Annexe A : Bibliographie commentée

Langages de programmation objets concurrents

ABCL/1

ABCL (*Actor Based Concurrent Language*) [Yonesawa & al. 86, 87, 90] est un langage de programmation orienté objet. Chaque objet peut être dans l'un des trois états suivants : endormi (*dormant*), actif (*active*), ou en attente (*waiting*). Un objet est initialement endormi et devient actif lorsqu'il reçoit un message. Chaque objet a son comportement décrit à l'aide d'un script. Quand un objet actif termine son exécution, il repasse en mode endormi. Un objet actif doit parfois attendre un message spécifique, pour cela, il passe dans le mode attente. Il redevient actif lorsqu'il reçoit un message.

Les transmissions de messages sont de trois types : asynchrone (*past)*, synchrone (*now)*, anticipée (*future)*. Le receveur du message est implicite pour les messages de type *now* et *future* et explicite pour les messages de type *past*. De plus, deux modes de messages existent : standard et express. L'arrivée d'un message express suspend l'exécution d'un message standard. Un acteur ABCL/1 dispose donc de deux boîtes aux lettres. Enfin, les messages peuvent être transmis par passage de continuation explicite via l'opérateur @.

La commande *parallel* permet l'envoi simultané d'un certain nombre de messages. L'exécution parallèle se termine lorsque tous les composants du message ont été envoyés et exécutés. C'est par ce moyen que l'on simule une communication synchrone avec des messages asynchrones.

Ce langage initialement implanté en Common LISP, et maintenant en C, repose sur une machine virtuelle parallèle appelée ABCL/M [Yonesawa 90].

Références bibliographiques

[Yonezawa & al. 86] A. Yonezawa, E. Shibayama, E. Takada & Y. Honda, «Object-Oriented Concurrent Programming in ABCL/1», ACM International Conference on Object-Oriented Programming Systems, Languages and Applications, In Proc. of the OOPSLA'86, pp 258-268.

[Yonezawa & al. 87] A. Yonezawa & M. Tokoro Ed., «Object-Oriented Concurrent Programming», MIT Press, Cambridge, 1987.

[Yonesawa 90] A. Yonezawa, «ABCL: An Object-Oriented Concurrent System.», MIT Press, 1990.

Ada 83

Les entités parallèles sont décrites dans ADA 83 [Ada 83] à l'aide de la notion de tâche. Une tâche ressemble à une procédure, excepté le fait qu'elle n'admet pas de paramètres formels et qu'elle dispose de son propre flot de contrôle. L'interface d'accès à cette tâche est décrite dans la partie spécification d'un paquetage, alors que son comportement est exprimé dans le corps du dit paquetage. Une tâche peut être définie soit comme un objet individuel, soit comme un type de tâche qui peut alors être instancié à volonté. Une tâche commence son exécution au moment de sa création.

Les synchronisations et les communications utilisent les mécanismes de rendez-vous (clause *accept*). Pour plus d'informations sur Ada 83, se référer à [Barnes 84 et Spec 90], quand à ses implémentations réparties se reporter à [Bekele & al. 94] et [Bishop 89].

Références bibliographiques

[Ada 83] U.S. Department of Defense, Ada joint Program Office, «Reference Manual for the Ada Programming Language», ANSI/MIL-STD 1815A, 1983 (ISO 8652-1987).

[Barnes 84] J. Barnes, «Programmer en ADA», InterEditions Ed., 1984, 495 pages.

[Bekele & al. 94] D. Bekele & J.M. Rigaud, «Ada et les Systèmes Répartis», Technique et Science Informatiques, Numéro spécial Ada, Vol 13, No 5, pp. 607-637, 1994.

[Bishop 89] J. Bishop, «Distributed ADA: Developments and experiences», The Ada Companion Series, Cambridge University Press, 1989, 321 pages.

[Spec 90] «Software Engineering with abstractions : An Integrated Approach to software development using Ada», Addison Wesley, 1990.

Ada 95

ADA 95, la définition révisée du langage de programmation Ada, intègre dorénavant la programmation par objets [Barbey & al. 94]. On peut donc créer des objets et utiliser l'héritage de classes et le polymorphisme. Ces nouvelles fonctionnalités employées en phase avec des méthodes de génie logiciel améliorent la qualité des applications, tout en améliorant le temps de développement [Rosen 95].

De nombreuses autres extensions ont été spécifiées concernant notamment la programmation répartie. Ainsi, dans Ada 95 [Ada 95 et Richard-Foy 94], un programme est un ensemble de partitions qui peuvent s'exécuter en parallèle les unes des autres. L'espace d'adressage de chaque partition est séparé. L'exécution répartie d'un programme consiste à exécuter un ensemble de partitions, éventuellement sur des machines distinctes.

Ada 95 décrit un système distribué comme l'interconnexion d'un ou plusieurs noeuds de traitement (ressource comprenant à la fois des fonctions de calcul et de stockage) et d'éventuellement des noeuds de stockage (ressource ne comprenant que des fonctions de stockage, avec de la mémoire adressable par un ou plusieurs noeuds de traitement). Les partitions peuvent être actives (noeud de traitement) ou passives (noeud de stockage). Une partition est identifiée par le regroupement des unités de bibliothèques durant l'édition de liens.

Le RPC est le paradigme de communication entre les partitions actives. Les appels de procédure distantes peuvent-être synchrones ou asynchrones (la sémantique associée est au plus une fois).

Références bibliographiques

[Ada 95] U.S. Department of Defense, Ada joint Program Office, «Reference Manual for the Ada Programming Language», ANSI/ISO/IEC 8652-1995.

[Barbey & al. 94] S. Barbey, M. Kempe & A. Strohmeier, «La programmation par objets avec Ada 9X», Technique et Science Informatiques, Numéro spécial Ada, Vol 13, No 5, pp. 639-669, 1994.

[Richard-Foy 94] M. Richard-Foy, «Ada Connect : un composant logiciel pour la distribution d'applications Ada», Génie Logiciel et systèmes experts, No 34, Mars 1994, pp. 36-41.

[Rosen 95] J.P. Rosen, «Méthodes de génie logiciel avec Ada 95», InterEditions, Collection IIA, 1995, 412 pages.

Amber

Amber [Chase & al. 89] est un système de programmation orienté objet qui permet à une application d'utiliser de manière transparente les ressources disponibles sur un réseau de machines parallèles à mémoire partagée homogène. Amber est le descendant direct de Presto [Bershad & al. 88a] et s'appuie sur son modèle de gestion des processus légers et des synchronisations.

Les objets sont à grain fin et passifs et sont définis dans la classe *Object*. Ils contiennent une partie données et des opérations publiques qui sont invoquées localement ou à distance. Les entités actives sont des processus légers qui possèdent une pile d'exécution et un contexte. Les processus légers s'exécutent sur un espace d'objets partagé sur tout le réseau. Le système maximise les performances en exploitant la concurrence dans des applications de durée de vie courte (il n'existe donc pas d'objets persistants).

La concurrence au sein d'un objet est réalisée en plaçant l'objet et tous les processus légers qui le manipulent sur le même noeud au début de l'exécution. Si l'objet migre, les processus légers qui l'utilisent migrent avec lui. Les processus légers invoquant un objet distant migrent dynamiquement sur le noeud où il réside. Il n'existe pas de mécanisme déclaratif pour contrôler la mobilité des objets ou pour spécifier quels objets sont en relation.

Amber est construit à partir de C++, augmenté de classes de gestion des processus légers et de migration des objets. Le système est composé d'un préprocesseur C++ et d'un noyau d'exécution intégré au programme de l'utilisateur. Le temps partagé et les priorités sont gérés par l'ordonnanceur d'Amber. Il est même possible de créer son propre ordonnanceur [Bershad & al. 88b]. La transparence au niveau réseau n'est pas offerte. Amber est opérationnel sur Topaz, le système d'exploitation de la machine parallèle Firefly de DEC [Thacker & al. 88] composé de microprocesseurs Vax.

Références bibliographiques

[Chase & al. 89] J. Chase, F. Amador, E. Lazowska, H. Levy & R. Littlefield, «The Amber system: Parallel programming on a network of multiprocessors», Proceedings of the 12th ACM Symposium on Operating System Principles«, New York, 1989, in ACM Operating System review, Vol. 23 (5), pp. 147-158.

[Bershad & al. 88a] B.N. Bershad & al., «PRESTO: A system for object oriented parallel programming», Software Practice and Experience, vol. 18 (8), Aug. 1988, pp. 713.

[Bershad & al. 88b] B.N. Bershad & al., «An open environment for building parallel programming language», Proceedings of the ACM SIGPLAN, July 1988.

[Thacker & al. 88] C.P. Thacker, L.C. Stewart & E.H. Satterthwaite, «Firefly: A multi-processor workstation», IEEE Transactions on Computer, Vol. 37 (8), August 1988, pp. 909-920.

Box

Le langage BOX [Geib & al. 93] a été créé en partant d'un double constat :

1) l'approche objet est une méthode de structuration des données en objets, mais initialement séquentielle elle ne s'intéresse pas à la structuration d'un problème en terme d'activités,
2) l'approche par processus communiquants est une méthode de structuration d'un problème en activités concurrentes et en leurs modes de coopération, mais ne s'intéresse que très peu à la structuration des données.

Les deux approches étant complémentaires, l'auteur a eu l'idée de les associer pour créer le langage BOX. Deux modèles ont été créés : le Modèle des Objets (chaque objet étant créé dynamiquement à partir d'une classe) et le Modèle des Processus Communiquants (les processus sont des entités actives autonomes). Les communications entre processus sont asynchrones pour prendre en compte les architectures décentralisées et les objets communiquent entre eux par RPC.

Un objet actif est appelé *fragment* et un objet passif est appelé un *objet*. La création d'un fragment lance l'exécution d'un processus léger. En utilisant des appels de méthodes, les relations objet/objet et fragment/objet sont du type client/serveur, alors que les relations fragment/fragment et fragment/objet sont du type délégation de tâches. La distribution des classes à travers les noeuds du réseau est statique, mais l'instanciation de ces classes est dynamique. Une manière de placer les nouvelles instances de classe créées consiste à trouver les noeuds sur lesquels la classe existe et à choisir celui qui contient le moins d'instances en exécution.

Il est aussi possible de créer des *Objets Actifs Complexes* qui contiennent des objets actifs et passifs et qui permettent de concevoir des composants logiciels orientés objets distribués et fortement réutilisables, grâce à une interface d'accès commune. Un compilateur a été réalisé et les applications s'appuient sur un exécutif réparti, nommé PVC, qui

utilse PVM [Geist & al. 93] avec des processus légers. Ainsi, le programmeur dispose d'outils pour répartir le code des classes utilisées dans des noeuds virtuels, ensuite associés à des machines physiques d'un réseau de stations Sun.

Références bibliographiques

[Geib & al. 93] J.M. Geib, C. Gransard & C. Grenot, «Distributed Objects in Box», TOOLS Europe 93, Workshop on Distributed and Concurrent Objects, Versailles, France, March 8-11, 1993.

[Geist & al. 93] A. Geist, J. Dongarra & R. Manchek, «PVM3 User's Guide and Reference Manual», Oak Ridge Natioanl Laboratory and University of Tenessee, May 1993.

C++ (Extension de Karaorman)

Dans [Karaorman & al. 93], la notion de concurrence est introduite par l'ajout de classes dérivant de la classe *concurrency*. Ces classes enrichissent le langage Eiffel, mais ont été portées sur C++ et modélisent une activité client-serveur. Un objet peut devenir actif seulement s'il hérite de la classe *concurrency*. La création d'un objet actif se fait en deux temps : d'abord un proxy est créé en utilisant la routine séquentielle *create*, puis en invoquant la méthode *split* du proxy. Cette méthode crée un nouveau processus avec son propre processus léger de contrôle, qui lance immédiatement sa routine *scheduler* chargée de gérer l'ordre de traitement des messages qu'il reçoit. Un client désirant accéder au serveur lance un proxy, qui utilise la routine *attach* pour mettre en place une association avec le serveur. Toute communication entre le client et le serveur passe par le proxy, qui envoie un message asynchrone au serveur. Tous les messages sont délivrés en utilisant des IPC. On passe automatiquement d'une classe séquentielle à une classe concurrente, par héritage de la classe *concurrency*.

Référence

[Karaorman & al. 93] M. Karaorman & J. Bruno, «Introducing concurrency to a sequential language», Communications of the ACM, Vol. 36 (9), September 1993, pp 103-116.

C*

C* [TM 88], est une extension du langage C réalisée pour la Connection Machine (qui est une machine SIMD) de la société Thinking Machines. Le langage suppose l'existence d'une machine «front-end» (un VAX par exemple) qui communique avec la Connection Machine. Cette dernière est composée d'un ensemble de processeurs possédant chacun une mémoire locale et qui exécutent tous le même code sur des données qu'ils peuvent s'échanger. La différence majeure entre C et C* est qu'à chaque donnée on doit associer un processeur. La notion de noeud virtuel est alors utilisée pour gérer les cas ou les données sont plus nombreuses que les processeurs (plusieurs noeuds virtuels étant alors regroupés sur un unique processeur).

Trois nouvelles structures ont été rajoutées au C :

1) le type *poly* pour identifier les données traitables en parallèles ;
2) le type *domain* pour grouper les données parallèles (c'est l'équivalent du contructeur *struct* en C). Toutes les données déclarées dans un domaine ont *poly* comme type par défaut ;
3) un mécanisme d'activation et de désactivation d'exécutions parallèles.

Ainsi, deux types de données existent, les données scalaires (de type *mono*) et les données parallèles (de type *poly*). Les données scalaires résident en mémoire centrale du front-end, alors que les données parallèles sont stockées dans la mémoire locale d'un processeur de la connection Machine.

Référence bibliographique

[TM 88] Thinking Machines Co., «C* Reference Manual», Version 4.3, May 1988.

CEiffel

CEiffel [Lohr 93] (*Concurrent Eiffel*) est basé sur le langage Eiffel [Meyer 92] dans sa version 3. Un programme CEiffel est vu par le compilateur comme un programme séquentiel. En effet, toutes les annotations relatives à la concurrence sont exprimées en terme de commentaires [Lohr 92]. Ainsi, un programme Ceiffel a deux significations possibles : une séquentielle et une concurrente.

L'invocation d'une opération, est appelée *Request*. Une fois la demande acceptée par l'objet, une activité est créée. Une activité est suspendue tant que les activités qu'elle a créées ne se sont pas terminées.

La notion de classe partagée est introduite et indique qu'un objet d'une classe peut être partagé par des activités parallèles. Une classe est atomique, si c'est une classe séquentielle ou si elle respecte les propriétés de l'atomicité (est-il possible de sérialiser des activités ?). La stricte atomicité peut être relâchée si le programmeur déclare certaines opérations compatibles (on parle alors d'opérations concurrentes). Des classes semi-concurrentes ne sont ni atomiques, ni concurrentes, mais possèdent des opérations compatibles. Les annotations de compatibilité sont effectives uniquement lors des invocations d'opérations d'un objet, et nullement en cas d'appel de routine local. Un système de verrous garantit l'exclusion mutuelle d'opérations non compatibles.

Une classe avec délai (*Delayed class*) permet l'évaluation de pré et de post-conditions en admettant un délai de grâce en cas de non conformité. Par contre, le non respect de la garde lèvera une exception dans le cas d'une classe séquentielle.

CEiffel offre trois types de routines concurrentes :

1) les routines autonomes, qui sont invoquées implicitement et ne sont en général pas exportées ;
2) les routines asynchrones dont l'invocation entraîne une synchronisation retardée (*lazy synchronisation*). Le contrôle est rendu à l'appelant et un proxy est lancé pour permettre de rendre le résultat ;
3) les routines synchrones.

Ce langage de haut niveau permet d'éviter l'anomalie d'héritage [Matsuoka & al. 93] tant qu'on ne raffine pas une classe. Un pré-compilateur transforme du CEiffel en Eiffel et un run-time permet un exécution concurrente sur plate-forme SUN.

Références bibliographiques

[Lohr 92] K.P. Lohr, «Concurrency annotations improve reusability», Proceedings of TOOLS USA'92, Prentice Hall, 1992.

[Lohr 93] K.P. Lohr, «Concurrency annotations for reusable software», Communications of the ACM, Vol. 36 (9), September 1993, pp. 81-89.

[Matsuoka & al. 93] S. Matsuoka & Y. Yonezawa, «Analysis of Inheritance Anomaly in Object Oriented Concurrent Programming Languages», Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, A. Yonezawa Ed., The MIT Press, 1993.

[Meyer 92] B. Meyer, «Eiffel: The Language», Prentice Hall, Englewoods Cliffs, 1992.

Distributed Smalltalk

Distributed Smalltalk [Bennet 87 & 90] est une implémentation de SmallTalk qui permet à des objets sur des machines physiques distinctes d’envoyer et de recevoir des messa-

ges. Il fournit une invocation d'objets transparente et fabrique automatiquement des références de retour lors des passages d'arguments.

L'implémentation de Distributed Smalltalk n'unifie pas les espaces d'objets utilisateur en mémoire dans un seul espace d'adressage. Chaque utilisateur possède donc son propre espace d'adressage, ce qui implique que les classes et les instances d'objets requis doivent être co-résidents dans chaque espace d'adressage. Certaines classes sont d'ailleurs répliquées sur chaque machine, ce qui pose des problèmes de mise à jour.

Le partage d'objets entre utilisateurs, l'expression de relations entre objets ou la gestion de la mobilité des objets sont des services qui ne sont pas implémentés dans Distributed Smalltalk.

Références bibliographiques

[Bennett 87] J.K. Bennet, «The design and implementation of Distributed Smalltalk», ACM International Conference on Object Oriented Programming Systems, Languages ans Applications, OOPSLA'87 Proceedings, 1987, pp. 318-330.

[Bennett 90] J. Bennet, «Experiences with Distributed Smalltalk», Software Practice and Experience, Vol. 20 (2), Feb 1990, pp. 157-180.

Docase

Le projet DOCASE [Müllhaüser & al. 93] (*Distributed Object in CASE*) a pour but de fournir tous les outils nécessaires pour permettre le développement d'applications orientées objets réparties. Pour cela deux langages ont été créés DODL (*DOcase Design Language*) et TsL.

DODL est un langage hybride dans le sens où il intègre à la fois la modélisation et l'implémentation de l'application. Pour cela trois catégories de types ont été définies et offrent des sémantiques prédéfinies pour des catégories d'applications données :

- **les types configurés** sont utilisés pour créer le squelette de l'application. Ces objets ont une durée de vie longue (l'auteur dit qu'ils sont stables) et en enlever ou en rajouter un modifie considérablement l'application. Ils sont reliés entre eux par des connexions unidirectionnelles. Les sous-catégories d'objets de ce type sont types actifs (qui modélise des acteurs ou des agents), agents d'environnement (qui permettent de gérer l'interopérabilité avec des logiciels déjà existants) et des sous-systèmes (ce qui permet la hiérarchie).
- **les types générés** sont créés et manipulés sans modifier la configuration de l'application. Ils sont créés en masse et ont une durée de vie limitée (un message est l'exemple type de cette catégorie d'objets).
- **les types de relation** sont utilisés pour spécifier une sémantique complexe de la relation qui unit deux objets. Ainsi, ces relations sont établies entre objets de types configurés et spécifient les détails des échanges entre objets. Les sous-catégories d'objets de ce type sont la colocation, la coopération et l'interaction. La colocation permet en cas de migration de conserver sur un même noeud des objets en relation. La coopération assigne un contrôle de flot global à des objets qui fournissent chacun une partie des fonctionnalités nécessaires à l'obtention d'une chaîne de traitements.

Des objets ou des traitements sont déclarables comme non encore définis, dans le but de créer un prototype qui grandira et s'étoffera avec le temps. Les noeuds logiques offrent l'interfaçage avec la configuration physique et reflètent la granularité de l'exécution.

TsL est un langage dit de superimposition, qui sépare dans le code l'aspect opérationnel des algorithmes utilisés. Des implants sont alors réalisés dans le code de l'application et permettent des règles de sélection qui identifient l'emplacement au sein d'un algorithme.

La concurrence est basée sur les processus légers et est divisée en trois niveaux. Les processus légers sont traités comme des composants de noeud logique, de type actif et utilisant la coopération. Chaque objet est localisé sur exactement un noeud logique. Chaque noeud logique est exécuté comme un processus du système d'exploitation.

Référence bibliographique

[Müllhaüser & al. 93] M. Müllhaüser, W. Gerteis & L. Heuser, «DOCASE: a methodic approach to distributed programming», Communications of the ACM, Vol. 36 (9), Sept. 1993, pp. 127-138.

Dowl

DOWL [Achauer 93] est une extension du langage parallèle Trellis [Moss & al. 87]. C'est un langage orienté objet réparti, qui permet des invocations transparentes d'objets distribués. La migration d'objets entre les noeuds du réseau est offerte.

