliser une même tâche lors d'appels entrants successifs.

Dans le même esprit, l'avortement d'une requête distante est aussitôt propagé à la partition destinataire de l'appel afin de ne pas gaspiller des ressources du côté de l'appelé. Ceci n'est pas obligatoire d'après le manuel de référence, mais permet de minimiser les différences de comportement entre les appels locaux et les appels distants. Notre protocole permet d'effectuer cet avortement distant de manière efficace, par le biais d'un unique message unidirectionnel.

Un autre problème que nous avons eu à traiter lors de la conception de cet exécutif est celui de la terminaison répartie. Il est facile de terminer une application monolithique à partir de l'observation de l'état des différentes tâches du système à un instant donné. Dans une application répartie, l'absence d'horloge globale interdit ce type d'observation ponctuelle. Nous avons tout d'abord conçu un algorithme permettant de détecter l'absence d'activité à un instant donné dans le programme réparti. Cependant, cet algorithme ne traitait pas le cas des messages en transit dont l'arrivée chez leur destinataire, prêt à terminer précedemment, pouvait générer une nouvelle activité et donc contredire la terminaison globale apparemment acquise de l'application répartie.

Nous avons donc conçu et implémenté un algorithme original, basé sur des propagations de vagues successives, permettant de confirmer l'absence de messages en transit ainsi que l'absence d'activité sur les nœuds du système. En l'absence de pannes, cet algorithme détecte la terminaison globale en au plus deux vagues successives. Dans certaines configurations particulières, une seule vague suffit à engendrer la terminaison.

De même que la génération de code à l'intérieur du compilateur, l'exécutif des systèmes répartis est conforme à la norme et a été validé officiellement sur plusieurs plate-formes. Nous avons toutefois souhaité aller plus loin, et avons ajouté des extensions permettant d'augmenter la portée potentielle de l'annexe des systèmes répartis, dans les limites autorisées par le manuel de référence.

Tout d'abord, à partir d'observations des autres types de systèmes répartis, nous avons étendu le modèle de terminaison en y ajoutant la possibilité de développer des applications client-serveur. En effet, la terminaison globale telle que présentée ci-dessus ne permet pas la terminaison des clients avant celle des serveurs, comme on le souhaite généralement pour cette classe d'applications. Nous avons donc implémenté trois modes de terminaison distincts : le mode Global_Termination, correspond à la description faite précédemment, le mode Local_Termination, autorise la terminaison d'un client dès lors qu'il devient inactif, et le mode Deferred_Termination permet la création de partitions dont la durée de vie est potentiellement infinie. La combinaison de ces trois modes permet de développer potentiellement tous les types de systèmes répartis rencontré traditionnellement.

Il nous a également semblé nécessaire d'ajouter la possibilité pour les architectes de construire des systèmes répartis tolérants aux pannes. Pour cela, nous avons offert la possibilité de choisir la politique à adopter en cas de disparition prématurée d'une partition et des unités qu'elle supportait. Le programmeur peut là également opter pour trois modes distincts. Le mode Rejected_On_Restart, choix par défaut, permet de considérer toute disparition d'une partition comme définitive. Le mode Fail_Until_Restart supporte la réapparition d'unités disparues, mais considère toute référence à ces unités comme des erreurs tant qu'elles n'ont pas refait surface. Le dernier mode, Wait_Until_Restart, permet de masquer complètement l'erreur au programmeur qui utilise un service en bloquant les appels dans l'exécutif jusqu'à ce que l'unité concernée ait refait son apparition.

Une autre extension utile que nous avons ajouté consiste en l'utilisation de filtres permettant de coder et de décoder de manière transparente pour l'utilisateur les données échangées entre les partitions. Cela permet par exemple de respecter une politique de sécurité définie à l'intérieur d'une entreprise en utilisant chiffrement et authentification, ou d'utiliser des liens à bas-débit en compressant au vol l'ensemble des informations échangées.

Nous avons également été confrontés au problème du déverminage d'une application répartie. En présence de nombreuses partitions, il n'est pas toujours aisé de rejouer une application exactement de la même manière lorsque l'on cherche à identifier un problème survenu lors d'une exécution précédente. Pour pallier ce défaut, nous avons conçu et implémenté la possibilité d'enregistrer les communications d'une partition avec le reste du système et de simuler une nouvelle exécution de cette partition, conforme à la précédente, en l'absence des autres acteurs de l'application répartie.

L'exécutif léger, qui est la dernière extension que nous avons réalisée, consiste en la possibilité pour un client n'offrant aucun service propre de réduire son empreinte mémoire et sa consommation de temps de calcul. En effet, un client monotâche et ne créant pas d'objet pouvant être interrogé à distance a un comportement purement séquentiel, ne nécessitant pas la présence de tous les modules de l'exécutif des systèmes répartis. Par exemple, un tel client n'a pas besoin d'embarquer les différents algorithmes de terminaison, étant donné qu'il ne peut pas être sollicité spontanément par d'autres partitions. Sa terminaison est donc obligatoirement locale. Dans le même ordre d'idées, toutes les unités gérant le réservoir de tâches sont également inutiles, puisqu'aucune requête entrante ne pourra jamais avoir lieu.

Au delà des extensions de l'exécutif lui-même, nous avons également souhaité ajouter des services externes utiles pour le développement d'applications réparties. Par exemple, un service de nommage de haut-niveau utilisable par l'utilisateur permet à celui-ci de désigner symboliquement les différentes ressources partagées de son système. Un mécanisme d'exclusion mutuelle totalement réparti permet de gérer les ressources dont l'accès doit être sérialisé sans créer de point central susceptible de subir une panne. Finalement, un nouveau service permettant de transmettre un flux de données de manière continue permet à un architecte de transmettre des flux multimedia de manière efficace sans recourir à des communications de bas-niveau.

Toutes ces réalisations ont été faites au vu de l'état de l'art et des autres types de systèmes répartis. Lorsqu'il n'était pas satisfaisant d'intégrer des algorithmes existants, ou lorsque cette intégration risquait de se révéler inefficace, nous avons développé de nouveaux algorithmes nous permettant d'atteindre notre but. Ces différentes techniques ont donné lieu à de nombreuses publications dans des conférences internationales.

Perspectives

Cette thèse ne saurait toutefois représenter une conclusion définitive dans le domaine des applications réparties en pur Ada, mais constitue plutôt une étape dans un long processus progressif. En effet, notre équipe poursuit activement sa recherche dans ce domaine et espère apporter dans les mois et années qui viennent de nouvelles idées et de nouveaux résultats, notamment dans le domaine de la coopération avec les autres modèles de systèmes répartis comme CORBA et dans celui de la tolérance aux pannes.

Thomas Quinot va notamment continuer, à l'occasion de sa thèse co-encadrée à l'ENST et à Paris VI, le projet Ciao [Quinot, 1999] : un service réparti décrit en Ada à l'aide des directives de compilation Remote_Call_Interface et Remote_Types est traduit en un service CORBA, en langage IDL. Un relais est automatiquement généré, permettant d'accéder, depuis CORBA, aux fonctionnalités offertes par Ada.

Notre équipe développe également un ORB appelé Broca [Gingold, 1999], qui partagera à terme ses primitives de communication avec l'exécutif de Glade. Nous souhaitons ainsi obtenir une solution mixte, permettant à des services CORBA d'accéder de manière intuitive à des partitions Ada et *vice-versa*.

Dans un autre domaine, Ada Core Technologies est sur le point de distribuer la première version de JGNAT, un compilateur générant du pseudo-code Java à partir de code source Ada [Comar et al., 1997]. Ce nouvel arrivant nous permettra d'orienter nos recherches dans deux directions successives :

1. Nous prévoyons, en collaboration avec Ada Core Technologies, d'adapter Glade pour cet environnement. Cela nous permettra d'offrir une nouvelle solution souple et originale apportant aux architectes d'applications réparties la possibilité de combiner un serveur performant, compilé sous forme de code objet natif, avec des clients extrêmement portables, disponibles par exemple sous forme d'appliquettes (“*applets*” en anglais).
2. Une fois cette technologie disponible, il sera aisé de construire des applications en langage Java accédant aux services de Glade à travers l'implémentation évoquée ci-dessus, faisant ainsi disparaître la frontière entre les mondes Ada et Java.

Ces projets, qui ne constituent que la partie visible de l'iceberg, montrent la vivacité des systèmes répartis en Ada et l'énorme potentiel de recherche les entourant. Nous ne pouvons que nous en réjouir.

Annexes

A

Réseaux de Petri colorés

La plupart des notions présentées ici sont extraites de [Brams, 1983].

A.1 Réseaux de Petri

A.1.1 Généralités

Un réseau de Petri est un quadruplet $R = \langle P, T, W^+, W^- \rangle$ dans lequel P représente l'ensemble des places, T l'ensemble des transitions, et W^+ et W^- respectivement les applications d'incidence avants et arrières de $P \times T$ dans $\mathbb{N}$. Chaque place d'un réseau de Petri contient zéro, un ou plusieurs jetons, jetons qui sont consommées lors du franchissement de transitions. Un jeton est habituellement noté $\langle \epsilon \rangle$, $\langle e \rangle$ ou $\bullet$.

Le marquage d'un réseau de Petri est une application de P dans $\mathbb{N}$, consistant donc à associer à chaque place du réseau un nombre de jetons donné. Pour chaque place p et chaque transition t, l'application d'incidence arrière $W^-(p,t)$ représente le nombre minimum de jetons devant se trouver dans la place p pour rendre la transition t franchissable. De la même manière, $W^+(p,t)$ représente le nombre de jetons qui seront ajoutés à la place p lors du franchissement de la transition t.

Une transition t est franchissable pour une marque M (noté $M[t >$) si et seulement si pour toute place p du réseau, le nombre de jetons présents dans p, noté $M(p)$ est au moins égal à $W^-(p,t)$:

$$M[t > \Leftrightarrow \forall p \in P, M(p) \geqslant W^-(p,t)$$

Après le franchissement de la transition t, le nouveau marquage M' est défini comme :

$$\forall p \in P, M'(p) = M(p) - W^-(p,t) + W^+(p,t)$$

Le fait que la transition t fasse passer de la marque M à la marque M' se note $M[t > M'$.

A.1.2 Représentation

Un réseau de Petri peut être représenté par deux matrices correspondant aux applications d'incidence avant et arrière W^+ et W^- ; il est cependant plus commode de le représenter à l'aide d'un graphe orienté biparti comprenant des arcs valués. L'ensemble des sommets du graphe est constitué de $P \cup T$ (places et transitions), et les arcs sont représentées par la fonction Γ, tandis que leur valeur est spécifiée par la fonction V, définies par :

$$\begin{gathered}\forall p \in P, \Gamma(p) = \{t \in T \mid W^-(p,t) > 0\} \\ \forall t \in T, \Gamma(t) = \{p \in P \mid W^+(p,t) > 0\} \\ \forall p \in P, \forall t \in T, V(p,t) = W^-(p,t) \text{ et } V(t,p) = W^+(p,t)\end{gathered}$$

On dit alors que le graphe $G = \langle P, T, \Gamma, V \rangle$ est associé au réseau $R = \langle P, T, W^-, W^+ \rangle$. La fonction d'incidence inverse de Γ notée Γ^{-1} est définie simplement par :

$$\begin{aligned}\forall t \in T, \Gamma^{-1}(t) &= \left\{p \in P \mid W^-(p,t) > 0\right\} \\ \forall p \in P, \Gamma^{-1}(p) &= \left\{t \in T \mid W^+(p,t) > 0\right\}\end{aligned}$$

Il est également commode de représenter graphiquement un réseau de Petri. La figure A.1 contient un réseau composé de deux places p_1 et p_2 ainsi que d'une transition t. La valeur de valuation des arcs par défaut est de 1. Le marquage initial M_0 consiste en une unique marque dans la place p_1. La transition t est toujours franchissable, résultant en une accumulation de jetons dans la place p_2.