Des objets peuvent être attachés entre eux pour permettre lors de la migration de grouper les transferts d'objets interdépendants. Trois types d'attachement existent : statique (pour toute la durée de vie de l'objet), automatique (durant l'exécution d'un bloc d'instruction) ou à la demande (on modifie alors dynamiquement les attributs des objets). Les instances créées à partir de type peuvent avoir des propriétés qui varient dans le temps, on les appelle alors des objets mutables, sinon ils sont nommés objets constants.

L'espace de travail de DOWL est distribué à travers les noeuds du réseau. Un processus DOWL est créé par noeud et les objets doivent s'exécuter dans l'espace d'adressage de ce processus. Les objets constants, donc non modifiables, sont répliqués alors que les objets mutables ne changent pas d'emplacement et sont accessibles par des proxy [Decouchant 86, Shapiro 86] les référençant. DOWL fonctionne sur VAX and DECstation sous Ultrix.

Références bibliographiques

[Achauer 93] B. Achauer, «The DOWL distributed object-oriented language», Communications of the ACM, Vol. 36 (9), September 1993, pp 48-55.

[Decouchant 86] D. Decouchant, «Design of a distributed object manager for the Smalltalk-80 system», Proceedings of OOPSLA'86, ACM / SIGPLAN, 1986, pp 444-452.

[Moss & al. 87] E. Moss & W. Kohler, «Concurrency features for the trellis / owl language», Proceedings of ECOOP'87, Paris, France, june 1987.

[Shapiro 86] M. Shapiro, «Structure and encapsulation in distributed systems: the Proxy principles», Proceedings of 6th Int. Conf. on Distributed Computing Systems, Cambridge, May 1986, pp 198-204.

Eiffel //

Eiffel// [Caromel 93] est une extension du langage Eiffel (dans sa version 2), et se situe dans la catégorie des langages de classes. Il offre une programmation parallèle asynchrone et impérative pour des machines de type MIMD. Tous les processus du modèle sont des objets, mais tous les objets ne sont pas des processus. Ainsi, deux types d'entités existent :

- les objets restent des structures de données classiques : passifs, ils exécutent leurs routines uniquement lorsque celles-ci sont appelées.
- un processus représente un objet qui a une activité propre, donc actif. Les communications entre processus se font grâce aux routines exportées par chaque processus et qui peuvent être appelées. Un processus est l'instance d'un classe qui hérite directement (ou indirectement) de la classe PROCESS.

Une des fonctionnalités très intéressante de ce langage est l'affectation polymorphe entre objets et processus. Ainsi, une entité qui n'est pas un processus va pouvoir dynamiquement référencer un processus. Statiquement, c'est un appel de routine et dynamiquement

c'est une communication entre processus. D'autre part, les communications se font par interruption et rendez-vous, ce qui permet à la requête d'être transmise. Puis, la réponse est transmise de manière asynchrone. Un mécanisme d'attente par nécessité a été créé et permet à tout processus de se mettre en attente automatiquement lorsqu'il tente d'utiliser une valeur non disponible (appelé un objet attendu).

L'implémentation réalisée s'est faite à partir du code source de la version commerciale d'Eiffel sur une plate-forme de machines SUN. Dans le futur, une version sur machine parallèle est envisagée et s'ajoutera à la possibilité de distribution offerte aujourd'hui grâce à un système de noms virtuels de machines.

Référence bibliographique

[Caromel 93] D. Caromel, «Towards a Method of Object Oriented Concurrent Programming», Communications of the ACM, 36 (9), September 1993, pp 90-102.

Emerald

Emerald [Black & al. 88] possède un langage orienté objet très fortement typé. Il a été conçu principalement pour permettre la migration d'objets et pour réduire les coûts de communication. Pour obtenir des performance optimales, le compilateur et le noyau système sont très fortement couplés.

Dans Emerald, chaque objet est composé de quatre parties : un identificateur unique sur le réseau (appelé OID), une représentation de données, un ensemble d'opérations et optionnellement un processus. Les objets Emerald qui contiennent un processus sont dits actifs. les objets actifs invoquent d'autres objets et un flot de contrôle d'un objet peut s'étendre sur d'autres objets situés sur d'autres machines. Plusieurs flots de contrôle peuvent s'exécuter dans un même objet. La synchronisation entre objets se fait par moniteur.

Ce langage ne permet pas l'héritage. La mobilité des objets peut être restreinte en décidant de les fixer à un noeud du réseau. Des objets peuvent être attachés entre eux pour permettre lors de la migration de grouper les transferts d'objets interdépendants. Enfin, un système original de passage de paramètres a été mis au point. On peut ainsi passer un argument à un objet en faisant :

- un appel à la référence de l'objet (*call by reference*) ;
- un appel par transfert (*call by move*) et dans ce cas on migre définitivement l'objet sur le site ou se trouve l'objet invoqué ;
- un appel par visite (*call by visit*) et dans ce cas on migre temporairement l'objet sur le site ou se trouve l'objet invoqué.

Les objets dans Emerald sont mobiles pour éviter des références multiples sur un même objet distant et il est possible de définir des catégories d'objets réutilisables (*template*). Enfin, un ramasse-miettes local et distribué se charge de récupérer les objets qui ne sont plus référencés.

Références bibliographiques

[Black & al. 88] A. Black, J.E. Levy & H. Hutchinson, «Fine Grained Mobility in the Emerald System», ACM Transactions on Computer Systems, Vol. 6 (1), feb 1988, pp. 109-133.

[Jul & al. 88] E. Jul, H. Levy, N. Hutchinson & A. Black, «Fine Grained Mobility in the Emerald System», ACM Transactions on Computer System, 6(1), 1988, pp. 109-133.

Epee

Epee [Jezequel 93] (*Eiffel Parallel Execution Environment*) est un langage qui étend Eiffel et qui se focalise sur le parallélisme de données, en arguant du fait que la programmation orientée objet est en général plus basée sur les données que sur les fonctions et le contrôle. Ainsi, le but est de partitionner les données et d'associer à chaque partition un

processeur. Chaque processeur exécute le même programme, ainsi la vision que l'utilisateur a de l'exécution de son programme est une vision séquentielle.

Epee cache tout le parallélisme à l'utilisateur (transparence totale), en encapsulant dans une classe spécifique toutes les opérations de communication d'un machine SIMD. Une programmation à deux niveaux est donc mise en place. Ainsi, l'utilisateur se sert des classes disponibles et le programmeur crée les classes nécessaires. Des classes sur les machines suivantes sont disponibles : réseaux de Sun, Hypercube intel iPSC/2 et iPSC/860 et Paragon XP/S. Aucune modification n'a été faite par rapport à Eiffel, tout passe par la définition des classes. Les migrations d'objets sont ainsi encapsulées dans des classes, pour s'affranchir d'un système de nommage global des objets. Enfin, tous les objets ont une composante locale sur chaque site. Ainsi le ramasse-miettes d'Eiffel est largement suffisant. Les gains en performances (*speed up*) sont quasi linéaires par rapport au nombre de processeurs, pour des calculs suffisamment longs. Notons que l'amélioration des compilateurs Eiffel, permettra d'augmenter les performances.

Référence bibliographique

[Jezequel 93] J.M. Jezequel, «EPEE : An Eiffel Environment to program distributed memory parallel computers», Journal of Object Oriented Programming, Vol. 6 (2), May 1993, pp 48-54.

Occam

Le langage Occam [Hoare 88 et Gaudiot & al. 92] est en développement chez Inmos depuis 82 et s'inspire très fortement de CSP définit par [Hoare 78]. La première version d'Occam offrait les constructions et mécanismes principaux qui formaient la base du langage. Une application est alors composée de processus pouvant s'exécuter en parallèle et communiquants par message selon le principe du rendez-vous. Dans Occam2, le typage fort, ainsi que la création de fonctions sans effet de bord ont, entre autres, été rajoutés. Occam a été créé pour permettre l'exécution de programmes sur des réseaux de processeurs et peut être implémenté sur tout type de machine. Il se trouve qu'Occam est particulièrement performant sur les architectures à base de Transputer.

Deux notations existent en Occam pour définir le type d'un processus : les processus parallèles sont déclarés par le mot clef PAR et les processus séquentiels par le mot clef SEQ. Les interactions entre processus se font via des canaux en utilisant les primitives d'envoi (!) et de réception (?) de message sur un canal.

Le programmeur peut gérer la configuration de l'application en indiquant quel processus s'exécute sur quel processeur par le constructeur du langage PLACED PAR. Des priorités sont assignables aux processus qui s'exécutent sur un même processeur.

Références bibliographiques

[Gaudiot & al. 92] J.L. Gaudiot & D.K. Yoon, «A comparative study of Parallel Programming Languages: The Salishan Problem», J.T. Feo Ed., Elsevier Science, 1992, pp 217-262.

[Hoare 78] C. Hoare, «Communicating Sequential processes», Communications of the ACM, Vol. 21 (8), August 1978.

[Hoare 88] C. Hoare, «Occam 2 reference Manual», Prentice Hall, 1988.

PlasmaII

Le langage acteur PLASMAII [Arcangeli 94] est un langage concurrent non typé. La structure de données est la séquence (analogue à la liste Lisp). Le filtrage est utilisé comme mécanisme de sélection des messages, mais il sert aussi à créer l'environnement. Les acteurs sont créés à l'aide de la primitive SERIALIZE, à partir d'un comportement initial et traitent les messages reçus en série (c'est pourquoi on les appelle acteurs sérialisés). Le changement de comportement n'est pas obligatoire et certains acteurs sont même

insensibles à l'histoire de leur calcul. Ainsi, PLASMAII permet la création d'acteurs de comportement qui réagissent comme des fonctions.

Au niveau des communications, des protocoles ont été rajoutés par rapport au modèle initial de Agha tels que : la transmission bloquante, la transmission avec attente différée (clause *futur*), la transmission avec priorités et interruptions et la diffusion à un groupe d'acteurs.

PLASMAII sépare la description du problème de son implémentation. Le programme créé s'exécute alors sur une machine virtuelle Smart [Francès 89], constituée d'un ensemble de processeurs virtuels. Le placement des acteurs sur les processeurs se fait automatiquement en prenant en compte le nombre d'acteurs par processeur. Des primitives permettent de forcer le placement d'un acteur sur un processeur et de migrer un acteur d'un processeur virtuel vers un autre. On peut aussi moduler le grain de parallélisme en regroupant un ensemble d'actions à exécuter.

PLASMAII est un langage interprété qui est portable sur de nombreuses architectures (réseau de Sun, réseaux de transputers, Butterfly TC2000) grâce à Smart. Smart gère des MVE (*Machines Virtuelles Elémentaires*) communiquant par passage de messages. Les transmissions entre MVE se font par sockets (Unix), par des liens transputer ou par mémoire partagée (Butterfly TC2000).

Référence bibliographique

[Arcangeli & al. 94] J.P. Arcangeli, A. Marcoux, C. Maurel & P. Sallé, «La programmation concurrente par acteurs : PLASMAII», Calculateurs Parallèles, «les langages à objets», No 22, Juin 1994, pp 99-121.

Pool

POOL [America 87] est un langage de programmation pour la construction d’applications larges et complexes qui s’exécutent sur une machine multi-processeurs sans mémoire partagée. Dans POOL, un système est une collection d’objets composés :

- de variables pour le stockage des données ;
- d’un ensemble de méthodes pour accéder et modifier ses données encapsulées ;
- d’un corps qui est un processus local qui s’exécute en parallèle avec les corps des autres objets du système.

POOL introduit la notion d'objet actif, représenté par un processus. Ainsi, tous les objets possèdent une activité propre décrite dans une méthode spéciale qui devient active dés la création de l'objet. Une fois créé, un objet (i.e. processus) peut rester actif après avoir rendu la main à son créateur. La communication entre objets se fait par envoi de messages synchrones. POOL dispose d’un ramasseur de miettes à la volée.

Références bibliographiques

[America 87] P. America, «Pool-T : a Parallel Object-Oriented Programming», A. Yonezawa Ed., Object-Oriented Concurrent Programming, The MIT Press, 1987, pp 199-220.

[America & al. 90] P. America & F. Van Der Linden, «A Parallel Object-oriented Language with Inheritance and Subtyping», In Proc. of OOPSLA / ECOOP'90, ACM SIGPLAN, Vol. 25 (10), Oct. 1990.

Langages de description d'architectures logicielles

Aesop

Aesop [Garlan & al. 94] est un générateur d'environnement. Il demande en entrée une description d'un style architectural qu'Aesop combine avec quelques infra-structures de bases pour produire un environnement complet (appelé une fable). La définition du style consiste à décrire l'ensemble des types d'objets (les éléments du vocabulaire), ainsi qu'un ensemble d'outils de visualisation et d'analyse de l'architecture à intégrer dans l'environnement. Les éléments du vocabulaire sont au nombre de six : les composants, les connecteurs, les configurations, les ports, les rôles, les représentations et les liaisons. Les interactions entre **composants** sont modélisées par des **connecteurs**. Les **configurations** décrivent la topologie formée par l'interconnexion des composants et des connecteurs. Un composant dispose d'interfaces avec l'environnement appelées des **ports**. Les interfaces des connecteurs sont appelées des **rôles** et identifient les participants d'une interaction. Les composants et les connecteurs peuvent-être hiérarchiques, dans ce cas on décrit leur contenu par **une représentation**. L'usage de représentation implique la mise en place de correspondances entre les éléments de la configuration interne et externe. Ainsi, **une liaison** (connecteur) est la correspondance entre un port (un rôle) interne avec un port (rôle) externe.

Aesop vérifie que la composition des éléments de style satisfait les contraintes de topologie du style (recherche de cycles et de connecteurs pendants). Un environnement Aesop est structuré comme un ensemble ouverts d'outils qui accèdent à une base de données orienté-objet commune. Cette base de données stocke les styles et fournit une interface de haut niveau aux outils. Chaque outil s'exécute sous la forme d'une processus indépendant et accède à la base de données par une interface basée sur des RPC (appelé la machine abstraite de la fable). L'environnement dispose d'un mécanisme de diffusion de messages pour notifier à la base de données et annoncer à des outils des évènements récents [Reiss 90].

Références bibliographiques

[Garlan & al. 94] D. Garlan, R. Allen & J. Ockerbloom, «Exploiting Style in Architectural Design Environment», Proc. of the ACM SIGSOFT'94, New Orleans, LA, December 1994.

[Reiss 90] S.P. Reiss, «Connecting Tools Using Message Passing in the Field Program Development Environment», IEEE Software, July 1990.

Aster

Aster est un environnement de configuration de logiciels hétérogènes. Aster fournit un environnement de programmation pour la construction d'applications distribuées qui supporte la sélection et/ou construction automatique de systèmes d'exécution distribuée, spécialisés suivant les exigences de l'application. Aster permet aussi la réutilisation du logiciel applicatif et système, notamment par la prise en compte des standards d'architectures logicielles comme Corba, Tina ou le WWW.

L'environnement Aster est constitué d'un langage d'interconnexion de module (*MIL*) et d'un environnement d'exécution (*runtime*) basé sur l'ORB de Corba. L'idée centrale d'Aster est de considérer l'ORB de l'OMG comme une instance d'un bus logiciel abstrait et de s'en servir pour personnaliser des systèmes distribués. On cherche alors à répondre aux besoins des applications au dessus de n'importe quel type d'architecture sans modifier l'implémentation de l'application.

Dans le langage de configuration d'Aster, l'élément de base est le composant. Celui-ci est décrit au travers :

- de ses interfaces, par la déclaration des signatures et des opérations clientes et de service. Cette déclaration est quasiment immédiate avec l'IDL de Corba, il suffit juste de définir les opérations des clients.
- de son implémentation, en associant un fichier de code à chacune de ses interfaces.
- de sa hiérarchie. Il est possible de regrouper des composants de manière hiérarchiques au sein d'un même composant, mais via **des liaisons appropriées** (notamment de type multi-parties).

La description de l'implémentation et de la hiérarchie se fait par le rajout de clauses à l'IDL de l'OMG. Ces clauses sont fortement inspirées de Polylith [Purtilo 94], si on fait exception de la hiérarchie.

La notion de connecteur n'existe pas dans Aster, elle est remplacée par la notion de bus logiciel spécialisé. Dans tous les cas, un bus logiciel est alors considéré comme un composant procurant une interface avec laquelle l'application s'interconnecte verticalement. L'expression de la spécialisation du système est alors réalisée par la définition des propriétés d'exécution, via un treillis de propriétés. Ces propriétés sont rassemblées sur un arbre de propriétés (*property tree*) spécifiant l'ensemble des propriétés reconnues par Aster. Ainsi, la déclaration des contraintes applicatives et des comportements est réalisée avec des listes de propriétés dans des clauses du langage (*require*/*provide*). La sélection et la construction des systèmes spécialisés est réalisable de deux manières :

- de manière explicite par un programmeur ;
- de manière automatique, par interconnexion d'un système de base avec des composants systèmes implantant les fonctions complémentaires requises et disponibles dans Aster. Ce mécanisme repose sur l'appariement des spécifications formelles de propriétés de l'exécution.

La phase de génération de l'application est réalisée par modification automatique des composants logiciels (code source) pour un interfaçage avec la plate-forme cible. Ainsi, chacun des appels de procedure est traduit en un appel à des actions du système spécialisé. Le prototype actuel d'Aster fonctionne avec Orbix (l'ORB de Iona Technology) et a été écrit en C++.

Références bibliographiques

[Issarny & al. 96a]	Issarny V. & Bidan C., «Aster: A Framework for Sound Customisation of Distributed Runtime Systems», Proceedings of the 16th International Conference on Distributed Computing Systems, May 1996.
[Issarny & al. 96b]	Issarny V. & Bidan C., «Aster: A Corba-based Software Interconnection System Supporting Distributed System Customization», International Conference on Configurable Distributed Systems, Annapolis (MD, USA), May 1996.
[Purtilo 94]	J.M. Purtilo, «The Polylith Software Bus», ACM Transactions on Programming Languages and Systems, Vol. 16 (1), pp. 151-174, 1994.
Site Internet	http://www.irisa.fr/solidor/work/aster

Darwin

Darwin [Fossa & al. 96, Magee & al. 95] est un langage d'interconnexion de modules (MIL). Tout programme distribué est construit par une description de la configuration hiérarchique d'un ensemble d'instances de composants et de leur connexion. Pratiquement, Darwin structure un programme distribué ou parallèle en terme de groupes de processus qui communiquent par envoi de messages. Darwin est aussi utilisé pour décrire l'évolution dynamique de la structure d'une application au cours de l'évolution de son exécution.

Dans Darwin, chaque composant définit son interface strictement en terme de services fournis et de services requis. Darwin utilise une notation déclarative pour la description des composants hiérarchiques, dans laquelle doivent être présents à la fois les noms de composants utilisés (mot clef *use*), les instances des composants contenus (mot clef *inst*), mais aussi des liaisons (mot clef *bind*) entre ces composants. La liaison entre composants consiste à associer les services requis par l'un à ceux fournis par d'autres. Darwin permet aussi la réplication des instances de composants et des interfaces, des configurations conditionnelles avec évaluation de gardes et des définitions récursives de composants avec des instanciations retardées (*lazy instantiation*).

Références bibliographiques

[Fossa & al. 96] H. Fossa & M. Sloman, «Implementing Interactive Configuration Management for Distributed Systems», Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis, Maryland, May 1996, IEEE Press.

[Magee & al. 95] J. Magee, N. Dulay, S. Eisenbach & J. Kramer, «Specifying Distributed Software Architecture», Fifth European Software Engineering Conference (ESEC'95), Barcelona, September 1995.

Durra

Durra [Barbacci & al. 91, 92 & 93] est un langage et un noyau d'exécution permettant de développer des applications distribuées. Durra sépare la structure de l'application de son comportement. Un utilisateur décrit alors une application comme un ensemble de composants, un ensemble de configurations alternatives explicitant l'interconnexion des composants lors de l'exécution et un ensemble de transitions de configuration.

Les deux notions essentielles du langage sont les tâches (*tasks*) et les canaux (*channel*). Les tâches sont des composants actifs qui initient toutes les opérations de passage de messages. Une tâche est soit primitive (*primitive*), soit composée d'un ensemble de tâches (*compound*). Les canaux sont des composants passifs qui attendent et réagissent à des requêtes envoyées par les tâches. On peut donc utiliser plusieurs protocoles entre tâches ou utiliser différents canaux de communication pour qu'une même tâche puisse réaliser des communications synchrones et asynchrones.

Les tâches et les canaux sont utilisés par le programmeur comme une boite noire, il n'a pas accès à son contenu. Le programmeur spécifie la distribution physique des composants de l'application en leur assignant des grappes séparées. Ces grappes spécifient le groupement physique des composants dans une image exécutable de l'application qui doit correspondre aux connections du modèle. Une grappe est obtenue par la liaison (*binding*) des implémentations des tâches et des canaux, de l'environnement d'exécution (*run-time*) de Durra et des tables de configuration. Le compilateur Durra génère automatiquement des programmes Ada distribués [Doubleday & al. 91]. L'assignation physique de tâches et de canaux à des processeurs s'effectue lors de l'exécution. La reconfiguration dynamique est prise en charge par le gestionnaire de grappes. Les communications locales à une grappe utilisent des rendez-vous Ada, alors que les communications entre grappes utilisent les IPC. Le gestionnaire de grappes est responsable du lancement et de la terminaison des processus, du passage des messages entre composants, de la surveillance de l'obtention de conditions de reconfiguration et de la gestion des reconfigurations. Enfin une grappe particulière maître est chargée de l'initiation des reconfigurations. Le statut de maître est donné dynamiquement à un coût faible, car chaque grappe dispose de l'ensemble des tables définissant l'application.

Références bibliographiques

[Barbacci & al. 91] M. Barbacci & R.W. Lichotta, «Durra: An Integrated Approach to Software Specification, Modeling and Rapid Prototyping», 2nd Int. Workshop on Rapid System Prototyping, Research Triangle Park, North Carolina, USA, pp. 67-81, June 1991.

[Barbacci & al. 92] M.R. Barbacci, D.L. Doubleday, C.B. Weinstock, M.J. Gardner & R. Lichota, «Building fault tolerant distributed applications with DURRA», Proceedings of the International Workshop in Configurable Distributed Systems, IEEE press, pp. 128-139, England, March 1992.

[Barbacci & al. 93] M.R. Barbacci, C.B. Weinstock, D.L. Doubleday, M.J. Gardner & R. Lichota, «DURRA: a Structure Description Language for Developing Distributed Applications», Software Engineering Journal, Vol. 8 (2), pp. 83-94, 1993.

[Doubleday & al. 91] D.L. Doubleday, M.R. Barbacci, C.B. Weinstock, M.J. Gardner & R. Lichota, «Building Distributed Ada Applications from Specifications and Functional Components», Proceedings of the Tri-Ada 91 Conference, 1991.

Olan

Olan est un environnement de développement d'applications coopératives implémenté au dessus d'une plate-forme distribuée basée sur des objets offrant la distribution, la partage et le regroupement (*clustering*) d'objets.

OCL (*Olan Configuration Language*) est un langage de configuration qui s'inspire de Darwin et qui est dédié à l'intégration et la distribution de composants logiciels hétérogènes dans l'environnement Olan. Les deux objets de bases d'OCL sont les composants et les connecteurs.

Les composants

Le composant est l'unité d'encapsulation et de distribution. Un composant est une instance d'une classe et une classe de composants est définie par une interface, une réalisation et une distribution.

L'interface décrit les services fournis par le composant aux autres composants. Une interface comporte :

- **des services** : ce sont des procédures ou des fonctions fournies ou requises par le composant.
- **des notifications** [Boyer 95] qui sont des évènements émis par le composant vers l'extérieur.
- **des réactions** qui sont les réponses à des notifications émises par des composants.
- **des attributs** qui sont des variables typées publiques et qui définissent un ensemble de propriétés associées au composant.

La différence entre un service et une notification est qu'une demande de service à une interface doit toujours être traitée, alors qu'une notification peut être ou ne pas être traitée par un composant.

La réalisation fait correspondre l'interface et la partie logicielle qui l'implémente. Deux types de réalisation existent :

1) **la réalisation d'un composant primitif** comprend le code des entités (classes écrites en C++ ou Python, modules, bibliothèques) qu'il encapsulent.
2) **la réalisation d'un composant complexe** définit la représentation structurelle du composant en terme des composants qu'il contient et des interactions entre ces composants contenus. Un composant complexe ne contient pas à proprement parler de code, mais les éléments qui permettent de mettre en oeuvre son interface.

L'administration d'une application OLAN est réalisée par une interface spéciale et par la spécification d'attributs de gestion. Ces attributs sont visibles dans des outils d'administration ou pas et sont modifiables ou non. Les attributs servant au placement des composants d'une application sont actuellement de deux types :

- le noeud (*Node*) décrivant le site d'exécution désiré ;

- l'utilisateur (*User*) décrivant le possesseur du programme.

Chaque attribut est composé de prédicats qui définissent des critères de choix de placement (tel que le système d'exploitation, ou la charge de la machine ou le nombre d'utilisateurs connectés). Ces critères sont pris en compte lors de la configuration de l'application.

Enfin, le concept de **collection** de composants est nécessaire à la description de la dynamicité de la structure interne d'un composant. Une collection est un ensemble nommé d'instances d'un même composant. Ainsi, pendant l'exécution du composant, la structure interne d'un composant est associée à la création et à la destruction d'instances et à l'ajout et au retrait des interactions entre composants. L'instanciation d'un composant est réalisée :

- de manière immédiate : dès que le composant englobant est créé ;
- de manière retardée : dès qu'un composant essayera de faire appel à lui (*lazy instanciation*).

Les connecteurs

Les connecteurs sont les entités qui décrivent les interactions en établissant un lien entre les composants et en spécifiant le protocole de communication devant être mis en oeuvre [Allen & al. 94].

La description d'un connecteur inclue :

- la mise en conformité des interfaces connectées. Il faut alors vérifier le respect des signatures et éventuellement mettre en place un protocole d'adaptation ;
- la définition du protocole utilisé pour mettre en oeuvre la communication (synchrone/asynchrone, par diffusion d'évènements ou par RPC) ;
- la spécification du comportement exprimé sous la forme d'un ensemble de contraintes décrivant la qualité de service attendue.

C'est pourquoi dans le langage, la description d'un connecteur comprend le type du connecteur, un ensemble de sources, un ensemble de destinations et la spécification du comportement.

La plate-forme OLAN supporte différents types de connecteurs dont ceux pour l'interconnexion de composants et ceux de liaison d'une interface avec une réalisation. Les connecteurs sont construits manuellement sans l'aide d'un langage de spécification et doivent-être inclus dans la plate-forme Olan.

Enfin, dans la terminologie Olan, **une interaction** est la communication effective qui a lieu entre un ensemble de composants. En fait, les interactions sont des instances d'un connecteur dans un contexte clairement défini.Elles sont décrites par la liaison entre les services fournis et requis ou les notifications et les réactions des sous composants et par le choix d'un protocole de communication.

L'environnement d'exécution

Le choix primordial qui a été fait pour l'implémentation de l'environnement d'exécution est de créer un objet pour chaque entité décrite avec OCL. Ainsi, les composants, les connecteurs, les collections sont des objets distribués. C'est pourquoi, la compilation d'un programme OCL génère des structures pour les composants et les connecteurs et du code pour la machine virtuelle de configuration. Ce code est responsable de la transformation du programme OCL en structures d'exécution et de l'installation de ces structures sur un système distribué. Les structures d'exécution créées sont les composants, les collections, les connecteurs et les interconnexions de composants avec des connecteurs. Le déploiement de l'application est réalisé par une phase de correspondance de la hiérarchie des composants sur les machines hôtes.

L'architecture d'exécution de l'environnement OLAN est composée de quatre machines virtuelles gérant l'exécution des composants et des connecteurs, la configuration de l'application et son administration (cf. Figure 88).

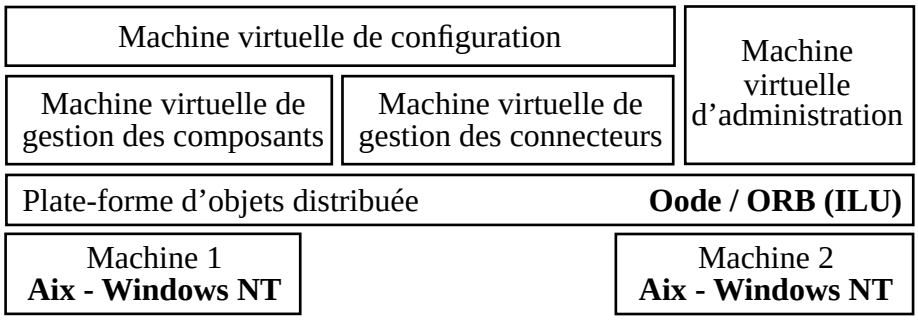

Figure 88 : L'environnement d'exécution OLAN

Actuellement, l'implémentation de l'environnement OLAN comprend le compilateur OCL (sans la génération du code de la machine de configuration) et l'environnement d'exécution OLAN au dessus d'AIX et de Solaris.

Les prototypes actuels des machines virtuelles ont été construits :

- avec l'environnement d'exécution Oode [Bull 94] de Bull. Oode dispose d'un langage orienté objet (une extension de C++), qui permet la programmation transparente de la distribution et la manipulation d'objets distribués avec des mécanismes d'accès concurrents. La machine virtuelle a été écrite en IDL pour la spécification des interfaces d'objets et en OC++ (une extension de C++ gérant des objets persistants). Les limites de cette plate-forme sont dûes au typage statique d'objet en C++, qui complique la création dynamique d'un composant à partir d'une classe de composants.
- soit en s'appuyant sur ILU [Xerox 95], un ORB conforme à Corba. Grâce à ILU, l'interopérabilité des objets écrits dans des langages différents et s'exécutant sur des plate-formes différentes est envisageable. La machine virtuelle a quant à elle été écrite en Python, un langage interprété semi-compilé et portable sur des plate-formes Windows, Macintosh et Unix. Python offre les fonctionnalités suivantes : le typage dynamique, la mobilité du code et l'interprétation dynamique du code.

Références bibliographiques

[Allen & al. 94] R. Allen & D. Garlan, «Formal Connector», Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University., March 1994.

[Bellissard & al. 96] Bellissard L., BenAtallah S., Boyer F. & Riveill M., «Distributed Application Configuration», Proceedings of the 16th International Conference on Distributed Computing Systems, ICDCS'96, pp. 579-595, IEEE Computer Society, Hong-Kong, May 1996. Site Internet : <http://sirac.imag.fr/>.

[Bellissard & al. 95] Bellissard L., BenAtallah S., Kerbat A. & Riveill M., «Component-based Programming and Application Management with Olan», Object-based Parralel and Distributed Computation France-Japan Workshop, OBPDC'95 Selected Papers, Briot J.P., Geib J.M. & Yonesawa A. eds., LNCS 1107, Tokyo, Japan, June 1995.

[Boyer 95] F. Boyer, «Coordinating Software Development Tools with Indra», Proceedings of the 7th Conference on Software Engineering Environments (SEE'95), IEEE Computer Society Press, Nederland, pp. 1-13, April 1995.

[Bull 94] Bull Open Software System, «Oode : une Plate-forme Objet pour les Applications Coopératives», Actes des journées d'études sur la répartition et le parallélisme dans les systèmes d'information, Afcet Eds., Paris, France, pp. 253-266, 22-23 Novembre 1994.

[Xerox 95] Xerox Corp, «ILU 1.8 Reference Manual», Xerox Parc, Palo Alto CA01, October 1995.

QNAP2

Le logiciel de simulation QNAP2 (*Queuing Network Analysis Package*) [Simulog 88 & Potier 86] est un outil de description et d'analyse de réseaux de files d'attente. Il provient du travail conjoint de chercheurs de l'INRIA et de Bull. Ce produit, programmé en Fortran 77 pour en assurer la portabilité, est actuellement diffusé par la société Simulog.

Un programme QNAP2 se décompose en trois parties :

1) la configuration du réseau ;
2) le traitement effectué sur chaque station ;
3) le contrôle de la résolution du réseau.

Un réseau de files d'attente est composé de stations déclarées par le mot clef «/STATION/» à travers lesquelles circulent des clients, selon des règles de routage données. Dans QNAP2, la création de classes de clients simplifie la programmation et permet une meilleure compréhension des résultats de la simulation.

Une station est caractérisée par :

- son nom,
- son type (positionné par défaut à SERVER). Si une station possède n serveurs, alors elle doit être paramétrée par l'instruction TYPE=MULTIPLE(n) ou TYPE=INFINITE si le nombre de serveurs est infini. Ce dernier cas est utilisé pour modéliser le cas ou chaque client arrivant dans la station est immédiatement servi. Une station peut aussi être de type RESSOURCE simple (SINGLE), multiple (MULTIPLE(n)) ou infinie (INFINITE). Les synchronisations se font à l'aide de stations de type SEMAPHORE simple ou multiple. Enfin, une station de type SOURCE est utilisée pour modéliser la production de clients.
- la discipline de service de la file d'attente (mot clef SCHED), positionnée par défaut à FIFO (premier arrivé, premier servi). D'autres disciplines existent telles que LIFO (dernier arrivé, premier servi), PRIOR (les clients sont classés en fonction de leur priorité), QUANTUM (un serveur est alloué aux clients pour une période de temps fixée) et PREMPT (un client de plus grande priorité réquisitionne le serveur au client en service; le service du client réquisitionné est repris quand le serveur est réalloué).
- l'initialisation du nombre de clients par classe (mot clef INIT). Par défaut, si les classes ne sont pas précisées, le simulateur crée le même nombre de clients pour chaque classe.
- le paramètre de service (mot clef SERVICE) pour chaque classe de clients. Ce paramètre est toujours suivi soit par une procédure exprimant une durée définie pour une distribution de probabilité, soit par une instruction algorithmique, soit par une combinaison des deux.
- le paramètre de transit (mot clef TRANSIT) décrit les règles de routage des clients à la fin de leur service.
- la priorité de chaque client entrant dans la station en fonction de sa classe (mot clef PRIOR),
- le service d'allocation de quantum pour chaque classe de clients dans la station (mot clef QUANTUM).

Les solveurs présents dans QNAP2 fournissent des résultats de simulation exacts (CONVOL [Baskett & al. 75], MVA [Reiser & al. 80], etc.) ou approchés (HEURSNC [Neuse & al. 81], DIFFU [Gelenbe & al. 77], etc.).

Références bibliographiques

[Baskett & al. 75] F. Baskett, K.M. Chandy, R.R. Muntz & F.G. Palacios, «Open, Closed and Mixed Networks of Queues with Different Classes of Customers», Journal of ACM, Vol 22 (2), April 1975, pp 248-260.

[Gelenbe & al. 77] E. Gelenbe & G. Pujolle, «A Diffusion Model for Multiple Class Queuing Networks», Rapport de Recherche INRIA, No 242, août 1977.

[Neuse & al. 81] D. Neuse & K. Chandy, «SCAT: A Heuristic Algorithm for Queuing networks on Computing Systems, ACM Sigmetrics, Vol 10 (1), 1981, pp 59-79.

[Potier 86] D. Potier, «The Modeling Package QNAP2 and Applications to Computer Networks Simulation», Computer Networks and Simulation, Tome III, Schoemaker S. Ed., North Holland, 1986.

[Reiser & al. 80] M. Reiser & S.S. Lavenberg, «Mean Value Analysis of Closed Multichain Queuing Networks», Journal of the ACM, Vol 22 (2), April 1980, pp 313-322.

[Simulog 88] Simulog (Ed.), «Manuel de référence de QNAP2», 1988.

Rapide

Rapide est un langage orienté-objet, concurrent et basé sur la gestion d'évènements. Rapide [Luckham & al. 95] fournit des constructions pour créer des prototypes exécutables d'architecture et adopte un modèle d'exécution dans lequel la concurrence, la synchronisation et le flot de données sont représentés. Rapide a été créé pour permettre la validation, l'exécution et la mesure de performance d'une architecture avant son implémentation. C'est pourquoi, l'architecture est définie avant l'écriture des modules, par composition des modules et établissement de contraintes de communications entre les modules et le système.

Dans Rapide, une architecture consiste en un ensemble de spécification (les interfaces) d'un module, un ensemble de règles d'interconnexion (qui définissent les communications directes entre les interfaces) et un ensemble de contraintes formelles qui décrivent les gabarits de communications légales ou illégales. Une interface peut contenir une définition abstraite du comportement du module. Les comportements sont définis par un ensemble de règles réactives. Les connections définissent des communications de données synchrones ou asynchrones entre interfaces. Enfin, les contraintes formelles spécifient des restrictions sur des aspects divers des connections et des interfaces.

Le modèle d'exécution, appelé *poset*, est basé sur l'envoi et la réception d'évènements. Ainsi, les interfaces sont en attente de la réception de certains évènements et réagissent à la réception de ces évènements par la génération d'autres évènements. Les connections sont gérées par des règles réactives. Les contraintes restreignent l'envoi d'évènements à la fois aux interfaces et sur un ensemble de connexions. Dés qu'un module est assigné à une interface, il devient un composant exécutable et le comportement de l'interface est restreint par le comportement du module. Chaque module et chaque connexion peut ensuite s'exécuter concurremment et tous les évènements des interfaces doivent satisfaire à tout moment les contraintes d'architecture.

Rapide est composé de cinq parties :

1) le langage de types (*type language*) pour décrire les interfaces du composant ;
2) le langage d'architecture (*architecture language*) pour décrire le flot d'évènements entre composants ;
3) le langage de spécification (*specification language*) pour écrire des contraintes abstraites sur le comportement des composants ;
4) le langage exécutable (*executable language*) pour écrire des composants exécutables ;
5) le langage de gabarit (*pattern language*) pour décrive des gabarits d'évènements.

Rapide supporte la programmation de modules en ADA, en C++ et en Verilog. L'exécution d'architecture (ie. la simulation) est réalisée sur tout le système et permet de tester les interfaces, les connexions ainsi que certains comportements de l'architecture. Une architecture peut être assignée à une interface comme composant d'une autre architecture.

Références bibliographiques

[Luckham & al. 95] D.C. Luckham, L.M. Augustin, J.J. Kenney, J. Veera, D. Bryan & W. Mann, «Specification and Analysis of system architecture Using Rapide», IEEE Transactions on Software Engineering, Vol. 21 (4), pp. 336-355, April 1995.

[Thomas & al. 91] D.E. Thomas & P.R. Moorby, «The Verilog Hardware Description Language», New York, Kluwer-Academic, 1991.

Regis

Regis [Magee & al. 94] est un environnement de programmation supportant le développement et l'exécution de programmes distribués. Le développement d'applications est réalisé en séparant la structure de l'application de son implémentation.

Regis est le successeur :

- du système Conic [Magee & al. 89] qui séparait la structure du comportement des composants d'une application, mais était restreint à un langage de programmation unique (un Pascal augmenté de d'un ensemble de primitives de communication). L'ensemble des instances des composants et leur interconnexion étaient figés à l'exécution.
- du système Rex (*Reconfigurable and EXtensive Parallel and Distributed Systems*) [Kramer & al. 92] qui succéda à Conic. REX supportait que les composants logiciels d'une application soient programmés dans différents langages, mais en communiquant qu'avec un jeu réduit de primitives. REX disposait d'un gestionnaire de reconfiguration dynamique puissant, mais l'inclusion de primitives de reconfiguration rendaient difficiles la lisibilité et la traçabilité des programmes.

Regis apporte par rapport à ses successeurs des capacités de recomposition dynamique lors de l'exécution des liaisons entre les composants logiciels d'une application et une gestion des communications découplées du programme. La recomposition des éléments constituant l'application modifie donc dynamiquement la configuration des composants logiciels, mais pas leur architecture.

Pour Regis, un programme est constitué d'une collection limitée de types de composants et de multiples instances de ces types. L'environnement d'exécution et les éléments de communication de Regis sont programmés en C++. Regis combine ces éléments avec une description de la configuration du programme, écrite dans le langage de configuration Darwin [Magee & al. 95 et Fossa & al. 96].

L'interaction des composants est mise en place dans Regis à l'aide de classes d'interaction définies en C++ (en utilisant des modèles ou *template*). Un service fournit par un composant est réalisé par une instance d'une de ces classes. Cette instance est référencée à distance par le composant qui désire accéder au service.

L'environnement d'exécution de Regis est implémenté par un processus démon Unix présent sur chaque machine candidate à l'exécution distribuée. Des informations concernant les ressources et l'environnement requis, ainsi que le programme à exécuter sont copiés sur la machine cible. Un outil graphique de gestion interactive de l'architecture et de la configuration d'une application a été implémenté [Fossa & al. 96, Ng & al. 95] et a donné lieu à un prototype au dessus d'ANSAware. Il est possible de créer une instance d'un composant et de le lier dynamiquement à d'autres. Des extensions futures sont envisagées avec Corba.

Références bibliographiques

[Fossa & al. 96] H. Fossa & M. Sloman, «Implementing Interactive Configuration Management for Distributed Systems», Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis, Maryland, May 1996, IEEE Press.

[Kramer & al. 92] J. Kramer, J. Magee, M. Sloman & N. Dulay, «Configuring Object Based Distributed Programs in REX», IEE Software Engineering Journal, Vol. 7 (2), March 1992, pp. 139-149.

[Magee & al. 89] J. Magee, N. Dulay & J. Kramer, «Constructing Distributed Systems in Conic», IEEE Transactions on Software Engineering, SE-15 (6), 1989, pp. 663-675.