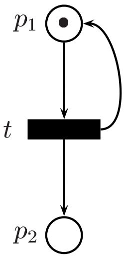


FIG. A.1 – Réseau de Petri simple

A.1.3 Conflits dans les réseaux de Petri

Deux transitions t_1 et t_2 sont dites en conflit structurel si et seulement si elles ont au moins une place commune en entrée, c'est-à-dire si

$$\exists p \mid W^-(p,t_1).W^-(p,t_2) \neq 0$$

Elles sont de plus en conflit effectif pour un marquage M si, en plus d'être en conflit structurel, les deux transitions ne peuvent pas êtes franchies, ce qui se résume à :

$$M[t_1 >, M[t_2 > \text{ et } \exists p \mid M(p) < W^-(p,t_1) + W^-(p,t_2)$$

A.1.4 Marquages accessibles

La liste des marquages accessibles d'un réseau $R = \langle P, T, W^+, W^- \rangle$ depuis un marquage M_0 est notée $Acc(R, M_0)$. On peut la définir comme :

$$M \in Acc(R, M_0) \Leftrightarrow \left\{ \begin{array}{ll} & M = M_0 \\ \text{ou} & \exists M', t \in T \mid M' \in Acc(R, M_0) \text{ et } M'[t > M \end{array} \right.$$

A.1.5 Vivacité d'un réseau

La quasi-vivacité d'une transition t représente le fait que, depuis un marquage initial M_0, il existe un marquage accessible rendant la transition t franchissable :

$$\exists M \in Acc(R, M_0) \mid M[t >$$

Par extension, le réseau marqué $\langle R, M_0 \rangle$ est dit quasi-vivant si toutes ses transitions sont quasi-vivantes. Cette propriété de quasi-vivacité est primordiale dans la simplification des réseaux de Petri, car une transition qui n'est pas quasi-vivante peut être éliminée du réseau.

De la même manière, une transition t est vivante si à partir d'un marquage initial M_0 il existe toujours un marquage accessible rendant la transition t franchissable :

$$\forall M \in Acc(R, M_0), \exists M' \in Acc(R, M) \mid M'[t >$$

Par extension, le réseau marqué $\langle R, M_0 \rangle$ est dit vivant si toutes ses transitions sont vivantes.

A.2 Réseaux de Petri colorés

L'expressivité considérable des réseaux de Petri rencontre cependant des limites lorsqu'il s'agit de modéliser des systèmes complexes réels. Afin de simplifier leur écriture et leur analyse, une extension couramment utilisée est la notion de réseaux de Petri colorés [Jensen, 1991, Genrich et al., 1979, Jensen, 1981]. Ces réseaux de haut niveau ont été notamment utilisés pour la validation de protocoles de communication [Girault, 1980].

A.2.1 Définition

Soit un ensemble fini de couleurs $C = \{c_1, c_2, \ldots, c_n\}$. Le marquage d'un réseau coloré est défini comme une fonction de l'ensemble des places du réseau vers des applications de C vers l'ensemble des entiers naturels :

$$M : P \rightarrow \{C \rightarrow \mathbb{N}\}$$

Cela signifie que $M(p)(c_i)$ désigne le nombre d'éléments de couleur c_i présents dans la place p. La valuation des arcs se fait au moyen de fonctions de couleurs et non plus d'entiers. La présence sur plusieurs arcs parvenant à une transition de plusieurs valuations de la couleur c_i nécessite que cette couleur soit présente dans toutes les places en aval de cette transition pour rendre cette dernère franchissable. Cette condition représente l'équivalent d'une jointure dans le domaine des bases de données.

A.2.2 Exemple

Considérons à titre d'exemple le réseau coloré représenté figure 7.1 page 108. La transition "*send job request*" ne devient franchissable qu'à partir du moment où l'on trouve dans la place "*job*" un ensemble $c_1 = \langle id, q\rangle$ dont le champ id de la couleur c_1 est égal à la couleur c_2 présente dans la place "*unused jobs*". Cela illustre également le fait que les couleurs ne correspondent pas toutes à des valeurs du même type. En effet, nous avons les deux couleurs c_1 et c_2 définies comme suit :

$$\begin{aligned} c_1 &= \langle id_1, q_1\rangle \\ c_2 &= \langle id_1\rangle \end{aligned}$$

A.2.3 Réseaux hiérarchiques

En dépit de sa simplicité, un modèle de programme basé sur un réseau de Petri coloré peut être difficile à comprendre par un humain lorsque le nombre de places et de transitions est trop élevé. Pour cette raison, il est possible d'utiliser des réseaux hiérarchiques, dans lesquels une ou plusieurs transitions sont représentées comme des aggrégats, aggrégats décrits sur un autre schéma.

Un exemple est donné sur les figures 7.2 page 109 et 7.3 page 110 ; la première figure représente une vision hiérarchisée (par l'intermédiaire des places grisées) du réseau présenté sur la seconde.

A.3 Utilisation des réseaux colorés à partir de programmes Ada

L'application des réseaux de Petri colorés aux programmes concurrents écrits en Ada n'est pas nouvelle, mais connaît à ce jour un nouvel intérêt. [Shatz et al., 1996] et [Bruneton et Pradat-Peyre, 1999] ont par exemple montré de quelle manière un programme concurrent Ada pouvait être traduit de manière automatique en réseau de Petri coloré, réseau qui est par la suite réduit automatiquement et analysé. L'avancée de ces travaux est malheureusement insuffisante à ce jour pour envisager d'appliquer cette traduction automatique à l'ensemble de l'exécutif de GLADE.