[Magee & al. 94] J. Magee, J. Kramer & M. Sloman, «Regis: A Constructive Development Environment for Distributed Programs», IEEE/IOP/BCS, Distributed Systems Engineering Journal, Vol. 1 (5), pp. 304-312, September 1994.

[Magee & al. 95] J. Magee, N. Dulay, S. Eisenbach & J. Kramer, «Specifying Distributed Software Architecture», Fifth European Software Engineering Conference (ESEC'95), Barcelona, September 1995.

[Ng & al. 95] K. Ng, J. Kramer, J. Magee & N. Dulay, «The Software Architect's Assistant - A Visual Environment For Distributed Programming», Proc. of the Hawaii International Conference on System Sciences (HICSS-28), January 1995.

Site Internet ftp://dse.doc.ic.ac.uk/regis.tar.gz

Unicon

Unicon (UNIversal CONnector) [Shaw & al. 95 et 96] est un langage de description d'architecture constitué de deux types d'objets : les composants et les connecteurs. Leur structure est symétrique en tout point excepté pour les connecteurs qui ne peuvent être hiérarchique. Il sont décrits par les attributs suivant :

- leur nom ;
- une spécification appelée une interface pour un composant et un protocole pour un connecteur ;
- un type ;
- un ensemble d'assertions globales (une liste des propriétés) ;
- une collection d'unités d'association ;
- une implémentation.

Un connecteur est constitué d'un protocole qui spécifie la classe d'interaction que le connecteur fournit et son implémentation.

Un composant est représenté par une interface qui spécifie les capacités d'exportation et ses implémentations. Une implémentation peut être primitive ou composite. Un composant primitif est implémenté directement dans un langage de programmation donné. Ces composants sont intégrés dans Unicon en utilisant des techniques d'intégration (*wrapping*). Une implémentation primitive fournit alors une ou plusieurs manières (appelées des variantes) d'intégrer du code qui est soit un code source, soit un objet, soit un exécutable. Une implémentation composite instancie un ensemble de composants et les configure avec les connecteurs.

L'interface d'un composant doit-être consistante avec son implémentation et est constituée :

- du type du composant ;
- d'assertions et de contraintes qui s'appliquent sur le composant ;
- d'acteurs (*players*) constitués d'un nom et d'attributs optionnels.

Le type d'un composant restreint le nombre, les types et les spécifications des acteurs du composant. Les acteurs forment des unités de sémantique visibles à travers lesquelles les

composants interagissent, offrent et demandent des services ou réagissent à des évènements extérieurs. Les interactions entre composants sont néanmoins prises en charge par les rôles des connecteurs. Un acteur est une liste d'attributs et une liste de leurs valeurs associées.

Un système est un composant, généralement composite, capable d'effectuer des opérations indépendamment du contexte d'une machine et de son système. La spécification externe du système est l'interface du composant. Pour permettre la construction de systèmes complexes, un système peut être utilisé comme un composant. L'implémentation d'un composant composite doit fournir trois types d'information :

- les instanciations des composants et des connecteurs qui forment le composant composite ;
- les associations entre les acteurs et les rôles (ie. comment les instanciations de connecteurs sont liées aux instanciations de composants).
- la description de l'association entre les acteurs de l'interface et ceux de l'implémentation. En effet, l'interface du composant définit les acteurs que le composant doit fournir, ce sont les acteurs extérieurs. Dans l'implémentation les acteurs sont fournis quand les composants sont instanciés, ce sont les acteurs internes. Il faut donc définir les acteurs externes en terme d'acteurs internes.

Unicon dispose d'un éditeur de système graphique et textuel. Le modèle créé est ensuite compilé pour produire un programme exécutable.

Références bibliographiques

[Shaw & al. 95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young & G. Zelesnik, «Abstraction for Software Architecture and Tools to Support Them», IEEE Transactions on Software Engineering, Vol. 21 (4), pp. 314-335, April 1995.

[Shaw & al. 96] M. Shaw & D. Garlan, «Software Architecture: Perspective on an Emerging Discipline», Prentice Hall Eds., 1996.

Normes et spécifications

ATM

ATM (*Asynchronous Transfer Mode*) a été développé par le CNET [CNET 91] dans les années 80, pour répondre à plusieurs problèmes :

- intégrer plusieurs types de trafic de nature très différente : vidéo, audio, données;
- offrir un débit important et variable dans le temps;
- allouer dynamiquement dans le temps des ressources du réseau à la demande de l'application.

Le Tableau 45 présente quelques exemples des débits moyens et des débits crêtes requis pour les services actuels disponible sur un réseau.

Tableau 45: Exemples de services et des débits associés

Services	Débit Moyen	Débit Crête
Télévision	25 Mbit / s	34 Mbit / s
Télévision HD	100 Mbit / s	135 Mbit / s
HI FI Audio	2 Mbit / s	2 Mbit / s
Téléphone	64 Kbit / s	64 Kbit / s
Vidéoconférence	2 Mbit / s	10 Mbit / s
Télécopie Couleur	2 Mbit / s	2 Mbit / s
Transfert de Fichier	20 Kbit / s	10 Mbit / s
Interconnexion de LAN	64 Kbit / s	34 Mbit / s

L'apport des réseaux hauts-débits basés sur ATM au sein des architectures de réseaux sont multiples. D'abord, ils vont permettre de dépasser les limitations actuelles des protocoles de communications tels que TCP/IP. En effet, ces protocoles ne sont pas parallélisables intrinsèquement [Nahum & al. 94]. Ensuite, ils offrent une bande passante plus large, sur des distances plus grandes. Enfin, ils permettent de gérer de manière très fine la qualité de service requise pour un service donné.Présentation d'ATM

La technique de transmission utilisée est le multiplexage temporel asynchrone. L'information est structurée en unités de données de longueur fixe appelées cellules. Le choix de la taille d'une cellule (53 octets) a été dictée par des considérations de performances.

Le service ATM fonctionne en mode connecté. On dénombre deux types de connexions :

- les connexions de conduit virtuel VP (VPC - *Virtual Path Connection*) qui relient un ou plusieurs liens de VP.
- les connexions de voie virtuelle VC (VCC - *Virtual Channel Connection*) qui relient un ou plusieurs liens de VC. La VCC est unidirectionnelle.

Dans une liaison physique réelle, il peut y avoir plusieurs VPC. Sur une VPC, plusieurs VCC peuvent être multiplexées entre le même couple émetteur - récepteur. La Figure 89 décrit le multiplexage des connexions VCC sur une connexion VPC, elles mêmes multiplexées sur une ligne physique. L'en-tête d'une cellule ATM contient obligatoirement les deux champs contenant le VPC et le VCC.

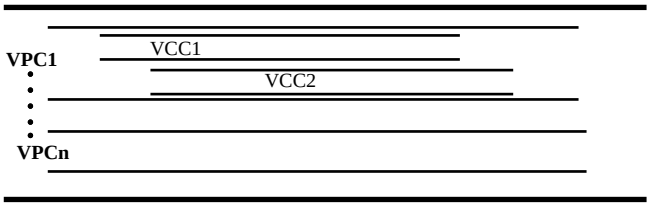

Figure 89 : Multiplexage VCC/VPC/Liaison

Un réseau ATM est un réseau maillé, dont les noeuds (ou commutateurs) sont reliés en point à point selon une topologie quelconque. Les commutateurs actuels ont des temps de latence minimaux de quelques microsecondes [Pujolle & al. 92]. La Figure 90 donne un exemple de commutation correspondant à une VCC multiplexée sur une VPC à travers des commutateurs de VP et de VP/VC. Quand la commutation se base uniquement sur le numéro du VP, le VC n'est pas utilisé et le routage est plus rapide. Dans l'esprit des concepteurs d'ATM, une connexion de VP est semi-permanente et est destinée à supporter un grand nombre de VCC. Ces dernières sont donc de nature temporaire en fonction des besoins de l'application qui les utilise. Malgré son caractère semi-permanent, il est possible de faire varier la bande passante allouée à une VPC, toujours sur demande de l'application. La gestion des connexions (ouverture, fermeture) est faite sur le réseau par signalisation.

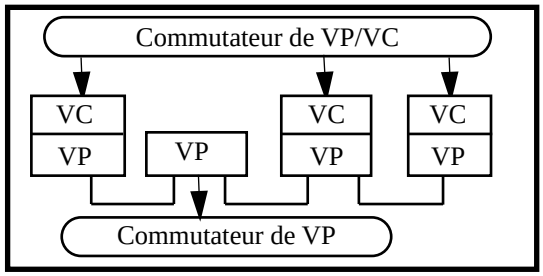

Figure 90 : Commutation dans un réseau ATM

L'architecture globale du protocole ATM est découpée en trois couches :

1) la couche Physique qui est la couche la plus basse de la pile de protocoles. Elle se décompose en deux sous couches. La première appelée Media Physique est responsable de l'émission et de la réception des éléments binaires sur le support. La seconde appelée Convergence de Transmission assure la construction et l'adaptation du flux des cellules au système de transmission utilisé.
2) la couche ATM est chargée du contrôle de flux, du routage, du multiplexage des cellules et du respect de la qualité de service demandé.
3) La couche AAL (*ATM Adaptation Layer*) assure l'adaptation des contraintes des applications aux caractéristiques du réseau ATM. Entièrement mise en oeuvre dans le terminal, elle opère de bout en bout et masque les insuffisances du service de transport des cellules aux applications. Elle se décompose en deux sous couches : la couche de Convergence (ou CS) et la couche plus basse de segmentation et de réassemblage (ou SSAR). Quatre classes de services ont été définies pour la couche AAL (cf. Tableau 46).

Le lecteur désirant plus d'informations pourra se reporter aux ouvrages de [Pujolle & al. 94] et de [De Prycker 94].

La qualité de service

ATM a aussi été conçu pour prendre en compte les exigences des utilisateurs en terme de qualité de service. Certains flux d'informations ne supportent pas des retards d'achemi-

nement, voire des coupures de liaison (comme la vidéo-conférence par exemple). Une phase de négociation doit donc être effectuée, lors de l'établissement de la communication, sans qu'on puisse être assuré de son succès. Par contre, une fois la liaison établie, l'utilisateur est assuré d'avoir une qualité de service constante, ce qui lui laisse la possibilité d'évaluer avec plus de précision les performances de son application.

Tableau 46: Classes de service de la couche AAL

	Classe A	**Classe B**	**Classe C**	**Classe D**
Relation temporelle entre la source et la destination	Requise	Requise	Non Requise	Non Requise
Débit	Constant	Variable	Variable	Constant
Mode de connexion	Orienté Connexion	Orienté Connexion	Orienté Connexion	Sans Connexion

Références bibliographiques

[CNET 91] L'écho des recherches, «Spécial ATM», No 144, Vol I & II, Centre National d'Etudes des Télécommunications, 1991.

[De Prycker 94] M. De Prycker, «ATM: Mode de transfert Asynchrone», Masson-Prentice Hall Eds., 327 pages.

[Nahum & al. 94] E. M. Nahum, D. J. Yates, J.F. Kurose and D. Towsley, «Performance Issues in parallelized network protocols», Proceedings of the first USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey (CA), November 1994.

[Pujolle & al. 92] G. Pujolle, «Commutateurs ATM : classification et architectures», Technique et Science Informatique, Hermes Ed., Vol 11 (1), pp 11-29.

[Pujolle & al. 94] G. Pujolle, «Les réseaux large bande et l'ATM», Eyrolles Ed., 1994.

COM

COM [COM 95, Orfali & al. 96] (*Component Object Model*) est une spécification décrivant :

- l'interaction entre les applications entre elles et le système à travers un ensemble de fonctions appelées interfaces. Une interface est un type (et pas une classe) immutable et accessible par un identificateur unique. Chaque objet COM implémente une interface spécifique (*IUnknown*) qui est utilisable par les clients pour découvrir dynamiquement si une interface est supportée par un objet. D'ailleurs, tous les services OLE sont simplement des interfaces COM. L'utilisation d'une interface COM entraîne la création d'un contrat entre le client et le serveur. Ce contrat ne doit pas être rompu lors de l'évolution des objets, c'est pourquoi COM utilise, entre autres, des numéros de version pour gérer ses contrats.
- une gestion des noms intelligents persistants par l'intermédiaire d'objets de liaison (*moniker*). Ces noms sont utilisés pour réaliser des liens depuis un objet source vers un objet cible pouvant être contenu dans un document composite. Il existe ainsi par exemple un moniker pour gérer les URL (*Uniform Ressource Locator*) nécessaires à la localisation d'un document sur Internet.
- un mécanisme d'empaquetage (*marshalling*) et de désempaquetage (*unmarshalling*) des paramètres d'interface à travers les processus, les machines et le réseau. Le passage de pointeurs sur des interfaces est réalisé localement par des LRPC et à distance par des RPC DCE. Ce mécanisme est transparent à la fois pour le client et le serveur.
- un système de stockage structuré d'objets persistants qui permet d'être interopérable avec des systèmes de gestion de fichiers propriétaires. COM utilise des fichiers

composés (*compound file*), qui sont des regroupements hiérarchiques de données et de répertoires représentant un document composite.

- un mécanisme de transferts de données uniforme (*Uniform Data Transfer*). Les transferts de données entre applications se font à travers des objets «tampons», selon un protocole de type «producteur-consommateur».
- une architecture et une infrastructure (*CoGetClassObject*) d'identification transparente et de chargement dynamique d'un objet local ou distant dans le système sous la forme d'un exécutable (*EXE*) ou d'une librairie dynamique (*DLL*).
- une librairie de fonctions pour les développeurs.

COM fournit un modèle de programmation simple, mais fortement couplé au langage C++. COM ne supporte pas l'héritage pour se soustraire au problème de la classe de base fragile et utilise deux mécanismes à la place :

1) la délégation d'interfaces. Un objet «externe» utilise les opérations d'un objet «interne» pour mettre en oeuvre ses propres opérations (cette opération équivaut à une spécialisation sans sous typage).
2) l'agrégation d'interfaces. Un objet «externe» peut publier directement les opérations d'un objet «interne» en plus des siennes. Avec ce mécanisme, chaque appel au code d'une classe parente d'un objet est explicite, la hiérarchie est donc gérée de manière claire et sécurisée par le programmeur [Wayner 94].

Références bibliographiques

[COM 95] Microsoft Corporation, «The Component Object Model Specification», Version 0.9, October 24, 1995, site internet : ftp.microsoft.com / developr / drg / OLE-info / COMSpecification.

[Orfali & al. 96] R. Orfali, D. Harkey & J. Edwards, «The Essential Distributed Objects Survival Guide», Wiley Ed., pp. 429-452, 1996.

[Wayner 94] P. Wayner, «Object on the March», Byte, pp. 139-150, January 1994.

Corba

CORBA 1.1 [OMG 91] est une spécification d'infrastructure de communication au niveau applicatif fournissant des mécanismes au moyen desquels des objets peuvent émettre des requêtes et recevoir des réponses. CORBA permet ainsi à des applications actives sur différentes machines d'un environnement distribué hétérogène de coopérer selon un modèle client-serveur (cf Figure 91). Corba généralise les mécanismes d'appels de procédures à distance (RPC) par appel d'opérations sur des objets.

Le client est un programme ayant accès à l'environnement CORBA et qui a été écrit dans un langage possédant une projection IDL. Le serveur (aussi appelé réalisation) est un code qui implante les services d'un objet (un programme exécutable, un démon Unix, une base de données, etc.) offrant des services par l'intermédiaire de ses interfaces. Lors d'une communication entre un client et un serveur, le noyau de l'ORB transporte les requêtes sur le réseau et assure l'uniformité des accès aux objets. L'adaptateur d'objet (*Object adapter*) reçoit les requêtes adressées aux objets et adapte la sémantique d'invocation des objets CORBA à celle de l'implantation des objets. Un adaptateur d'objet :

- génère et interprète des références d'objets ;
- regroupe les objets dans un seul contexte ;
- authentifie les requêtes ;
- active et désactive les implantations ;
- active et désactive les objets ;
- permet l'appel des méthodes à travers le bout serveur.

Les squelettes d'opérations décodent les requêtes CORBA pour invoquer le code des objets (c'est le complément de l'adaptateur d'objet).

Autour de l'ORB, se greffent deux types d'interfaces clients pour accéder aux objets distants (cf. Figure 91) :

1) l'interface d'invocation statique (*Static Invocation Interface*) est un ensemble d'opérations, fonctions ou procédures applicatives, accessibles dans le langage du client et utilisée pour l'invocation d'un objet distant de manière transparente et via une ou plusieurs souches.
2) l'interface d'invocation dynamique (*Dynamic Invocation Interface*) est un ensemble de fonctions génériques pour la composition de requêtes nécessaires à l'appel des opérations dont le sélecteur et les paramètres n'étaient pas connus lors de la compilation de l'appelant. C'est ce service qui offre la découverte d'objets et de leurs interfaces, qui crée les requêtes, qui réalise l'invocation distante du service et qui reçoit les réponses. Ce service s'appuie sur le dépôt d'interfaces (*Interface Repository*) qui contient les objets qui représentent les interfaces IDL. Il rend ainsi accessible à l'exécution les interfaces pour découvrir les services offerts par les objets du système.

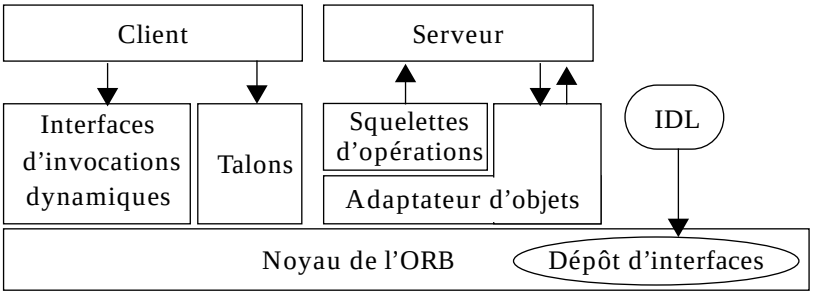

Figure 91 : Les composants de l'architecture fonctionnelle CORBA

Corba 1.2 [OMG 92] est une révision de la norme basée sur des retours d'expérience des principales implémentations de Corba.

L'architecture de Corba 2.0 [OMG 95] peut être vue comme une extension de celle de Corba 1.x qui reste compatible uniquement au niveau des interfaces. L'apport de Corba 2.0 s'est fait dans deux directions : l'interopérabilité entre ORB et une meilleur intégration avec les langages de programmation C++ et Smalltalk.

En ce qui concerne la communication entre ORB, Corba 2.0 définit :

- une architecture pour des communications entre ORB respectant l'architecture CORBA ;
- une API pour ajouter des passerelles entre ORB ;
- un protocole de communication entre ORB, appelée UNO (*Universal Network Object*). Grâce à UNO, un client d'un ORB donné peut invoquer une opération d'un objet disponible sur un ORB différent. Pour cela deux protocoles existent :
  - **GIOP** (*General Inter-ORB Protocol*) qui spécifie une syntaxe de transfert standard et un format de messages pour les communications entre ORB. Il existe une spécialisation du GIOP pour Internet (ie. sur des réseaux utilisant le protocole TCP/IP) appelée **IIOP** (*Internet InterOrb Protocol*).
  - **ESIOP** (*Environment Specific Inter-Orb Protocol*). ESIOP a été créé principalement pour offrir l'interopérabilité avec des systèmes distribués existants non conformes à GIOP et est basé sur CIOP (*Common Interoperability Protocol*) de DCE.

UNO impose une représentation commune des types de données du modèle objet de Corba. Avec cette représentation, appelée CDR (*Common Data Representation*), il est possible de coder tout type descriptible en IDL.

Interopérabilité CORBA / CORBA dans la réalité

Du point de vue du dialogue entre ORB, l'objet client doit obtenir la référence de l'objet qu'il appelle et qui devient l'objet serveur, avant de le solliciter. Cette référence, appelée IOR (*Interoperable Object Reference*) est actuellement délivrée sous la forme d'un fichier texte. L'IOR contient à la fois le gestionnaire de l'objet et l'adresse physique de l'objet serveur actif. Cet IOR est totalement transparent pour le client, qui n'a pas besoin de connaître la manière dont s'effectue la localisation d'un objet. Tout objet serveur est configuré de manière à écrire dans un fichier une représentation de son IOR. Ce point est l'une des principales difficultés pour l'administration des objets CORBA, car le transfert des fichiers IOR entre ORB est difficilement gérable à grande échelle. Des services de nommage inter-ORB doivent être mis au point et intégrés aux outils actuels pour résoudre ce problème.

Interopérabilité CORBA / OLE - COM

Les deux standards de gestions d'objets répartis, à savoir CORBA et OLE/DCOM, malgré une conception et des intérêts radicalement différents convergent l'un vers l'autre. L'objectif de l'OMG est de définir un environnement d'exploitation pour interpréter directement les abstractions propres aux langages orientés objets (classe, héritage, invocation de méthode) et les traduire au niveau de l'exécution. C'est une approche de haut niveau basée sur un modèle d'objet neutre. L'OMA définit donc les abstractions qui devront être comprises par l'environnement d'exécution, en utilisant un langage de description d'interfaces commun (l'IDL). L'approche de Microsoft avec OLE/COM, par contre, se définit d'emblée comme un standard d'interopérabilité binaire, par opposition avec tout standard d'interopérabilité de plus haut niveau. COM est un ensemble d'API donnant accès aux mécanismes à travers lesquels un client peut se connecter à différents fournisseurs du service requis de manière polymorphe. Un objet COM est l'implémentation d'un ensemble d'interfaces, chacune d'entre elles étant réellement un tableau de pointeurs vers des fonctions exécutables issues de la compilation d'un langage source.

Malgré ces différences, l'interopérabilté entre CORBA 2.0 et COM est actuellement en cours de normalisation au sein de l'OMG. Le RFP5 (*Request For Proposal*) décrit un mécanisme d'interopérabilité à deux niveaux : soit entre OLE2 et CORBA, soit entre COM et CORBA. En attendant les résultats de ces travaux, les intégrations possibles entre OLE et CORBA sont les suivantes :

1) les fournisseurs de produits OLE implémentent CORBA, ce qui entraîne que tout objet OLE est vu comme un objet CORBA ;
2) les éditeurs d'outils de développement modifient leurs produits pour qu'ils génèrent des applications OLE ou CORBA ;
3) les fournisseurs de médiateurs développent des passerelles logicielles entre les deux ;
4) les solutions d'intégration sont mises en place au niveau des bibliothèques systèmes.

Interopérabilité SOM-DSOM/CORBA

SOM et DSOM sont conformes à CORBA et Opendoc s'appuyant sur SOM l'est donc aussi.

Références bibliographiques

[OMG 91] Dec, HP, Hyperdesk, NCR, ODI, SunSoft, «The Common Object Request Broker : Architecture and Specification», OMG document No 91.12.1, Revision 1.1, X/Open and OMG Eds, Framingham, USA, Décembre 1991.

[OMG 92] Dec, HP, Hyperdesk, NCR, ODI, SunSoft, «The Common Object Request Broker : Architecture and Specification», OMG document No 93.xx.yy, Revision 1.2, Framingham, USA, Décembre 1992.

[OMG 95] Object Management Group, «CORBA 2.0 / Interoperability, Universal Networked Objects», OMG document No 95.3.10, Framingham, USA, Mars 1995.

Site Internet http://www.omg.org.

ECMA PCTE

Une norme pour l'intégration des différents composants de l'architecture logicielle, appelée ECMA PCTE [PCTE 91 & 93], a été édictée conjointement par l'ECMA (*European Computer Manufacturers Association*) et le NIST (*National Institute of Standards and Technology*). Cette norme propose un modèle de référence pour les structures d'accueil d'environnements de génie logiciel. L'intégration des outils est réalisée en terme de services offerts au sein d'une architecture fonctionnelle d'environnement. On retrouve dans cette norme les cinq axes d'intégration définis par [Wasserman 90] à savoir :

1) **l'intégration au niveau données**, qui est l'aspect le plus achevé de cette norme, est réalisée via une gestion d'objets partagés. Les relations entre objets sont définies dans un schéma (*Schema Definition Set*). Les objets dans un schéma représentent les données utilisées par les outils et les liens représentent les relations entre un outil et des données;
2) **l'intégration au niveau présentation**, pour conserver la même interface graphique utilisateur quel que soit l'outil utilisé;
3) **l'intégration au niveau contrôle**, pour que la notification d'évènements entre outils soit possible. ECMA PCTE définit des services de communication par trois types de messages : **requête** (demande de service), **notification** (réponse à une requête) et **exception** (évènement exceptionnel);
4) **l'intégration au niveau procédé de fabrication**, pour favoriser le développement rapide d'application (*Rapid Application Development*);
5) **l'intégration au niveau de la plate-forme d'exécution**, qui consiste à garantir la transparence d'accès aux outils dans un environnement réseau et système hétérogène.

Références bibliographiques

[PCTE 91] ECMA, «A reference model for frameworks of software engineering environments (version 2).», ECMA Report N° TR/55, NIST Report N° SP 500-201, December 1991.

[PCTE 93] ECMA, «A reference model for frameworks of software engineering environments (version 3).», ECMA Report N° TR/55, NIST Report N° SP 500-xxx, April 1993.

[Wasserman 90] A. Wasserman, «Tool Integration in Software Engineering Environments», Software Engineering Environments, Springer Verlag, LNCS, No 467, pp. 138-150, 1990.

ODP

ODP *(Open Distributed Processing)* [BMODP 96 et CNIS 95] est issu d'une procédure de normalisation internationale concernant **les systèmes répartis ouverts**. Cette norme

est constituée du modèle de référence appelé RM-ODP et d'un document technique décrivant les caractéristiques d'un courtier (*trader*).

Le modèle de référence

Le modèle de référence est découpé en quatre parties :

1) Introduction et Guide : présente les concepts clefs du modèle de référence et les motivations des problèmes à résoudre ;
2) Modèle Descriptif : définit les concepts et un cadre analytique pour une description normalisée des systèmes répartis ;
3) Modèle Prescriptif : spécifie les caractéristiques requises pour juger de la conformité d'un système à la norme. S'il est conforme, alors il est qualifié «d'ouvert». Pour cela, on utilise les outils techniques du Modèle Descriptif ;
4) Sémantique Architecturale : formalise les concepts du Modèle Descriptif en interprétant chacun d'entre eux avec différentes techniques de description formelle.

L'architecture générique

L'idée de base d'ODP est qu'il est inutile de créer de nouveaux des API pour rendre un système distribuée ouvert, mais que la conception de son architecture est primordiale. Or, la conception de l'architecture est l'étape la plus difficile. La norme définit donc un modèle générique d'architecture, qui n'est pas lié à un domaine d'applications ou à un type donné de systèmes.

L'architecture générique est obtenue en :

- normalisant des points de vue. RM-ODP s'est d'ailleurs largement inspiré d'ANSA, car on y retrouve les cinq points de vues Entreprise, Information, Traitement, Ingénierie et Technologie.
- prescrivant des modèles spécifiques à l'aide desquels il est possible de spécifier les principes et les règles de conformité adoptées dans chaque point de vue.
- identifiant des fonctions nécessaires à la réalisation d'applications réparties ouvertes.

Dans RM-ODP, l'unité de gestion est l'objet. Les objets ont des liaisons entre eux et les vues sur les objets sont appelées des interfaces et sont de deux types : opérationnelles (ie. déclenchant une opération sur l'objet) et de flux (pour le transfert d'informations à intervalles de temps régulier). Pour créer un objet conforme à la norme, on utilise des profils d'objets (*template*). Ainsi, dans le point de vue calcul, il existe deux catégories d'objets (les objets de bases et les objets de liaison qui décrivent les communications entre les objets de bases) et trois types d'interfaces (opérations, signaux et stream). Un template d'objet basique contient un ensemble de template d'interfaces, une spécification de comportement et un contrat d'environnement. Le contrat d'environnement spécifie la qualité de service requise (délai, volume, sûreté) et l'usage de l'objet (localisation, sélection des transparences).

Si on se place dans le point de vue Ingénierie, cinq éléments clefs sont définis :

1) **le noeud**, qui représente un ensemble de ressources (ordinateurs, etc.);
2) **le noyau**, qui donne aux objets ODP l'accès aux ressources;
3) **la capsule**, qui est l'unité de gestion de ressource par le noyau;
4) **la grappe**, qui est l'unité d'activation de sauvegarde et de migration;
5) **le canal**, qui réalise la liaison entre objets et met en oeuvre les communications.

Mise en oeuvre de l'architecture

La spécialisation de ce modèle générique peut conduire à des modèles de références spécifiques. L'exemple le plus probant est issu du monde des opérateurs de télécommunications, qui travaillent sur une architecture de réseaux intelligents, nommée TINA [Brown 94 et Stephani 95] (*Telecommunication Information Network Architecture*) .

Références bibliographiques

[BMODP 96] ISO / IEC IS 10746-x, «Basic Reference Model of Open Distributed Processing» :
ISO / IEC IS 10746-1 - ITU-T Rec. X901, Part 1. Overview and Guide to Use, May 1996 ;
ISO / IEC IS 10746-2 - ITU-T Rec. X902, Part 2. Foundations, January 1995 ;
ISO / IEC IS 10746-3 - ITU-T Rec. X903, Part 3. Architecture, January 1995 ;
ISO / IEC DIS 10746-4 - ITU-T Draft Rec. X902, Part 4. Architectural Semantics1995
ISO / IEC IS 14750 - ITU-T Draft Rec. X9nn, Open Distributed Processing - Interface Definition Language1997

[Brown 94] D.K. Brown, «Practical Issues Involved in Architectural Evolution from IN to TINA», 3rd International Conference on Intelligence in Networks (ICIN'94), October, 1994.

[CNIS 95] «Special Issue on ODP», Computer Networks and ISDN Systems, Vol. 27 (8).

[Stephani 95] J.B. Stephani, «Open Distributed Processing - An Architectural Basis for Information Networks», Computer Communications, Vol. 18 (11), pp. 849-862.

OMA

L'OMG (Object Management Group) est un consortium fondé en 1989 à l'initiative de grands constructeurs et éditeurs informatiques (Data General, Hewlett Packard, Sun, Canon, American Airlines, Unisys, Philips, IBM, etc.) afin de normaliser les systèmes à objets. Les spécifications de l'OMG contiennent un modèle objet, une architecture de référence l'OMA (*Object Management Architecture*) et un langage de définition d'interfaces (appelé IDL).

Le but principal de l'OMA est de fournir des mécanismes de coopération pour des applications actives sur différentes machines d'un environnement distribué et hétérogène.

Le modèle Objet

Le modèle de l'OMG définit une sémantique commune des objets pour spécifier leurs caractéristiques externes visibles de manière indépendante des implémentations. Le modèle objet de base [OMG 92a] intègre les notions d'objet, d'opération décrite par une signature, de valeur (non objet), de sous typage, d'héritage et de requête.

- **un objet** est une entité informatique identifiable et encapsulée, comportant un état non directement accessible et un comportement composé de plusieurs services susceptibles d'être appelés par un client, désigné par un identifiant immuable.
- **une opération** est une transformation qui s'applique sur un objet et qui permet de consulter et/ou modifier son état.
- **une requête** est l'invocation d'une opération sur un objet indiquant le sélecteur de l'opération et des valeurs de paramètres optionnelles, envoyées par un client.
- **une valeur** est une entité qui ne représente pas un objet et qui est non personnalisable par un identifiant d'objet.
- **un type** est une entité identifiée par un nom avec un prédicat associé caractérisant les valeurs ou les objets membres.
- **un sous type** est un type apparaissant comme une spécialisation ou un raffinement d'un autre type appelé sur-type.

- **une interface** est la description de l'ensemble des signatures des opérations qu'un client peut invoquer sur un objet.
- **l'héritage de type** est un mécanisme pour définir l'interface d'un type S en fonction d'un type T : si S est déclaré être un sous type de T, alors S hérite des opérations de T.

Le modèle d'architecture

L'OMA se compose des éléments suivants (cf Figure 92) :

- un bus logiciel de communication appelé ORB (*Object Request Broker*) ;
- les services objets (*Object Services*) [OMG 92b] ;
- les utilitaires communs (*Common Facilities*) [OMG 94] ;
- les objets d'application (*Application Objects*) ;

Les ORB (*Object Request Broker*) ou bus logiciels, ont été créés pour que les objets distribués sur le réseau puissent communiquer. La notion de courtier est alors apparu. Un objet client envoie une requête à un objet serveur, par l'intermédiaire d'un ORB, qui joue le rôle du courtier entre les deux. L'ORB est capable, grâce à un référentiel, de trouver sur le réseau l'objet capable de rendre le service demandé.

L'architecture adoptée pour les ORB est connue sous le nom de CORBA (*Common Object Request Broker Architecture*) [OMG 91]. C'est une infrastructure de communication pour des objets distribués hétérogènes collaborant localement (*co-located*) ou à distance (*remote*). Normalisée depuis 1992, pour faciliter l'interopérabilité entre objets de toute sorte, elle a de nouveau évolué en 1994. CORBA 2.0 [OMG 95] offre l'interopérabilité entre ORB différents. Les ORB s'appuient sur des services de communication de type RPC, MOM et les principaux ORB du marché sont DSOM (IBM), ORB-Plus (HP), object broker (Digital), Orbix (Iona Technologies), NEO (Sun), DCOM (Microsoft).

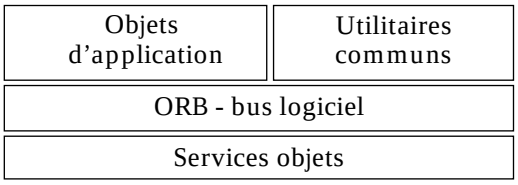

Figure 92 : L'architecture de référence de l'OMA

Les services objets standardisent la gestion des objets pendant leur durée de vie. Sont ainsi définis les services de gestion de noms d'objets, de gestion d'évènements, de gestion du cycle de vie des objets, de gestion d'associations entre objets, de gestion d'objets persistants et de gestion de transactions. Ce sont des objets qui sont définis par leur interface et les clients y accèdent par des requêtes via l'ORB. Leurs fonctionnalités peuvent être étendues ou spécialisées par héritage.

Les utilitaires communs offrent un ensemble de fonctions génériques, configurables pour des besoins spécifiques (telles que les fonctions d'impression ou d'aide). Il en existe deux types : les facilités horizontales utiles quel que soit le domaine d'application et les facilités verticales correspondants à des segments de marché divers (finance, santé, etc.). Quatre types de facilités horizontales ont été définies : l'interface utilisateur, les services de gestion d'information, les services de gestion de système, les services de gestions des tâches. Les objets d'application sont spécifiques à chaque application et peuvent offrir des services à d'autres applications. Les applications doivent donc définir des interfaces conformes à l'OMA. Ces objets sont aussi appelés des objets métiers.

Le langage d'interface (IDL)

IDL est le langage décrivant les interfaces que les clients appellent et que les implémentations d'objets fournissent. Or, pour assurer la communication dans un milieu réparti et hétérogène, les systèmes ont recours à des mécanismes d'encodage (*marshalling*) de l'opération distante à déclencher, des arguments à lui passer et des résultats éventuels. C'est pourquoi, les objets sont décrits dans un IDL (Interface Description Language) [OMG 95] qui sépare la description des interfaces d'accès à l'objet de son implémentation. Dans le modèle objet, cela revient à décrire les méthodes exportées des classes des objets. Les projections (*mapping*) sont le mécanisme de traduction d'une spécification IDL vers une langage d'implantation C ou C++. Les talons (*stubs*) sont des parties de codes générés automatiquement et sont utilisés pour les communications à travers l'ORB (cf. Figure 93). Les talons clients et serveurs sont dépendants de l'ORB (si on en change, il faut les recompiler) et sont générés avec la même version du compilateur. IDL diffère d'un langage de configuration dans le sens où il n'intègre pas de déclaration des opérations clients, ni de composants hiérarchiques au travers de la composition des interfaces.

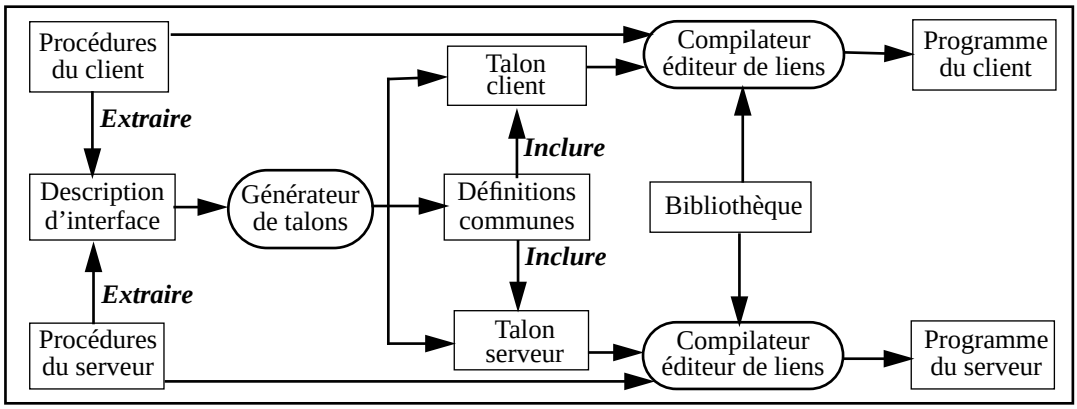

Figure 93 : Génération des programmes à partir des interfaces

Références bibliographiques

[OMG 91] Dec, HP, Hyperdesk, NCR, ODI, SunSoft, «The Common Object Request Broker : Architecture and Specification», OMG document No 91.12.1, Revision 1.1, X/ Open and OMG Eds, Framingham, USA, Décembre 1991.

[OMG 92a] Object Model Task Force, «The OMG Object Model», OMG draft No 92.xx.yy, Framingham, USA, Mars 1992, Site Internet : <http:/ / www.omg.org>.

[OMG 92b] Object Management Group, «Objects Services Architecture», OMG document No 92.8.4, Framingham, USA, Mars 1992.

[OMG 94] Object Management Group, «Common Facilities Architecture», OMG TC document No 94.11.9, Issue 3.0, Framingham, USA, Novembre 1994.

[OMG 95] Object Management Group, «CORBA 2.0 / Interoperability, Universal Networked Objects», OMG document No 95.3.10, Framingham, USA, Mars 1995.

Protocoles d'interactions

ActiveX

Microsoft a utilisé ses travaux sur OLE pour construire une nouvelle architecture ouverte, nommée de manière générique ActiveX, pour créer des contenus actifs et des applications pour l'Internet. Les contrôles ActiveX sont des composants logiciels qui incluent : les anciens contrôles OLE (cf. Figure 94), des applets Java et de nouveaux contrôles ActiveX (cf. Figure 95). A cela s'ajoute :

- **le registre**, qui est une base de données contenant des composants ActiveX et des informations concernant leur configuration ;
- **le SSPI** (*Standard Security Providing Interface*), qui sécurise l'invocation via COM ou DCOM d'un composant ActiveX.

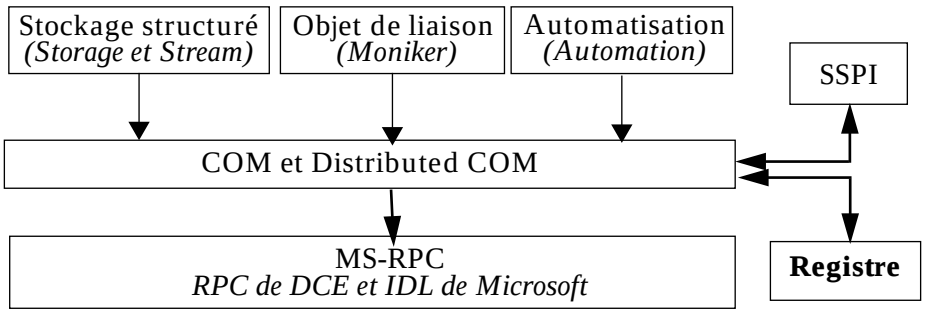

Figure 94 : Technologies de base d'ActiveX

ActiveX a été conçue pour prendre en compte les spécificités d'Internet. Ainsi, par exemple, l'explorateur internet de Microsoft est lui même considéré comme un contrôle ActiveX (cf. Figure 95). Cette nouvelle architecture lui permet ainsi d'héberger des contrôles ActiveX ou de fonctionner comme un contrôle ActiveX. Ainsi, le support des documents ActiveX lui permet d'ouvrir directement des documents bureautiques tels que Word ou Excel.