Procédant de la démarche opposée, [Kaiser et Pradat-Peyre, 1997] ont présenté une méthodologie permettant, à partir d'une spécification sous forme de réseau de Petri coloré prouvée correcte, de programmer la solution d'un problème bien connu (celui des philosophes) de manière à garantir l'absence de verrou mortel. Cette analyse a ensuite été étendue par GARCÍA et VILLARROEL aux réseaux de Petri temporels [García et Villarroel, 1999].

A.4 Pratique et simulation

Il existe un grand nombre d'outils permettant de réduire et d'analyser certaines propriétés d'un réseau de Petri. Une des possibilités offertes par ces outils est de montrer les différents chemins d'exécution (processus) empruntés par une instance de ce réseau.

Les figures A.2 et A.3 illustrent un cas décrit dans la section 7.1.2 page 109 simulé à l'aide de **viptool** [Freytag, 1999]. La première figure montre que la requête correspondant à l'identificateur 1 a été avortée, mais que le résultat correspondant à cette requête est tout de même parvenu à la partition. Étant donné qu'aucune requête en cours n'était plus associée avec l'identificateur 1, le résultat a été ignoré.

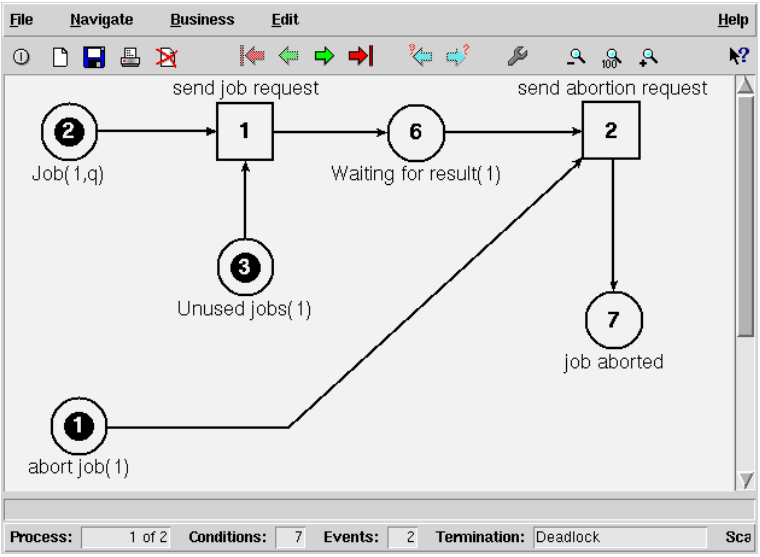


FIG. A.2 – Requête avortée

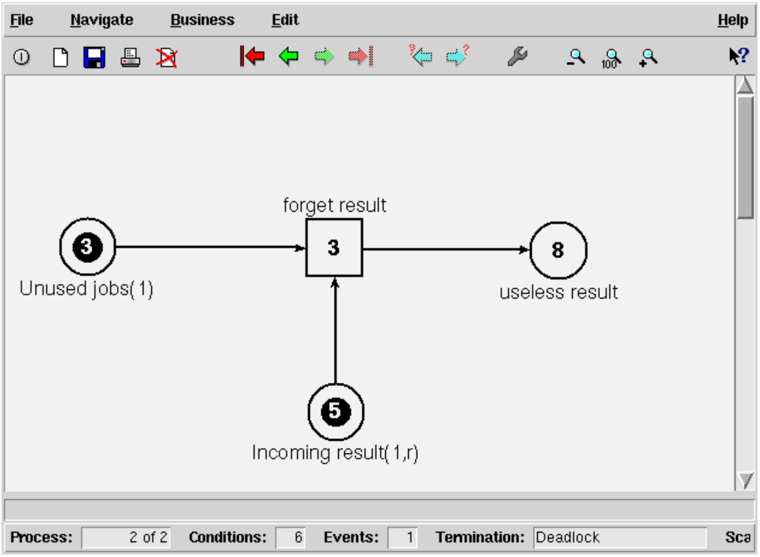


FIG. A.3 – Résultat arrivant trop tard

B

Déclaration de System.RPC

Exemple B.1 Déclaration de System.RPC

```
with Ada.Streams;
package System.RPC is

   type Partition_ID is range 0 .. implementation-defined;

   Communication_Error : exception;

   type Params_Stream_Type
     (Initial_Size : Ada.Streams.Stream_Element_Count)
   is new Ada.Streams.Root_Stream_Type with private;

   procedure Read
     (Stream : in out Params_Stream_Type;
      Item   : out Ada.Streams.Stream_Element_Array;
      Last   : out Ada.Streams.Stream_Element_Offset);

   procedure Write
     (Stream : in out Params_Stream_Type;
      Item : in Ada.Streams.Stream_Element_Array);

   procedure Do_RPC
     (Partition  : in Partition_ID;
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);

   procedure Do_APC
     (Partition  : in Partition_ID;
      Params     : access Params_Stream_Type);

   type RPC_Receiver is access procedure
     (Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);

   procedure Establish_RPC_Receiver
     (Partition : in Partition_ID;
      Receiver  : in RPC_Receiver);

private
   ... -- not specified by the language
end System.RPC;
```

C

GNAT et sa suite d'outils

Le tableau suivant décrit brièvement la liste d'outils présents dans GNAT 3.12. À l'exception de gnat1, appelé automatiquement par gcc, tous ces utilitaires sont disponibles pour l'utilisateur.

Outil	Fonction
gcc	Aiguille vers les autres programmes
gnat	Aiguille vers les autres programmes
gnat1	Compile un fichier source (appelé par un aiguilleur)
gnatbind	Vérifie la cohérence et choisit un ordre d'élaboration
gnatchop	Sépare des fichiers source concaténés
gnatelim	Élimine les sous-programmes non utilisés
gnatfind	Localise les définitions de symboles
gnathtml	Génère du HTML à partir de sources Ada
gnatkr	Tronque de manière cohérente le nom des fichiers source
gnatlbr	Gère les bibliothèques Ada
gnatlink	Appelle l'éditeur de liens du système avec les bons paramètres
gnatls	Liste les dépendances d'une unité
gnatmake	Compile toutes les unités devant être compilées ou recompilées
gnatmem	Détecte les fuites de mémoire dans les programmes Ada
gnatprep	Substitue des symboles dans des fichiers source
gnatpsta	Imprime la déclaration du paquetage Standard
gnatstub	Génère des corps compilables à partir de spécifications
gnatpsys	Imprime la déclaration du paquetage System
gnatxref	Imprime les références croisées

Bibliographie

[Adleman et al., 1978] L. Adleman, R. L. Rivest et A. Shamir (1978). A Method for Obtaining Digital Signature and Public-Key Cryptosystems. *Communication of the ACM*, 21(2).