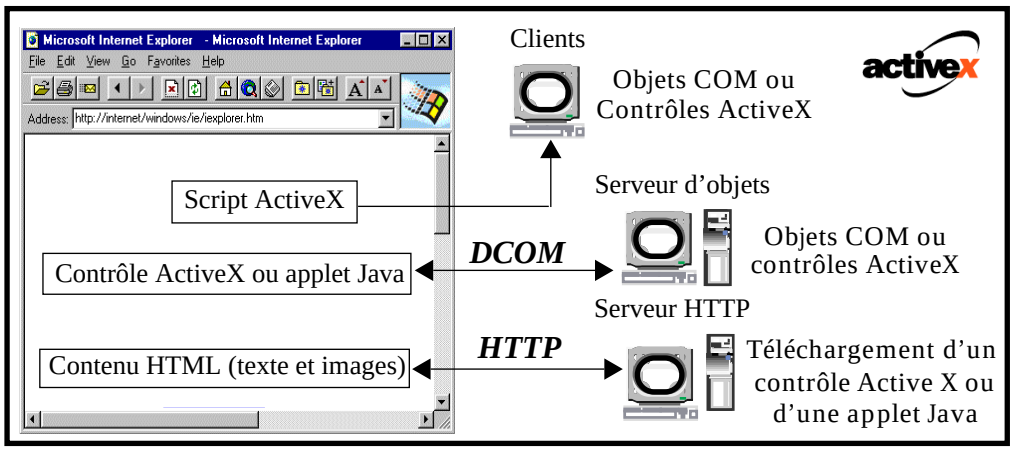

Figure 95 : ActiveX : l'architecture OLE de Microsoft adaptée à Internet

La mise en oeuvre des références Java dans Windows permet aux développeurs d'utiliser des contrôles ActiveX dans Java, en écrivant des fichiers «.java» et en ouvrant les interfaces COM des classes Java à l'aide d'une bibliothèque de types. Ces fichiers «.java» peuvent ensuite être compilés pour générer des fichiers «.class» au format de code Java. Les fichiers «.class» s'exécutent ensuite dans l'environnement Java sous Windows en

commençant par une vérification du code. Le code «.class» est ensuite optimisé pour l'interpréteur Java de la machine de l'utilisateur (cf. Figure 96).

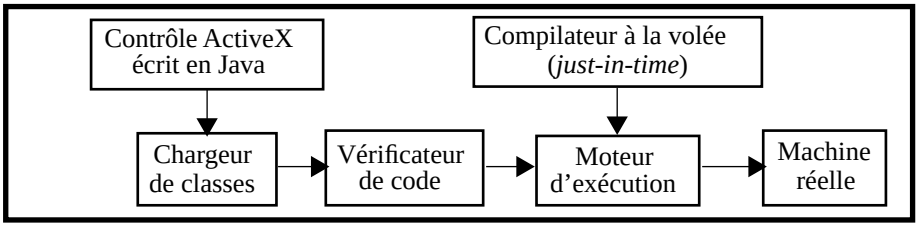

Figure 96 : L'architecture de la mise en oeuvre des références Java dans Windows

Référence bibliographique

Site internet http://www.microsoft.com/activex/

DCOM

Le protocole DCOM [DCOM 96] (*Distributed COM*), anciennement Network OLE, est un protocole de niveau application définissant des RPC [RPC 94] orientés objets (ORPC). Avec DCOM l'invocation transparente de serveurs OLE distants sur des machines hétérogènes est possible. The *IUnknow* Interface est remplacée par l'interface *IRemUnknown*, qui agit de la même manière mais à distance. Un nouveau type d'objet, l'exportateur de références (*Object Exporter*), a été créé. Il sauvegarde dans un cache et retourne aux clients distants les références sur les objets exportés par la machine.

Références bibliographiques

[DCOM 96] C. Kindel & N. Brown, «Distributed Component Object Model Protocol - DCOM/1.0»,.Internet Draft, May 1996.

[RPC 94] CAE Specification, «X/Open DCE: Remote Procedure Call», X/Open document C309, 1994, site : <http://www.osf.org/mall/dce/free_dce.htm>.

DDE

DDE [Williams 94] est uniquement utilisé pour l'échange de données entre applications qui s'exécutent sur une même machine. DDE est basé sur un modèle client serveur, où le client initie l'échange de données, après s'être connecté au serveur. La serveur établit alors un lien qui peut être de trois types : froid (*cold*), tiède (*warm*) ou chaud (*hot*) (cf. Figure 97).

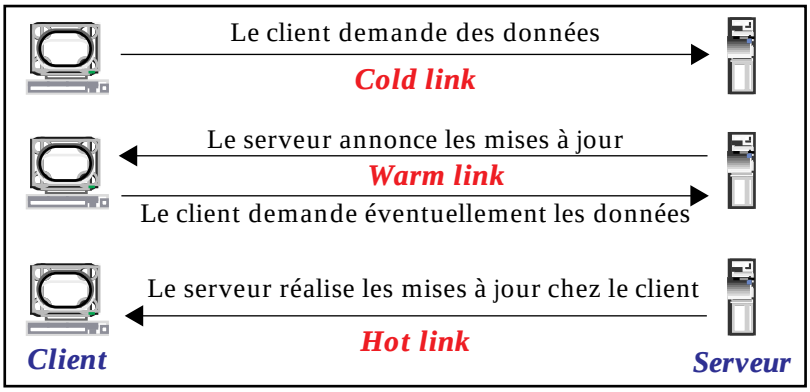

Figure 97 : Les trois types d'accès possibles à un serveur DDE

Dans le cas d'un «lien froid», le serveur joue un rôle passif, car il ne fait que répondre aux requêtes du client. Le client est donc chargé de gérer la politique de rafraîchissement et de cohérence des données. Lorsque le client et le serveur partage la responsabilité de

la mise à jour des données, on parle de «lien tiède». Le serveur notifie au client toutes les mises à jour, le client prenant alors la décision de les répercuter ou pas. Si le serveur se charge automatiquement des mises à jour chez le client, on parle de «lien chaud». Grâce à ces trois types de lien, il est possible pour chaque application cliente de mettre en place une politique adaptée de gestion de la cohérence des données.

L'organisation des données sur un serveur DDE est réalisée par une hiérarchie à trois niveaux. Chaque niveau gère un type d'entité :

- le premier niveau, la racine de l'arbre de hiérarchie, gère les services (*service*) fournis par le serveur.
- le second niveau gère les sujets (*topics*). Les sujets sont le plus souvent des documents composites (un document ouvert dans Word par exemple).
- le troisième niveau gère les articles (*item*). Un article peut être de n'importe quel type du moment qu'il est reconnu par le système d'exploitation et/ou l'environnement graphique.

DDE est une architecture client serveur où un client peut se connecter à de multiples serveurs et un serveur gérer plusieurs clients. Même si son architecture a été pensée pour faciliter l'échange de données, il est néanmoins possible pour un client d'exécuter un certains nombres de commandes sur le serveur.

Référence bibliographique

[Williams 94] A. Williams, «OLE 2.0 and DDE Distilled: A Programmer's Crash Course», Addison Wesley Ed., 1994.

DSOM

DSOM ajoute les fonctionnalités d'un ORB à SOM. Une application cliente DSOM invoque des méthodes sur des objets distants à travers un objet de procuration local (*proxy*) disponible dans l'environnement d'exécution. Ainsi, quand un objet distant est créé à travers l'ORB DSOM, une référence à un objet procuration local est retournée au client. Cet objet devient alors le représentant local de l'objet distant. Il reçoit les requêtes de services de l'objet distant à travers des invocations de méthodes et les transmet aux objets réels correspondants. Les objets distants ne sont alors plus implémentés sous formes de bibliothèques partagées, mais de processus. Au moment où l'objet procuration est créé, l'interface vers l'objet distant est obtenu par la base de données d'interfaces (*Interface Repository Database*) maintenue par le compilateur SOM.

La base de données d'implémentations (*Implementation Repository Database*) est utilisée, du coté serveur, pour retrouver l'information concernant l'implémentation d'un serveur. Ceci inclut sa machine hôte, son identificateur d'implémentation, la classe du serveur d'objets et le chemin nécessaire au chargement du code exécutable. L'environnement d'exécution du serveur comprend un certain nombre d'objets dont les plus importants sont l'objet adaptateur SOM et l'objet serveur d'une application spécifique.

L'objet adaptateur SOM (*SOM Object Adapter*) définit l'interface principale entre le programme serveur et l'environnement d'exécution DSOM. Après que la communication avec le serveur ait été établie, l'objet SOMOA reçoit les requêtes de service des clients. En coopération avec l'objet serveur d'une application spécifique (qui est généralement une instance de la classe prédéfinie *SOMDserver*) il crée et/ou résoud les références DSOM à des objets locaux et distribue les méthodes sur ces objets. L'objet serveur gère les services de création et de destruction d'objets par des applications clientes et offre des services de gestion d'objets nécessaires aux clients.

Références bibliographiques

[Benantar 96] M. Benantar, B. Blakey & A.j. Nadalin, «Approach to Object Security in Distributed SOM», IBM System Journal, Vol 35(2), pp. 192-203, 1996.

[Orfali & al. 95] R. Orfali & D. Harkey «The Server Side of Corba», OS/2 Developer Journal, July/August 1995, pp. 26-30, 1996.

OLE

La définition et l'implémentation des services OLE ont tellement évolué ces dernières années [Brockschmidt 95a], qu'il est difficile aujourd'hui de s'y retrouver. C'est avant tout un standard de communication entre applications provenant de différents vendeurs et un moyen de gestion des relations complexes entre des documents.

Dans OLE, un service (aussi appelé un composant) est fourni à travers un ou plusieurs objets, chaque objet consistant en un groupe logique offrant certaines spécificités du composant. Ces objets forment les liens de communication entre l'utilisateur de ces objets (du code) et le fournisseur de ces objets. Un client qui maintient la cohérence d'objets persistants est appelé un conteneur (*container)*. Le fournisseur d'un composant (et les objets qui le constituent) est appelé un serveur. Pour résumer : un client utilise un composant offert par un serveur et la communication se fait à travers des objets OLE.

OLE 1.0

La première version d'OLE, OLE 1.0, est apparue avec Microsoft Windows 3.1 en 1991 et utilisait les services de DDE. Des outils d'encapsulation (*embedding*) et de liaison (*linking*) des objets dans des documents étaient alors disponibles. La première limitation venait du fait que les liens pouvaient uniquement être établis avec des données stockées sous forme de fichiers sur l'ordinateur. Si l'objet était encapsulé dans un autre document, il n'existait aucun moyen pour le client d'activer l'objet ou de faire des mises à jour automatiques. La seconde limitation avait trait à la gestion du stockage et au transfert des données entre conteneur. Il fallait que chaque objet soit chargé en mémoire pour que les liens puissent être activés, ce qui était souvent rédhibitoire.

OLE 2.0

OLE 2.0 [OLE 94], successeur logique d'OLE 1.0, devait en améliorer les performances et les fonctionnalités. Mais, tout en assurant la compatibilité ascendante, il dépassa largement le champ de la gestion de documents composites pour devenir une architecture générique de services à part entière [Brockschmidt 95a]. OLE 2.0 s'appuie sur les fonctions de bas niveau de COM (cf Figure 98) et non plus sur DDE.

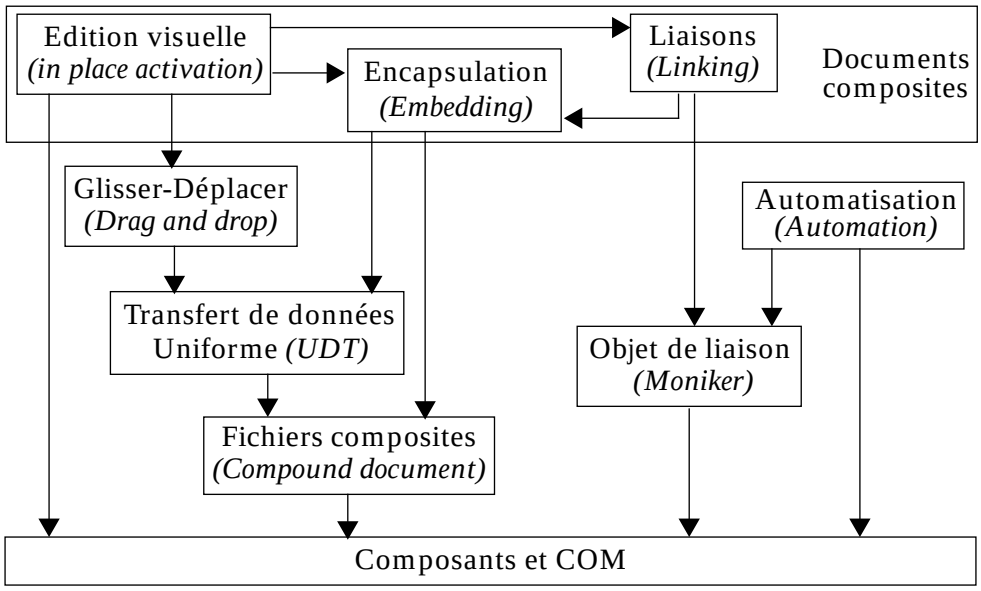

Figure 98 : L'architecture fonctionnelle OLE/COM

Les principaux services offerts par OLE sont :

- *OLE compound document* est un service de gestion de documents composites créés par n'importe quelle application OLE et orienté utilisateur. Il dispose d'une nouvelle fonctionnalité, appelée activation sur place (*in place activation*), décrivant le processus d'édition d'un objet serveur à l'intérieur d'un conteneur OLE. Cette fonctionnalité permet d'éviter d'activer et de changer d'application pour éditer un conteneur. On peut donc visualiser un document du tableur Excel dans le traitement de texte Word sans quitter Word (le menu de Word étant automatiquement remplacé par celui d'Excel).
- *OLE automation* est un service de gestion de macro-commandes pour le pilotage d'une application par une autre. La sous-traitance d'une tâche à une autre application devient alors possible. Ce service est plutôt utilisé par les serveurs.
- les *OLE controls* sont des objets de type *OLE CompoundDocument* contrôlés via des objets *OLE Automation*. Ils sont donc capables de réagir à des évènements extérieurs en exécutant des macro-commandes.
- *OLE drag and drop* est disponible dans *OLE compound document* et OLE *controls* sous trois formes : inter-fenêtres, inter-objet et inter-icones.
- OLE *structured storage* est ensemble d'interfaces et d'implémentations associées pour la création de support de stockage d'objets composites et partagés. On cherche ainsi à gérer des fichiers composites, de sorte à inclure dans un seul fichier physique une structure composée de deux types d'éléments : les fichiers de données appelés chaînes d'octets (*stream*) et les documents composites appelés répertoires (*storage*). De l'imbrication des chaînes d'octets et des répertoires découle une gestion hiérarchique des fichiers. Ainsi, un répertoire de niveau N décrit des fichiers de niveau N+1 correspondant à des répertoires et des chaînes d'octets.

Critique du modèle

Les technologies OLE de Microsoft, qui ne s'appliquent pas réellement à des objets mais à des composants propriétaires. Les composants OLE 1.0 ne contiennent aucun traitement et disposent uniquement d'un lien vers l'application source appelée lorsqu'ils sont accédés. Avec OLE 2.0, le mécanisme d'activation sur place permet de modifier un objet sans quitter l'application. Néanmoins, il n'existe aucune possibilité d'étendre des composants ou de modifier leur comportement. Il n'existe pas non plus d'héritage. Lorsqu'on édite un «objet» OLE 2.0, ce n'est pas une fonction spécifique qui est appelée, mais toute l'application propriétaire du composant. De plus, en pratique, on constate que l'activation de plusieurs composants OLE 2.0 créés par la même application entraîne l'exécution d'autant d'instances de l'application que de composants activés. L'accès à des fonctions spécifiques, sans lancer toute l'application est quand même réalisable avec *OLE Automation*. Cette fonctionnalité n'est offerte qu'au programmeur d'application et en aucun cas à l'utilisateur final. Remarquons enfin que l'appel d'une fonction OLE entraîne l'exécution de l'ensemble de l'application Serveur d'*OLE Automation.*

OLE 2.0 est orienté poste de travail car il utilise le mécanisme d'appel à distance LRPC (*Lightweight RPC)*, qui n'autorise pas de communication à distance mais seulement une requête sur le serveur.

Le plus grand avantage d'OLE est né du modèle de composant logiciel, nommé OCX, qu'il supporte. Un OCX est une combinaison de serveur OLE automation (service de gestion de macro-commandes pour le pilotage d'une application par une autre) et OLE d'activation locale (*In-process Server*). Un OCX définit des propriétés et des évènements standards sur lesquels il déclenche des opérations. Le marché des composants logiciels OCX est florissant, même si les OCX sont plus difficiles à développer que les composants logiciels Visual basic (nommés VBX). Les performances des OCX sont moins bonnes que celles des VBX, tout en utilisant plus de ressources. Enfin, la nouvelle génération

de composants logiciels de Microsoft, appelés ActiveX, sont un mélange de composants OCX et de services internet.

L'avenir

OLE est désormais un environnement unifié de services basés sur des objets, supportant l'adaptation des services offerts et l'extension de son architecture. Du fait de cette évolution (notamment due à COM), on ne parle plus d'OLE version 1 ou 2, mais simplement de l'architecture OLE.

Le futur son système d'exploitation Cairo, tel que le présente Microsoft, contiendra une nouvelle version d'OLE qui se décomposera en six groupes de fonctions :

- *OLE Document*, qui prendra la suite de celui existant actuellement en offrant des fonctions avancées de gestion de documents et de calcul ;
- *OLE Control* qui proposera des composants visuels ou non et réutilisables aussi bien à la conception qu'à l'exécution ;
- *OLE Automation* qui s'enrichira de nouvelles macro-commandes ;
- *OLE DB* qui sera utilisé pour le stockage de tous les types de données dans une base de données ;
- *OLE Transaction* qui servira à la coordination des composants OLE dans une application et régira les communications entre les différents objets à la manière d'une ORB ;
- *Network OLE* qui est un procédé de communication entre les objets sur un réseau (basé sur des RPC). Il réalise la distribution des composants au moyen de DCOM.

Interopérabilité Opendoc - OLE

Opendoc gère l'interaction des applications et ne définit pas de standard pour les documents comme OLE. Les composants Opendoc (ou *parts*) sont de grains plus fins qu'OLE 2 et ne nécessitent pas l'exécution d'applications complètes. Néanmoins, Novell distribue *ComponentGlue*, une passerelle logicielle bidirectionnelle entre OLE et Opendoc, mais uniquement dans l'environnement PC Windows et Macintosh. Avec *ComponentGlue* un conteneur Opendoc voit un objet OLE comme un composant Opendoc et vice versa. La couche de translation opère directement au niveau des API. Tout appel à l'API Opendoc, crée un serveur/conteneur OLE qui s'exécute sur une machine Windows 95 ou NT. Les composants Opendoc et les documents OLE peuvent partager le même conteneur Bento ou *DocFiles* de Windows. Plus encore, une correspondance directe est réalisée entre les services *OLE Automation* et les évènements sémantiques d'Opendoc. Le contrôle de composants à la fois OLE et Opendoc par des scripts conformes à OLE ou OSA est donc possible.

Références bibliographiques

[Brockschmidt 95a]	K. Brockschmidt, «OLE Integration Technologies», Dr Dobb's Special Report on «The Interoperable Objects Revolution», Miller Freeman Ed., Vol 19 (16), pp. 42-49, March 1995.
[Brockschmidt 95b]	K. Brockschmidt, «Inside OLE 2, Second Edition», Microsoft Press Ed., 1995.
[OLE 94]	Microsoft, «OLE 2 Programmer's Reference», Vol. 1 et 2, Microsoft Press Ed, 1994.
Site Internet	http://www.microsoft.com

Opendoc

Opendoc est une architecture distribuée de développement et d'échange de documents, conçue pour intégrer des outils orientés objets de gestion de documents composites. Les modules d'une application interopèrent de manière simple, en partageant la même inter-

face utilisateur. Les documents créés sont exploitables sur des plate-formes matérielles différentes, sans aucun service de conversion.

Ainsi, un document composite est représenté par un ensemble de composants (*parts*) imbriqués à partir d'un composant racine. Ces composants sont les briques de base que l'on assemble hiérarchiquement pour former des conteneurs (*container*). Les composants sont incorporés par copie ou liés par pointeurs. Chaque type de composant est associé à un éditeur pour créer et manipuler des instances du composant. On associe aussi à chaque type de composant une visionneuse pour la visualisation et l'impression du contenu du composant. D'un point de vue CORBA, un composant Opendoc n'est rien d'autre qu'un objet CORBA, qui utilise un ORB pour collaborer avec d'autres composants. L'ORB fournit un accès transparent aux composant distants et aux services CORBA (sécurité, nommage et mode transactionnel par exemple).

Opendoc dispose d'une interface standard pour visualiser et composer les éléments : les cadres (*frames*). Chaque cadre définit les frontières d'un type de composant. Si lors du chargement, ni éditeur, ni visionneuse ne sont disponibles, le cadre reste grisé. Dans le contraire, par simple clic, le clavier est affecté au pilotage de l'éditeur associé, réalisant ainsi une activation sur place (*in place activation*).

Opendoc gère le stockage et l'échange de documents composites grâce au gérant d'objets **Bento** d'Apple. Dans Bento, chaque objet est dans un conteneur. Un conteneur permet aux applications de sauvegarder et de retrouver une collection d'objets dans un unique fichier structuré, possédant des références ou des liens vers d'autres objets. Le format d'un conteneur est indépendant de la plate-forme. Un objet consiste en un ensemble de propriétés. Chaque propriété consiste en un ensemble de valeurs de types différents. Chaque objet possède un identificateur persistant unique et au moins une propriété (et celle ci doit avoir au moins une valeur). La description des objets est dynamique et il est possible de lui rajouter une propriété à tout instant. Les propriétés définissent un rôle pour les valeurs et le type d'une valeur décrit le type de la valeur. Bento gère aussi le numéro de version et le travail en groupe.

Le Transfert Uniforme de Données (*Uniform Data transfer*) à travers et dans une application est un service disponible dans Opendoc. Ainsi, la méthode d'invocation utilisée pour le stockage, peut aussi l'être pour le transfert des données via un couper-coller (*copy and paste*), un glisser-déplacer (*drag and drop*) ou un accès par référence (*linking*).

Opendoc dispose aussi d'un service d'automatisation (*Open Scripting Architecture*). OSA laisse les composants exporter leur contenu via des évènements sémantiques que tout langage de script peut invoquer. Ces commandes envoyées par l'intermédiaire d'évènements sémantiques (*Semantic Event*) opèrent sur un spécificateur d'objets (*Object Specifier*) qui fait le lien avec des objets visibles sur l'écran ou dans un contexte particulier. Un composant accessible par script doit offrir à l'environnement d'exécution la liste des objets qu'il contient et les opérations qu'il supporte.

Références bibliographiques

[Orfali & al. 96] R. Orfali, D. Harkey & J. Edwards, «The Essential Distributed Objects Survival Guide», Wiley Ed., pp. 339-424, 1996.

[Rush 95] J. Rush, «Opendoc», Dr Dobb's Special Report on «The Interoperable Objects Revolution», Miller Freeman Ed., Vol 19 (16), pp. 30-35, March 1995.

Site Internet http://www.cilabs.org

Environnements

ANSA

Le projet ANSA *(Advanced Networked Systems Architecture)* [ANSA 89 & 93] spécifie une architecture de système favorisant le développement et l'intégration d'applications distribuées.

Le modèle d'architecture

Son architecture est basée sur un ensemble de cinq modèles, chacun offrant des concepts et des règles :

1) le modèle de l'entreprise *(Enterprise Model)* pour exprimer les limites du système, ses règles et son utilité. Il décrit l'organisation de l'entreprise et ses changements ;
2) le modèle d'information *(Information Model)* pour exprimer la signification de l'information distribuée et des processus qu'on lui applique ;
3) le modèle de traitement *(Computational Model)* exprimant la décomposition fonctionnelle de l'application en unités de distribution ;
4) le modèle d'ingénierie *(Engineering Model)* décrivant les composants et les structures dont on a besoin pour supporter la distribution (les fonctions de l'infrastructure) ;
5) le modèle technologique *(Technology Model)* qui décrit la construction de l'application en terme de composants, avec des règles de conformance pour sa réalisation.

Les deux concepts clefs d'ANSA sont :

1) la négociation à l'interface entre le système et les composants. Pour cela, l'utilisation de négociateurs par les clients pour localiser les serveurs de ressources et les services disponibles est systématique ;
2) la fédération, pour gérer l'hétérogénéité de l'autorité dans les systèmes distribués. La sécurité, le nommage, le routage sont maintenus dans un contexte local en traitant un objet comme unité d'encapsulation des politiques de gestion. L'interopérabilité est offerte par négociation aux interfaces et décision de coopération entre composants.

Mise en oeuvre de l'architecture

Dans ANSA, toutes les données sont considérées comme distantes et un composant donné ne dispose pas d'un accès direct à un autre composant. Tous les services sont réalisés à l'aide de négociateurs (*trader*) en utilisant un modèle client-serveur basé sur le paradigme de communication RPC [Birrell & al. 84].

ANSAware [ANSAware 93] est un environnement commercial de programmation distribuée fondée sur les principes d'ANSA. Il fournit aux programmeurs un ensemble d'outils de développement et des librairies de fonctions et de services. L'unité de base pour la distribution est le noeud qui comprend une ou un groupe de machines hétérogènes (ou non) utilisant des systèmes d'exploitation hétérogènes (ou non).

La gestion des processus, appelés capsules, se fait sur les noeuds et les interactions entre capsules sont transparentes à la localisation. Les invocations de capsules peuvent être synchrones, asynchrones et synchrones non bloquantes. ANSAware fournit son propre protocole d'échange de messages MPS *(Message Passing System)* supportant le protocole de haut niveau REX *(Remote Execution Protocol)*.

Enfin, un langage de programmation appelé DPL *(Distributed Processing Language)* est disponible pour créer des applications en cachant la distribution et l'hétérogénéité. Il est alors possible de générer du code C et plus récemment C++. Les interfaces de service des objets sont spécifiés en IDL.

Références bibliographiques

[ANSA 89] Advanced Networked Systems Architecture, «Reference Manual», Architecture and Projects Management Ltd., Poseidon House, Castle Park, Cambridge CB3 ORD, UK, July 1989.

[ANSA 93] Advanced Networked Systems Architecture, «Architecture and Design Framework», Technical Report 38, Architecture and Projects Management Ltd., Poseidon House, Castle Park, Cambridge CB3 ORD, UK, February 1993.

[ANSAware 93] ANSAware version 4.1 Manual Set, «RM.099.02 : An overview of ANSAware», «RM.100.02 : System Manager's Guide», «RM.101.02 : System Programmer's Guide», «RM.102.02 : Application Programmer's Guide», Architecture and Projects Management Ltd., Poseidon House, Castle Park, Cambridge CB3 ORD, UK, March 1993.

[Birrell & al. 84] A.D. Birrell & B.J. Nelson, «Implementing Remote Procedure Call», ACM Transactions on Computer Systems, Vol. 2 (1), 1984, pp. 39-59.

Athapascan

Athapascan est le support d'exécution du projet Apache (*Algorithme Parallèle et Partage de Charge*), dont le but est l'obtention d'une plate-forme portable pour applications irrégulières. Athapascan est portable, car il permet le découplage entre l'écriture d'une application et l'optimisation de son exécution (ce qui implique qu'on n'utilise pas les finesses d'un langage de programmation propriétaire non portable).

Dans Athapascan, le parallélisme est soit explicite, soit implicite et est basé sur le paradigme *diviser pour régner.* Les algorithmes procèdent en découpant une structure de données (pas forcément de manière équitable), puis s'appliquent récursivement sur les sous-structures et enfin effectuent une synthèse des résultats. Ainsi, à chaque étape de décomposition d'un calcul, un algorithme adapte le grain d'exécution (calcul du grain et ordonnancement des sous calculs). Les calculs de décomposition font l'objet de régulation de charge.

La bibliothèque Athapascan est structurée en deux couches principales :

- Athapascan-0 qui définit une machine abstraite qui offre toutes les fonctionnalités pour effectuer des appels synchrones et asynchrones de procédure à distance. Athapascan-0 n'offre pas de migration et est indépendant des protocoles de communication.
- Athapascan-1 qui offre différents opérateurs pour effectuer un ordonnancement dynamique. Ainsi, à chaque appel à effectuer en parallèle est associé une décision de degré de parallélisme et de placement.

Tous les opérateurs d'Athapascan-0 sont disponibles dans Athapascan-1, la seule différence se situant au niveau du placement qui est explicite dans l'un et implicite dans l'autre.

Concrètement, un programme Athapascan composé de plusieurs tâches. Une tâche est un processus Unix sur lequel sont définis plusieurs points d'entrée. Chaque point d'entrée actif est appelé un fil d'exécution et plusieurs fils peuvent être associés à un même point d'entrée. En Athapascan-1, une seule application est exécutée sur tous les processeurs (programmation de type SMPD). Les tâches de l'application sont toutes des exécutions de ce programme.

Athapascan est basé sur les appels de procédure à distance (RPC). Chaque RPC active un processus léger, c'est pourquoi, l'application est modélisée par un graphe d'appel de pro-

cédures. Athapascan-0 est le serveur qui réagit à ces appels de procédures, qui sont en général distants, de manière synchrone ou asynchrone. Athapascan-0 utilise la multiprogrammation dans les noeuds de calcul (groupe de processus légers) pour placer les calculs sur un nombre donné de processeur.

Athapascan est basé sur des processus légers POSIX (sans préemption) et sur une interface de communication de type PVM. Un calcul distribué se déroule dans un ensemble hiérarchisé d'équipes de processus légers. La coopération entre processus légers est limitée aux membres d'une équipe et des équipes parentes. La coopération est basée sur l'envoi de messages entre processus légers, sur la communication par mémoire partagée répartie ou sur le partage de mémoire locale.

Actuellement, Athapascan est une bibliothèque écrite en C/C++, construite au dessus d'une couche de processus légers (conformes POSIX) et d'un noyau de communication (PVM ou MPI). Un outil de traces est aussi en cours d'implantation pour la réexécution déterministe, le déboggage et l'évaluation de performances des applications.

Références bibliographiques

[Christaller 94a] M. Christaller, «Athapascan-0a Control Parallelism Approach on Top of PVM», Proceedings of PVM User's Group Meeting, University of Tennessee, Oak Ridge, Juin 1994.

[Christaller 94b] M. Christaller, «Athapascan-0b for PVM: adding threads to PVM», Proceedings of EuroPVM User's Group, Oct. 1994.

[Christaller & al. 95] M. Christaller, J. Briat & M. Rivière, «Athapascan-0 : concept structurants simples pour une programmation parallèle efficace», Calculateurs Parallèles, Vol. 7 (2), pp. 173-196, 1995.

[Denneulin & al. 95] Y. Denneulin, J.M. Geib, C. Gransart & al., «SDI et régulation de charge», Parallélisme et applications irrégulières, Hermès Eds., Chapitre 9, pp. 203-237, 1995.

[Gautier & al. 95] T. Gautier, F. Guinand, J.-L. Roch & A. Vermeerbergen, «Regulation de charge et adapation de grain : Athapascan», Journées de recherche sur le placement dynamique et la répartition de charge: Applications aux systèmes répartis et parallèles, pp. 27-30, Université Pierre et Marie Curie, 1995.

Site Internet http://www-apache.imag.fr/apache

CAE

X-Open est un consortium de vendeurs de systèmes d'information, d'organisations d'utilisateurs et de sociétés de logiciels. L'environnement intégré CAE *(Common Application Environment)* [Dolberg 93] a été créé pour offrir un environnement de système ouvert (cf Figure 99).

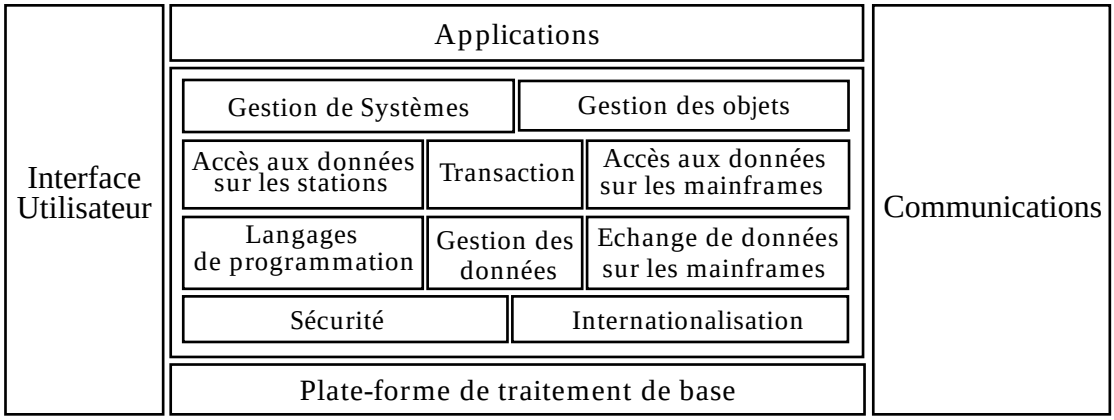

Figure 99 : L'architecture fonctionnelle mise en oeuvre dans X-Open-CAE

Mise en oeuvre de l'architecture

Les spécifications de CAE ont été développées en étendant les systèmes d'exploitation actuels (surtout les systèmes Unix), pour offrir une interface d'application unifiée. Un ensemble d'interfaces de services, indépendantes de l'implémentation, ont été définies, mettant la portabilité des applications au niveau du code source. Tous les membres du consortium supportent les interfaces plus connues sous le nom de XSI (X-Open System Interface) et comprenant plus de 1000 interfaces standards.

Références bibliographiques

[Dolberg 93] S.H. Dolberg, «X-Open in the 1990s», Open Information Systems, vol. 8 (1), January 1993, pp. 3-19.

[RPC 94] CAE Specification, «X/Open DCE: Remote Procedure Call», X/Open document C309, 1994, site : <http://www.osf.org/mall/dce/free_dce.htm>.

COOL V2

Le système Cool v2 (*Chorus Object Oriented Layer*) [Jacquemot 94] fournit une infrastructure pour développer des applications distribuées à base d'objets au dessus du micronoyau Chorus [Rozier & al. 90, 92]. Cool v2 est conforme aux spécifications CORBA et étend le C++ en introduisant les notions de distribution, de persistance et d'interopérabilité. Il dispose de son propre langage IDL, appelé COOL-IDL, compatible avec celui de Corba.

Son architecture s'appuie sur deux couches : une couche des mécanismes de base *(Cool Base)* et un couche langage qui offre une interface supportant un modèle spécifique *(Langage specific Run-Time)*. Trois techniques d'invocations distribuées sont possibles : l'utilisation de RPC, la migration d'objets et la mémoire virtuelle répartie.

L'espace d'adressage de Cool v2 est structuré par des clusters. Tout objet créé est alloué à un cluster, et ne peut le quitter que lors de sa destruction. Pour faire des invocations d'objets inter-cluster, on utilise un objet d'interface, représentant local de l'objet distant. La migration n'est disponible que pour des clusters et pas au niveau de l'objet. Des extensions à Cool v2 sont envisageables pour offrir de la répartition dynamique de charge [Chatonnay & al. 96].

Références bibliographiques

[Chatonnay & al. 96] P. Chatonnay, B. Herrmann, L. Philippe, F. Bourdon, P. Bar & C. Jacquemot, «Placement dynamique dans les systèmes répartis à objets», Calculateurs Parallèles, Vol. 8 (1), Mars 1996, pp 11-30.

[Lea & al. 93] R. Lea, C. Jacquemot & E. Pillevesse, «COOL: System Support for Distributed Programming», Communication of the ACM, Vol 36 (9), September 1993.

[Jacquemot 94] C. Jacquemot, «Chorus/Cool v2 Reference Manual», Technical Report, CS/TR-94-16, Chorus System, 1994.

[Rozier & al. 90] M. Rozier, V. Abrossimov, F. Armand & al., «Overview of the Chorus distributed Operating Systems», Technical Report, Chorus System, CS/TR-90-25, April 1990.

[Rozier 92] M. Rozier, «CHORUS», In Proc. of Usenix Micro-Kernels and Other Kernel Architectures, Washington, April 27-28, 1992.

Cords

La vision d'applications distribuées constituées de composants logiciels interagissants via des communications point à point, à conduit les membres du projet CORDS *(COnsortium for Research on Distributed Systems)* [Bauer & al. 94] à adopter un modèle orienté processus [Strom 86 et Bauer & al. 93], plutôt qu'orienté objet.

Le modèle d'architecture

La structuration des logiciels distribués se fait en utilisant des processus comme brique de base. Chaque processus est basé sur des concepts d'encapsulation et ne contient pas de parallélisme. Il n'existe pas de notion d'état global et toutes les données sont locales à un processus (il n'existe pas de variables partagées). Les données et les processus sont persistants. Un processus actif interagit avec un autre processus en créant un canal (*channel*) par l'intermédiaire de ports.

L'architecture fonctionnelle mise au point par CORDS comporte cinq couches (cf. Figure 100) :

1) **la couche d'application** qui gère les applications réparties développées, les outils de développement et les mécanismes de composition d'applications. Les applications sont considérées comme des services pour les autres applications.
2) **la couche d'environnement de service** (CSE) spécifie les services requis par les applications et les outils de la couche application. Elle cache les spécificités de la couche middleware et offre des interfaces d'accès garantissant l'indépendance par rapport aux évolution des couches basses. La mise au point de cette couche est l'un des objectif majeurs de CORDS. On y trouve les services de sécurité, les services de données, les services de communication, les services de présentation, les services systèmes et les services d'administration.
3) **la couche de Middleware** a pour but de s'interfacer avec les systèmes existants et éventuellement d'augmenter les services proposés si nécessaire.
4) **la couche de services de transport** nécessaire pour interconnecter les systèmes hétérogènes.
5) **la couche de services propriétaires** composée des ordinateurs, des systèmes d'exploitation et des services réseaux.

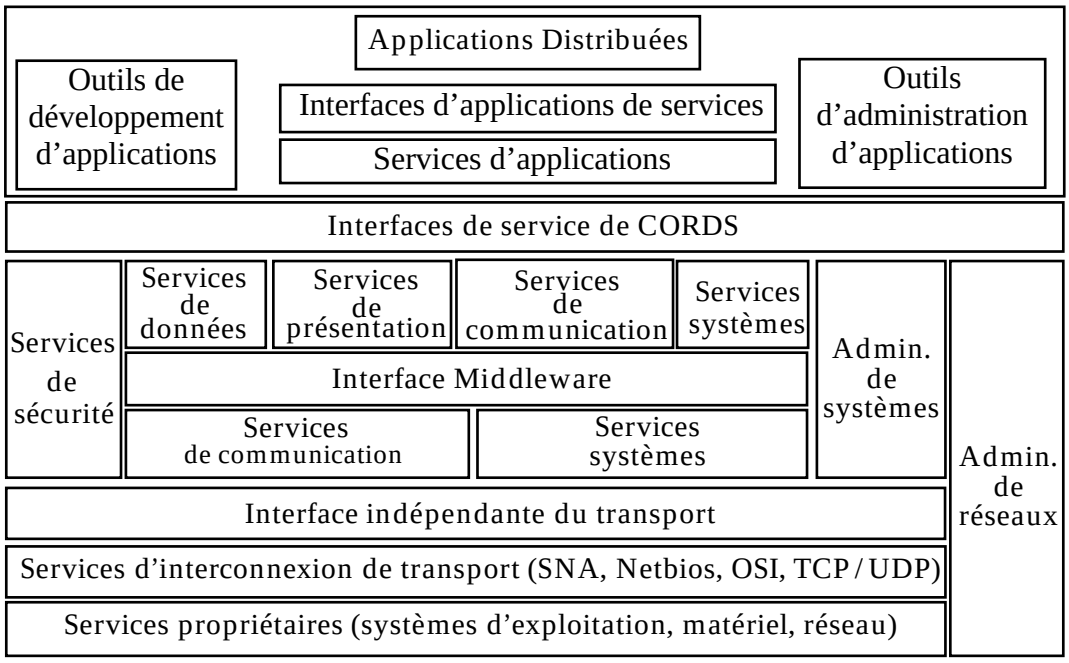

Figure 100 : L'architecture fonctionnelle mise en oeuvre dans CORDS.

Mise en oeuvre de l'architecture

Un prototype a été réalisé au dessus d'OSF/DCE [Attaluri & al. 93]. La mise en oeuvre a en fait consisté à intégrer un certain nombre de produits existants et à en créer de nouveaux. Ainsi, les principaux services intégrés sont :

- pour la couche présentation : X-Windows et les librairies systèmes graphiques GKS et PHIGS ;
- pour la couche Middleware OSF/DCE (CDS et les RPC), ainsi que le moniteur transactionnel Encina [Transarc 91] ;
- pour la couche de services d'interconnexion de transport TCP et UDP ;
- pour la couche de gestion des données, une multi-base de données donne une vue logique unique de différentes bases de données hétérogènes distribuées.

Références bibliographiques

[Attaluri & al. 93] G.K. Attaluri, D. Bradshaw, P.J. Finnigan & al., «Operation Jump Start: A CORDS Integration Prototype Using DCE», In Proc. of CASCON'93, IBM Center for Advanced Studies, Toronto, November 1993, pp. 621-636.

[Bauer & al. 93] M.A. Bauer, G. Strom, N. Coburn, D.L. Erickson, P.J. Finnigan, J.W. Hong, P. A. Larson & J. Slonim, «Issues in Distributed Architectures: A Comparaison of Two Paradigms», In Proc. of the International Conference on Open Distributed Processing, Berlin, Germany, September 1993, pp. 78-86.

[Bauer & al. 94] M.A. Bauer, G. Boochman, N. Coburn & al., «The CORDS Architecture: Version 2», IBM Centre for Advanced Studies, Toronto, 1994.

[Strom 86] R.E. Strom, «A comparaison of the Object-Oriented and Process Paradigm», SIGPLAN Notices, Vol. 21 (10), October 1986.

[Transarc 91] Encina Toolkit Programmer's Reference, Transarc Corporation, Pittsburg, 1991.

DCE

DCE *(Distributed Computing Environment)* [DCE 92] est un environnement de développement et d'exécution de systèmes distribués. Il intègre des technologies choisies par l'OSF (*Open Software Foundation*) pour l'interconnexion et la coopération de systèmes et d'équipements différents. L'OSF est un consortium qui à pour but de créer un ensemble de standards valables pour toute l'industrie de l'informatique.