[Aho et al., 1989] A. Aho, R. Sethi et J. Ullman (1989). *COMPILATEURS – Principes, techniques et outils*. InterEditions.

[Armitage et Chelini, 1985] J. W. Armitage et J. V. Chelini (1985). Ada software on distributed targets : a survey of approaches. *ACM SIGADA Ada Letters*, 4(4) : 32–37.

[Armstrong et al., 1998] J. Armstrong, R. Virding, C. Wikström et M. Williams (1998). *Concurrent Programming in Erlang*. Prentice Hall.

[Atkinson, 1991] C. Atkinson (1991). *Object Oriented Reuse, Concurrency and Distribution – an Ada based approach*. Addison Wesley.

[Atkinson et al., 1990] C. Atkinson, A. di Maio et R. Bayan (1990). Dragoon : an object-oriented notation supporting the reuse and distribution of Ada software. *ACM SIGADA Ada Letters*, 10(9) : 50–59.

[Atkinson et Goldsack, 1989] C. Atkinson et S. Goldsack (1989). An object oriented approach to Virtual Nodes – Are Package Types an answer. Paru dans *Proceedings of the 3rd international Workshop on Real Time Ada Issues*.

[Balter et al., 1991] J. Balter, J.-P. Banâtre et S. Krakowiak, coordinateurs (1991). *Construction des systèmes d'exploitation répartis*. Collection Didactique. INRIA.

[Bamberger et al., 1988] J. Bamberger, T. Coddington, R. Firth, D. Klein, D. Stinchcomb, R. V. Scoy et C. Colket (1988). Distributed Ada Real-Time Kernel. Rapport technique CMU/SEI-88-TR-17 ADA199482, Software Engineering Institute (Carnegie Mellon University).

[Birrell et Nelson, 1984] A. D. Birrell et B. J. Nelson (1984). Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1) : 39–59.

[Bishop et al., 1987] J. M. Bishop, S. R. Adams et D. J. Pritchard (1987). Distributing Concurrent Ada Programs by Source Translation. *Software Practice and Experience*, 17(12) : 859–884.

[Boussand et al., 1992] C. Boussand, E. Cavillon et F. Kordon (1992). TEPACAP : Transformation Et réPartition d'une Application Codée automatiquement en Ada Pour unix. Rapport technique, IBP/MASI. Rapport 92/22.

[Brams, 1983] G. Brams (1983). *Réseaux de Petri : Théorie et Pratique*. Masson.

[Bruneton et Pradat-Peyre, 1999] É. Bruneton et J.-F. Pradat-Peyre (1999). Automatic Verification of Concurrent Ada Programs. Paru dans *Proceedings of AdaEurope'99*, Santander, Espagne. Springer-Verlag.

[Brzezinski et al., 1993] J. Brzezinski, J.-M. Hélary et M. Raynal (1993). Distributed Termination Detection : General Model and Algorithms. Rapport technique BROADCAST#TR93-05, ESPRIT Basic Research Project BROADCAST.

[Carey, 1988] S. Carey (1988). Distributed Ada Real-Time Kernel (DARK) Technology Transition Plan. Technical Memos SEI-88-TP-1, Software Engineering Institute (Carnegie Mellon University).

[Chandy et Lamport, 1985] K. M. Chandy et L. Lamport (1985). Distributed Snapshots : Determining Global States of Distributed Systems. *TOCS*, 3(1) : 63–75.

[Clemmensen, 1982] G. B. Clemmensen (1982). A Formal Model of Distributed Ada Tasking. Paru dans *1982 AdaTEC Conference on Ada*, pages 224–237, New York, NY. ACM SIGAda.

[Comar et al., 1997] C. Comar, G. Dismukes et F. Gasperoni (1997). Targeting GNAT to the Java Virtual Machine. Paru dans ACM, coordinateur, *Proceedings of the TRI-Ada'97 Conference, November 9–13, 1997, St. Louis, MO*, pages 149–164, New York, NY 10036, USA. ACM Press.

[Comar et al., 1994] C. Comar, F. Gasperoni et E. Schonberg (1994). The GNAT Project : a GNU-Ada9X Compiler. Rapport technique, New-York University.

[Corbin, 1993] J. Corbin (1993). *The Network File System For System Administrators*. Sun Microsystems, Inc., Mountain View, Californie, USA.

[Cornhill, 1983] D. Cornhill (1983). A Survivable Distributed Computing System for Embedded Applications Programs Writtein in Ada. *Ada Letters*, III(3) : 79–87.

[Coulouris et al., 1994] G. Coulouris, J. Dollimore et T. Kindberg (1994). *Distributed Systems, Concepts and Design*. Addison-Wesley, seconde édition.

[Dahl et Nygaard, 1966] O.-J. Dahl et K. Nygaard (1966). SIMULA- An Algol Based Simulation Language. *Communications of the ACM*, 9(9) : 671–678.

[Dapra et al., 1984] A. Dapra, S. Gatti, S. Crespi-Reghizzi, F. Maderna, D. Belcredi, A. Natali, R. A. Stammers et M. D. Tedd (1984). Using Ada and APSE to Support Distributed Multimicroprocessor Targets. *Ada Letters*, III(6) : 57–65.

[Davis, 1998] J. Davis (1998). The Power is Out There. *Business 2.0*, pages 102–104.

[Dijkstra et Scholten, 1980] E. W. Dijkstra et C. S. Scholten (1980). Termination detection for diffusing computations. *Information Processing Letters*, 11(1) : 1–4.

[Dobbing, 1990] B. Dobbing (1990). Distributed Ada : A Suggested Solution for Ada 9X. *ACM SIGADA Ada Letters*, 10(9) : 94–102.

[Dolev, 1982] D. Dolev (1982). The Byzantine Generals Strike Again. *Journal of Algorithms*, 3 : 14–30.

[Dupouy et al., 1997] B. Dupouy, O. Hainque, L. Pautet et S. Tardieu (1997). The SPIF Project. *Lecture Notes in Computer Science*, 1251.

[Feit, 1993] S. Feit (1993). *TCP/IP : architecture, protocols, and implementation*. McGraw-Hill.

[Francez, 1980a] N. Francez (1980a). Distributed Termination. *ACM Transactions on Programming Languages and Systems*, 2(1) : 42–55. Voir également l'erratum [Francez, 1980b] et les remarques [Mohan, 1981, Francez, 1981].

[Francez, 1980b] N. Francez (1980b). Erratum : "Distributed Termination". *ACM Transactions on Programming Languages and Systems*, 2(3) : 463–463. Voir [Francez, 1980a, Mohan, 1981, Francez, 1981].

[Francez, 1981] N. Francez (1981). Technical Correspondence : Reply from Francez. *ACM Transactions on Programming Languages and Systems*, 3(1) : 112–113. Voir [Francez, 1980a, Mohan, 1981].

[Freytag, 1999] T. Freytag (1999). *VIPTOOL manual*. Institut AIFB, University of Karlsruhe, Karlsruhe, Germany. `http ://www.aifb.uni-karlsruhe.de/InfoSys/VIP/viptool/`.

[Gammage et al., 1987] N. D. Gammage, R. F. Kamel et L. M. Casey (1987). Remote Rendezvous. *Software Practice and Experience*, 17(10) : 741–755.

[García et Villarroel, 1999] F. García et J. Villarroel (1999). Translating Time Petri Net Structures into Ada 95 Statements. Paru dans *Proceedings of AdaEurope'99*, Santander, Espagne. Springer-Verlag.

[Gargaro et al., 1992] A. Gargaro, S. J. Goldsack, C. Goldthorpe, D. Ostermiller, P. Rogers et R. A. Volz (1992). Towards Distributed Systems in Ada 9X. Paru dans *Proceedings of the Conference for Industry, Academia and Government*, pages 49–54, New York, NY, USA. ACM Press.

[Gargaro et al., 1996] A. Gargaro, Y. Kermarrec, L. Pautet et S. Tardieu (1996). PARIS — Partitioned Ada for Remotely Invoked Services. *Lecture Notes in Computer Science*, 1031.

[Gargaro et al., 1990] A. B. Gargaro, S. J. Goldsack, R. K. Power, R. A. Volz et A. J. Wellings (1990). Adapting Ada for Distribution and Fault Tolerance. *ACM SIGADA Ada Letters*, 10(9) : 111–117.

[Geist et al., 1994] A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Mancheck et V. S. Sunderam (1994). *PVM : Parallel Virtual Machine*. MIT Press.

[Genrich et al., 1979] H. Genrich, K. Lautenbach et P. Thiagarajan (1979). Elements of General Net Theory. Paru dans *Proceedings of the Advanced Course on General Net Theory of Processes and Systems*, Hambourg, Allemagne.

[Gingold, 1999] T. Gingold (1999). BROCA : an Ada Object Request Broker. Master's thesis, École Nationale Supérieure des Télécommunications.

[Girault, 1980] C. Girault (1980). Proof of protocols in case of failures. Paru dans *Advanced course on parallel processing*, University of Technology, Loughborough.

[Goldsack, 1992] S. Goldsack (1992). Panel on Distributed Systems. Paru dans *Proceedings of the Tri-Ada'92 Conference*. Steve Goldsack indique, lors de cette table ronde, qu'il a probablement été le premier à évoquer la possibilité d'intégrer la répartition au langage Ada dans un article publié en 1980. Malheureusement, il nous a été impossible de retrouver cette référence. Steve Goldsack lui-même n'a pas été en mesure de nous la fournir.

[Hadzilacos et Toueg, 1993] V. Hadzilacos et S. Toueg (1993). Fault-Tolerant Broadcasts and Related Problems. Paru dans S. Mullender, coordinateur, *Distributed Systems*. Addison-Wesley, seconde édition.

[Hutcheon et Wellings, 1988] A. D. Hutcheon et A. J. Wellings (1988). Supporting Ada in a distributed environment. *ACM SIGADA Ada Letters*, 8(7) : 113–117.

[Intermetrics, 1995] Intermetrics, coordinateur (1995). *Ada 95 Rationale : The Language, The Standard Libraries*.

[Javasoft Java RMI Team, 1997] Javasoft Java RMI Team (1997). *Java Remote Method Invocation Specification*. Sun Microsystems.

[Jensen, 1981] K. Jensen (1981). Coloured Petri nets and the invariant method. *T.C.S.*, 14 : 317–336.

[Jensen, 1991] K. Jensen (1991). Coloured Petri Nets : A High Level Language for System Design. Paru dans K. Jensen et G. Rozenberg, coordinateurs, *High-level Petri Nets*. Springer-Verlag.

[Jessop, 1982] W. H. Jessop (1982). Ada packages and distributed systems. *ACM SIGPLAN Notices*, 17(2) : 28–36.

[Jha et al., 1989a] R. Jha, G. Eisenhauer, J. M. Kamrad, II et D. Cornhill (1989a). An implementation supporting distributed execution of partitioned Ada programs. *ACM SIGADA Ada Letters*, 9(1) : 147–160.

[Jha et al., 1989b] R. Jha, J. M. Kamrad et D. T. Cornhill (1989b). Ada Program Partitioning Language : A Notation for Distributing Ada Programs. *IEEE Transactions on Software Engineering*, 15(3) : 271–280.

[Jones, 1998] C. Jones (1998). Software Productivity Research – Quality and Productivity. *Crosstalk*.

[Kaiser et Pradat-Peyre, 1997] C. Kaiser et J.-F. Pradat-Peyre (1997). Comparing the Reliability Provided by Tasks or Protected Objects for Implementing a Resource Allocation Service. Paru dans *Proceedings of Tri-Ada'97*, Saint-Louis, Missouri, USA.

[Kamel et Lo, 1990] R. Kamel et S. L. Lo (1990). Design Issues in the Implementation of Remote Rendezvous. Paru dans *10th International Conference on Distributed Computing Systems*, pages 245–252, Paris (France). IEEE, IEEE Computer Society Press.

[Kamrad, 1998] M. Kamrad (1998). Ada Experience Report for BlazeNet, Inc. *ACM SIGADA Ada Letters*, 18(6) : 215–216.

[Kamrad et al., 1987] M. Kamrad, R. Jha, G. Eisenhauer et D. Cornhill (1987). Distributed Ada. *ACM SIGADA Ada Letters*, 7(6) : 113–115.

[Kermarrec, 1999] Y. Kermarrec (1999). CORBA vs. Ada 95 DSA – A programmer's view. Paru dans *Proceedings of SigAda'99*, Redondo Beach, California, USA.

[Kermarrec et Pautet, 1992] Y. Kermarrec et L. Pautet (1992). Ada Communication Components for Distributed and Real-Time Application. Paru dans *Proceedings of the Conference for Industry, Academia and Government*, pages 530–537, New York, NY, USA. ACM Press.

[Kermarrec et Pautet, 1993] Y. Kermarrec et L. Pautet (1993). A Distributed Shared Virtual Memory for Ada 83 and Ada 9X Applications. Paru dans C. B. Engle, Jr., coordinateur, *Proceedings of the Conference on TRI-Ada*, pages 242–252, Seattle, WA, USA. ACM Press.

[Kermarrec et al., 1995] Y. Kermarrec, L. Pautet, G. Smith, S. Tardieu, R. Theriault et R. Volz (1995). Ada 95 Distribution Annex Implementation for GNAT. Rapport technique. (Contract with a grant from Computer Sciences Corp.).

[Kesavan, 1998] R. Kesavan (1998). *Communication Mechanisms and Algorithms for Supporting Scalable Collective Communication on Parallel Systems*. Thèse de Doctorat, The Ohio State University.

[Kienzle et al., 1996] J. Kienzle, T. Wolf et A. Strohmeier (1996). Secure Communication in Distributed Ada. Paru dans A. Strohmeier, coordinateur, *Reliable Software Technologies - Ada-Europe'96 : proceedings, Ada-Europe International Conference* *on Reliable Software Technologies Montreux, Switzerland, June 10-14 1996*, volume 1088 of *LNCS (Lecture Notes in Computer Science)*, pages 198–210. Springer-Verlag. Également disponible sous forme de rapport technique (EPFL-DI No 96/173).

[Knight et Rouleau, 1988] J. C. Knight et M. E. Rouleau (1988). A new approach to fault tolerance in distributed Ada programs. *ACM SIGADA Ada Letters*, 8(7) : 123–126.

[Kordon et Sens, 1991] F. Kordon et P. Sens (1991). Répartir des programmes Ada sur un ensemble homogène de machines Unix, une expérience de réalisation. Paru dans *Actes des Journées Internationales Francophones Ada-France : "Ada, premier bilan d'utilisation"*, pages 123–135.

[Koster, 1971] C. H. A. Koster (1971). Affix Grammars. Paru dans J. E. L. Peck, coordinateur, *Algol 68 Implementation*, pages 95–109. North-Holland, Amsterdam.

[Krishnan et al., 1988] P. Krishnan, R. A. Volz et R. J. Theriault (1988). Implementation of task types in distributed Ada. *ACM SIGADA Ada Letters*, 8(7) : 104–107.

[Labelle, 1999] R. Labelle (1999). Interface ATM pour Ada. Master's thesis.

[LeBlanc et Mellor-Crummey, 1986] T. J. LeBlanc et J. M. Mellor-Crummey (1986). Debugging Parallel Programs with Instant Replay. Technical Report TR194, University of Rochester, Computer Science Department. [Same as Leblanc2].

[Levrouw et Audenaert, 1993] L. J. Levrouw et K. M. R. Audenaert (1993). Reducing the space requirements of Instant Replay. Paru dans *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 205–207, San Diego, California. [Extended abstract].

[Li et Hudak, 1986] K. Li et P. R. Hudak (1986). Memory Coherence in Shared Virtual Memory Systems. Paru dans *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY. ACM.

[Lindholm et Yellin, 1996] T. Lindholm et F. Yellin (1996). *The Java Virtual Machine Specification*. Addison Wesley.

[Litzkow et al., 1988] M. Litzkow, M. Livny et M. M. Condor (1988). A hunter of idle workstations. Paru dans *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, Californie, USA.

[Maio et al., 1989] A. D. Maio, C. Cardigno, R. Bayan, C. Destombes et C. Atkinson (1989). DRAGOON : An Ada-based Object Oriented Language for Concurrent, Real-Time, Distributed Systems. Paru dans *Ada-Europe International Conference*, Madrid.

[McDowell et Helmbold, 1989] C. E. McDowell et D. P. Helmbold (1989). Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4) : 593–622.

[Memmi et Vautherin, 1991] G. Memmi et J. Vautherin (1991). Analysing Nets by the Invariant Method, Theory and Application. Paru dans K. Jensen et G. Rozenberg, coordinateurs, *High-level Petri Nets*. Springer-Verlag.

[Meyer, 1988] B. Meyer (1988). *Object-oriented Software Construction*. Series in Computer Science. Prentice Hall International.

[Millet et Baker, 1998] L. Millet et T. Baker (1998). Porting the GNAT Tasking Runtime to the Java Virtual Machine. Paru dans *Proceedings of AdaEurope'98*.

[Miranda et al., 1996] F. J. Miranda, A. Alvarez, S. Arévalo et F. J. Guerra (1996). Drago : An Ada Extension to Program Fault-Tolerant Distributed Applications. Paru dans A. Strohmeier, coordinateur, *Proceedings of AdaEurope'96*, Montreux, Suisse. Springer-Verlag.

[Miranda et al., 1999] J. Miranda, F. Guerra, J. Martín et A. González (1999). How to Modify the GNAT Frontend to Experiment with Ada Extensions. Paru dans *Proceedings of AdaEurope'99*, Santander, Espagne. Springer-Verlag.

[Mohan, 1981] C. Mohan (1981). Technical Correspondence : On Francez's "Distributed Termination". *ACM Transactions on Programming Languages and Systems*, 3(1) : 112–112. Voir [Francez, 1980a, Francez, 1981].

[MPI, 1995] MPI (1995). *MPI : A Message-Passing Interface Standard*. Message Passing Interface Forum.

[MPI, 1997] MPI (1997). *MPI-2 : Extensions to the Message-Passing Interface*. Message Passing Interface Forum.

[Nana Tchamnda, 1996] L. Nana Tchamnda (1996). *Tolérance aux fautes et applications distribuées*. Thèse de Doctorat, Université de Rennes I.

[National Bureau of Standards, 1980] National Bureau of Standards (1980). Data Encryption Standard DES, Data Encryption Standard Model of Operation. Government Printing Office, Washington, D.C. Federal Information Processing Standards FIPS-81.

[Nye, 1992] A. Nye (1992). *Xlib Programming Manual*, volume 1. O'Reilly, troisième édition.

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

[Orfali et Harkey, 1997] R. Orfali et D. Harkey (1997). *Client/Server Programming with Java and CORBA*. John Wiley & Sons, Inc.

[Orfali et al., 1996] R. Orfali, D. Harkey et J. Edwards (1996). *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc.

[Patton, 1998] I. L. Patton (1998). Impact of Using the Ada 95 Distributed Annex vs. CORBA on the Development of a Distributed System. Master's thesis, George Mason University, Fairfax, Virginia, USA.

[Quinot, 1999] T. Quinot (1999). Mapping the Ada 95 Distributed Systems Annex to OMG IDL – Specification and implementation. Master's thesis, École Nationale Supérieure des Télécommunications.

[Reed et al., 1991] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief et B. K. Totty (1991). Scalable performance environments for parallel systems. Paru dans Q. Stout et M. Wolfe, coordinateurs, *The Sixth Distributed Memory Computing Conference*, pages 562–569. IEEE Computer Society Press.

[Renesse et al., 1994] R. V. Renesse, T. M. Hickey et K. P. Birman (1994). Design and Performance of Horus : A Lightweight Group Communications System. Rapport technique TR94-1442, Cornell University, Computer Science Department.

[Reynolds et al., 1983] P. F. Reynolds, J. C. Knight et J. I. A. Urquhart (1983). The implementation and use of Ada on distributed systems with high reliability requirements. *Final report on NASA grant no NAG-1-260*.

[Riley, 1996] J. D. Riley (1996). A Comparison of Two Approaches to Distributed Application Development in Ada : The Distributed System Annex and CORBA. Paru dans *Proceedings of the Tri-Ada Conference*, pages 73–82, new York. ACM Press.

[Rivest, 1994] R. L. Rivest (1994). The RC5 Encryption Algorithm.

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

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

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

[Rosenblum, 1987] D. S. Rosenblum (1987). An Efficient Communication Kernel for Distributed Ada Runtime Tasking Supervisors. *Ada Letters*, VII(2) : 102–117.

[Sacha, 1989] K. M. Sacha (1989). Ada : adding reliability and efficiency to task communication in programming distributed control systems. *ACM SIGADA Ada Letters*, 9(6) : 80–89.

[Satyanarayanan et al., 1987] M. Satyanarayanan, J. J. Kistler et E. H. Siegel (1987). Coda : A Resilient Distributed File System. Paru dans *Proceedings of the IEEE Workshop on Workstation Operating Systems*.

[Schonberg et Banner, 1994] E. Schonberg et B. Banner (1994). The GNAT Project : A GNU-Ada 9X Compiler. Paru dans *Proceedings of Tri-Ada'94*, Baltimore, Maryland, USA.

[Schroeder, 1993] M. D. Schroeder (1993). A State-of-the-Art Distributed System : Computing with BOB. Paru dans S. Mullender, coordinateur, *Distributed Systems*. Addison-Wesley, seconde édition.

[Shatz et al., 1996] S. Shatz, S. Tu, T. Muruta et S. Duri (1996). An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12) : 1307–1322.

[Tanenbaum, 1992] A. S. Tanenbaum (1992). *Modern Operating Systems*. Prentice-Hall International.

[Tomayko et Smith, 1990] J. E. Tomayko et B. Smith (1990). Experiences Porting the Distributed Ada Real-Time Kernel. Rapport technique CMU/SEI-90-TR-17 ADA226693, Software Engineering Institute (Carnegie Mellon University).

[Van Rossum, 1998] G. Van Rossum (1998). Java and Python : a perfect couple. *developer.com journal*.

[Volz, 1990] R. A. Volz (1990). Virtual Nodes and Units of Distribution for Distributed Ada. *ACM SIGADA Ada Letters*, 10(4) : 85–96.

[Volz et al., 1989] R. A. Volz, T. N. Mudge, G. D. Buzzard et P. Krishnan (1989). Translation and Execution of Distributed Ada Programs : Is It Still Ada ? *IEEE Transactions on Software Engineering*, 15(3) : 281–292.

[Wolf et Strohmeier, 1999] T. Wolf et A. Strohmeier (1999). Fault Tolerance by Transparent Replication for Distributed Ada 95. Paru dans *Proceedings of AdaEurope'99*, Santander, Espagne. Springer-Verlag.