Le modèle d'architecture

Dans DCE, l'unité de base d'opération et d'administration est la cellule. Une cellule est donc constituée d'utilisateurs, d'ordinateurs et de ressources orientées en général vers un objectif commun. Les principes de distribution s'appuie sur les mêmes concepts qu'une application locale, en conservant la sémantique des appels de procédure, grâce au paradigme de communication RPC (*Remote Procedure Call*) [Birrell & al. 84].

Les services distribués offerts par DCE sont (cf. Figure 101) :

- **le service d'annuaire distribué** *(Directory Services)*, utilisé pour la localisation des ressources. C'est une base de données globale requérant un nom hiérarchique pour retrouver l'adresse unique d'un objet. Ce service est conforme à la norme ITU-T X500 et ISO 9594. L'un de ses services est dédié à la gestion des cellules *(Cell Directory Service)*.
- **le service de gestion de temps** *(Time service)* pour recaler les horloges des machines entre elles au sein d'une cellule. Le serveur de temps global, qui est en général unique dans une cellule, se synchronise sur une source externe diffusant une heure de référence. Il donne ensuite l'heure aux serveurs locaux (s'ils existent) qui eux-mêmes rediffusent l'heure aux délégués qui se trouvent sur chaque machine de la cellule.

- **le service de sécurité** (*Security Service*) qui utilise un serveur d'authentification avec clefs publiques et privées. Ce service, basé sur Kerberos [Steiner & al. 88], sécurise les communications et contrôle l'accès aux ressources dans les systèmes distribués.
- **le service de gestion de fichier** (*Distributed File Service*). Il fournit à l'utilisateur un espace de nom global, des contrôles d'accès aux données étendus et sécurisés. D'autre part, DFS introduit la notion de *fileset*, pour structurer un nombre important de fichiers en groupes qui peuvent être dupliqués et contribuer à la répartition de charge.

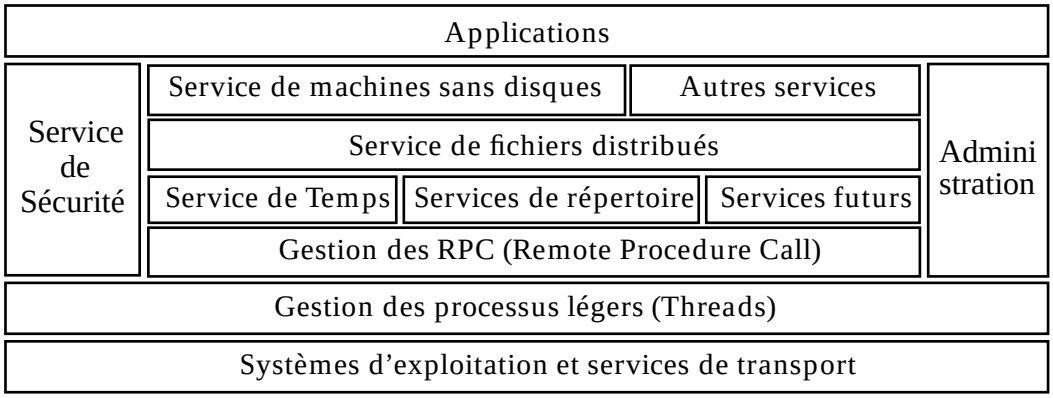

Figure 101 : L'architecture fonctionnelle mise en oeuvre dans OSF-DCE

Mise en oeuvre de l'architecture

Les RPC sont utilisés par les processus qui composent l'application pour communiquer entre eux. DCE intègre aussi un service de gestion des processus légers (conforme à POSIX 1003.4a, voire [Demeure 94]) prenant en charge leur création, leur manipulation et leur synchronisation au sein d'un processus.

La configuration minimale d'une cellule comporte le service d'annuaire des cellules, le service de sécurité et le service de gestion du temps. Les services de DCE fournissent un fonctionnement distribué suivant un modèle client serveur [Rosenberry & al. 93].

Références bibliographiques

[Birrell & al. 84]	A.D. Birrell & B.J. Nelson, «Implementing Remote Procedure Call», ACM Transactions on Computer Systems, Vol. 2 (1), 1984, pp. 39-59.
[DCE 92]	OSF, «DCE Administration Guide», Prentice Hall, 1992.
[Demeure & al. 94]	I. Demeure & J. Farhat, «Système de processus légers : concepts et exemples», Techniques et Science Informatiques, Vol. 13 (6), 1994, pp 765-795.
[Rosenberry & al. 93]	W. Rosenberry, D. Kenney & G. Fisher, «Comprendre DCE», Addison Wesley Ed., 1993, 279 pages.
[Steiner & al. 88]	J.G. Steiner, C. Neuman & J.I. Schiller, «Kerberos: An authentification Service for Open Network Systems», In Proc. of the Usenix Winter Conference, 1988, pp. 191-201.
site internet	http://www.osf.org/dce

Garf

L'environnement de Génération Automatique de Résistance aux Fautes (GARF) [Garbinato & al. 93] permet la programmation d'objets distribués, en séparant leur comportement distribué (objet comportemental), de leurs fonctionnalités (objet donnée). Les fonctionnalités d'un objet distribué sont définies par ses opérations, dont la sémantique ne se préoccupe pas des problèmes pouvant intervenir durant l'exécution distribuée. Le

comportement d'un objet distribué est défini par la partie de son code gérant les problèmes de désignation distante, de concurrence, de persistance et de résistance aux pannes.

La communication se fait grâce aux objets comportementaux suivants : le messager et l'encapsulateur. Il existe un messager sur chaque objet devant accéder à un autre objet. Chaque messager connaît un et un seul encapsulateur distant. Ainsi, la localisation d'un objet distribué est transparente pour le client, tant au niveau fonctionnel que comportemental et le messager fonctionne comme un proxy.

Associer un comportement à un objet revient à créer un objet comportemental, constitué d'un messager et d'un encapsulateur et à le lui associer. Un objet *futur* est destiné à contenir la réponse, lorsque celle ci sera disponible, quelque part dans le futur. Tant que l'appelant n'a pas besoin du contenu de la réponse, il peut manipuler l'objet futur comme tel. L'objet futur peur être explicite ou implicite (géré par l'environnement d'exécution système). GARF est une extension de l'environnement de programmation Smalltalk [Goldberg & al. 83] au moyen de classes d'objets. Une première version de GARF est opérationnelle sur un réseau de station Unix Sun.

Références bibliographiques

[Garbinato & al. 93] B. Garbinato, R. Guerraoui & K.R. Mazouni, «Distributed Programming in GARF», R. Guerraoui, O. Nierstrasz and M. Riveill Ed., Object Based Distributed Programming, Springer verlag, 1994.

[Goldberg & al. 83] A.J. Goldberg & A.D. Robson, «SMALLTALK-80: The language and its implementation», Addison Wesley, 1983.

Guide

Guide [Balter & al. 91] est un système réparti orienté objet et fournit un support de programmation d'objets répartis. C'est un système issu du centre de recherche Bull-Imag et qui a été lancé en 1986.

Le système Guide présente à l'utilisateur une machine virtuelle multi-processeurs dans laquelle le parallélisme est apparent et la distribution cachée. Une application Guide est composée d'objets distribués (ou non) sur différentes machines. L'espace des objets est organisé en unités logiques appelées domaines. Un domaine correspond à une application ou à une partie de l'application. Les objets peuvent être partagés entre domaines différents, ils peuvent donc être désignés dans plusieurs domaines. Les objets sont passifs et ne sont pas attachés à un site particulier. Chaque objet est désigné de manière unique pour permettre la persistance et le partage d'objets. Le partage d'objet est implicite et cache les mécanismes de communication.

Le parallélisme dans Guide est explicite et défini statiquement (à la compilation). Les entités actives (appelées activités) sont des flots d'exécution internes aux domaines. Les activités sont des processus légers dont le contexte est fourni par le domaine. Tous les objets d'un domaine sont accessibles aux activités de ce domaine. Les objets sont typés et persistants, ainsi une fois créé un objet existe tant qu'un autre objet le référence. La gestion d'activités au sein d'un domaine se fait grâce aux primitives cobegin/coend. Une activité mère peut ainsi créer plusieurs activités filles et reste bloquer en attente de la condition de terminaison. Localement, la synchronisation des méthodes objets est définie par des clauses déclaratives appelées des conditions d'activation ou des clauses comportementales. Dans Guide, une transaction est prise en charge par une activité unique en utilisant un verrouillage à deux phases.

L'implémentation de Guide repose sur la notion de machine virtuelle à objets. Cette dernière gère à la fois les processeurs virtuels et l'espace des objets. Un processeur virtuel est une machine qui interprète des appels de méthodes et qui s'appuie sur un ou plusieurs processeurs physiques. Un processeur virtuel gère une mémoire virtuelle qui représente une fenêtre sur la mémoire objet. On peut allouer ou libérer des processeurs virtuels.

Chaque machine dispose d'un démon Guide responsable du contrôle de l'activité locale. La communication est réalisée par sockets et files de messages.

Guide-2

Guide-2 [Chevalier & al. 96] est un système d'exploitation réparti objet opérant sur un ensemble de stations de travail reliées par un réseau local. L'architecture de Guide-2 [Krakowiak & al. 92, Balter & al. 93] repose sur le micro-noyau Mach [Accetta & al. 86] et est une extension de Guide-1.

Le modèle d'exécution de Guide-2 est celui des objets répartis et est basé sur une entité» logicielle spécifique : le job. Un job est un espace d'adressage virtuel disposant de ressources et de droits d'accès partagées entre différents fils d'exécution (*threads*) appelés des activités. Cet espace d'adressage réparti est composé d'espaces d'adressage locaux appelés les contextes. Un contexte est un espace d'adressage homogène local à un noeud et est implémenté par une tâche Mach.

Une application dans Guide-2 est généralement implémentée par un job et une ou plusieurs activités. Lorsqu'un job est lancé, le contexte de l'activité principale est créé, ainsi que le cluster contenant l'objet initial inséré dans le contexte. Lorsqu'une activité invoque la méthode d'un objet, le cluster dans lequel réside l'objet est transféré dans l'espace d'adressage virtuel du contexte courant de l'activité. Si l'objet ne peut être transféré localement (par manque d'espace mémoire par exemple), une invocation distante est alors réalisée.

Les objets dans Guide-2 sont passifs et sont manipulés par les activités définies dans le modèle d'exécution. Les objets sont regroupés dans des clusters pour augmenter les performances et limiter la surcharge système due à l'allocation de blocs disques et de pages mémoire. Par défaut les objets sont placés dans le même cluster que l'objet qui les a créés (principe de localité). Le cluster est à la fois l'unité de placement et de distribution dans Guide-2 et représente la granularité la plus fine possible pour l'équilibrage de charge [Damsgaard 96].

Guide-2 supporte plusieurs langages orientés-objets (C++, Eiffel) grâce au noyau Eliott (cf. Figure 102). Eliott fournit aux compilateurs de langages une abstraction simple du support de langage objet (tels que le partage et la persistance des objets). La surcharge système lors d'une invocation locale dans Guide-2 est de 4,4 µs et de 4 ms pour une invocation distante [Hagimont & al. 94]. C'est pourquoi les objets qui coopèrent doivent être placés dans le même cluster lorsque cela est possible. Des mécanismes de protection et de sécurité ont été rajoutés pour fiabiliser et administrer le système.

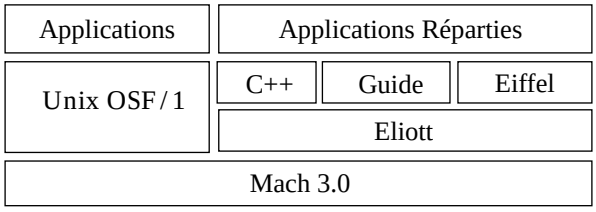

Figure 102 : Architecture fonctionnelle de Guide-2

Commentaires

L'un des résultats intéressants du projet Guide est que la programmation par objets persistants avec localisation transparente, bien que très pratique, ne facilite ni la mise au point de l'application, ni sa configuration [Riveill 94]. Néanmoins, l'utilisation du micro-noyau Mach a permis de fiabiliser le système (notamment grâce à la gestion de la pagination des segments de mémoire partagée) et de bénéficier de plusieurs flots de contrôle dans un même espace d'adressage (gestion des processus légers). Le système Guide

a donné naissance à un produit industriel commercialisé par Bull qui se nomme **Oode [Bull 94]**.

Références bibliographiques

[Accetta & al. 86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian & M. Young, «Mach: A New Kernel Foundation For Unix Development», In Proc. of the Usenix 1986 Summer Conference, June 1986.

[Balter & al. 91] R. Balter & al., «Architecture and implementation of Guide, an object oriented distributed system», Computing Systems, May 1991.

[Balter & al. 93] R. Balter, P.Y. Chevalier, A. Freyssinet & al., «Is the Microkernel Technology Well Suited for the Support of Object-Oriented Operating System: The Guide Experience», Proceedings of the 2nd Symposium on Microkernels and Other Kernel Architecture (MOKA), pp. 1-11, San Diego, California, September 93.

[Bull 94] Bull Open Software System, «Oode : une Plate-forme Objet pour les Applications Coopératives», Actes des journées d'études sur la répartition et le parallélisme dans les systèmes d'information, Afcet Eds., Paris, France, pp. 253-266, 22-23 Novembre 1994.

[Chevalier & al. 96] P.-Y. Chevalier, D. Hagimont, J. Mossière & X. Rousset De Pina, «Le système réparti à objets Guide», Technique et Science Informatiques, Vol 15 (6), pp. 801-830, 1996.

[Damsgaard 96] C. Damsgaard Jensen, «Fine grained load distribution in object based systems. An experiment with grain sizes in Guide 2», Calculateurs Parallèles, Vol. 8 (1), pp. 31-48, 1996.

[Hagimont & al. 94] D. Hagimont, P.-Y. Chevalier, A. Freyssinet & al., «Persistent Shared Object Support in the Guide System: Evaluation and Related Work», OOPSLA'94, ACM SIGPLAN Notices, Vol. 29 (10), pp. 129-144, Portland, Oregon, USA, October 1994.

[Krakowiak & al. 92] S. krakowiak & Rousset De Pina, «Architecture du système Guide 2», rapport de recherche IMAG, 1992.

[Riveill 94] M. Riveill, «GUIDE : Un langage à objets pour la programmation concurrente», Calculateurs Parallèles, «les langages à objets», No 22, Juin 1994, pp 99-121.

site:internet http://www.imag.fr.

ISA

Le projet ESPRIT ISA *(Integrated System Architecture)* No 2667, a permis la réalisation d'un support de développement d'applications distribuées [Beguin 90], qui dispose d'outils de transparence sélective.

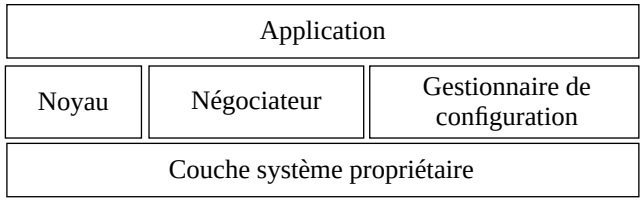

Figure 103 : L'architecture fonctionnelle de Esprit-ISA

Le modèle d'architecture

L'architecture d'ISA s'appuie sur la notion d'objets et est basée sur ANSA. L'architecture est constituée de trois couches (cf. Figure 103). Entre l'environnement matériel hétérogène et les applications distribuées, s'intercale une couche spécifique formant la plate-forme d'interconnexion. Cette plate-forme dispose de trois types de transparence : la transparence d'accès (une seule syntaxe indépendante de la localisation), de localisation (l'implantation des objets se fait sur n'importe quelle machine) et des accès concurrents (la gestions des conflits d'accès).

Mise en oeuvre de l'architecture

La plate-forme est basée sur l'utilisation des noyaux, qui sont les représentants des couches basses des machines. Ils interagissent entre eux pour former une plate-forme. Les capsules (ensemble d'objets constituant l'application ou une partie de l'application) communiquent entre elles par des interfaces qu'elles exportent ou importent. Un objet particulier, appelé négociateur *(trader)*, effectue les opérations de contrôle d'accès et de correspondance entre les interfaces importées et exportées. Il fournit ensuite un référence du serveur au client, qui peuvent alors directement communiquer entre eux.

Référence

[Beguin 90] A. Beguin, «Workshop Européen sur les Sytèmes Bureautique Répartis (Projet Esprit ISA tâche A01)», Rapport Technique RP/SPT/SCE/83, SEPT Caen, France, Mai 1990.

Java

L'architecture d'applications client-serveur de Sun Microsystems est basée sur un langage de programmation appelée Java, sur un système d'exploitation minimal nommée Java OS et sur un modèle de composants logiciels portant le nom de Java Beans.

La plate-forme Java

La plate-forme Java (cf. Figure 104) s'étend donc de la couche système aux API définies pour le développement d'applications spécifiques. La connexion aux bases de données à partir d'un programme Java est réalisée grâce à JDBC. La communication entre programmes Java distants s'appuie sur l'invocation de méthodes distantes (*Remote Method Invocation*) et le nommage est réalisée avec JNDI (*Java Naming and Directory*).

La définition des interfaces est réalisée avec un langage IDL propriétaire, étendant celui de CORBA. L'exécution d'un programme Java se fait soit sur un système d'exploitation minimal appelé Java OS, soit sur une machine virtuelle interprétant le code Java (et éventuellement l'optimisant à la volée pour la plateforme cible).

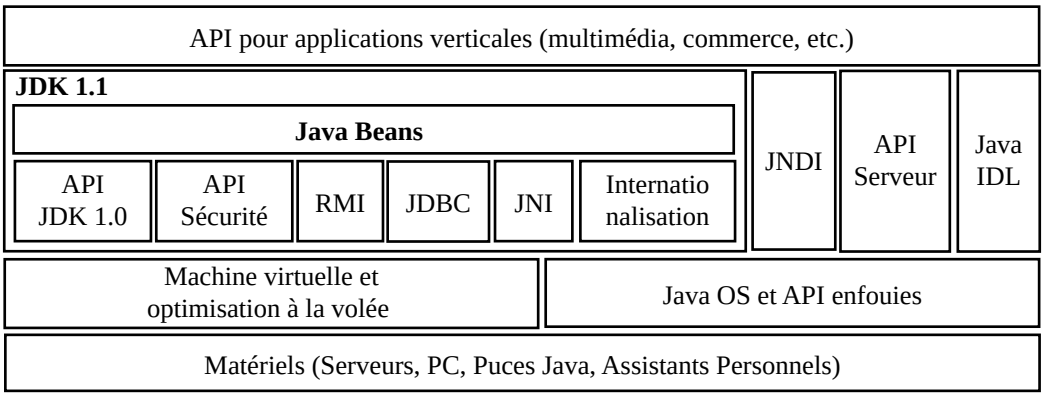

Figure 104 : Description de la plate-forme Java

Les langage Java

Java [Gosling & al. 96] pierre angulaire de l'architecture, est un langage orienté objet, proche de C++ pour sa syntaxe et d'Objective C pour ses interfaces. Java est un langage interprété et portable qui s'appuie une machine virtuelle spécifique [Lindholm & al. 96]. Le choix de la machine virtuelle a été guidée par des considérations de sécurité, de sûreté de programmation, de portabilité, de distribution et de performance. C'est pourquoi :

- Java supprime l'utilisation des pointeurs. Lors de la création d'un objet (méthode *new*), une référence sur cet objet est retourné et pas un pointeur ;

- Java réclame un typage fort. Il n'existe pas de conversion automatique de types ;
- Java gère lui même la mémoire grâce à un ramasse miettes (*garbage collector*) ;
- Java n'offre pas l'héritage multiple et le remplace par la notion d'interface. Avec une interface, on spécifie des protocoles et des comportements ;
- Java gère les processus légers (*thread*) à travers la notion d'objet actifs ;
- le résultat de la compilation d'un programme Java est un fichier qui contient des pseudo-codes interprétables (*Byte Code* ou code octet) par une machine virtuelle quelle que soit l'architecture matérielle et le système d'exploitation (*write once, run anywhere*) ;
- le chargement du code Java est dynamique et la compilation à la volée du code octet en mémoire dans le langage propre à la machine augmente sensiblement les performances d'exécution.

Il existe plusieurs versions du langage Java et de la machine virtuelle adaptée. La version 1.1, n'a par exemple, pas grand chose à voir avec la version 1.0 (si on considère le nombre de classes et de services qui ont été rajoutées). La prochaine version annoncée est la 1.2 et sera produite courant 1998.

Java et le WWW

Sur le WWW les programmes Java qui sont téléchargés et qui s'exécutent sur le poste client sont appelés des applets et ceux qui s'exécutent sur le serveur sont appelés des servlets.

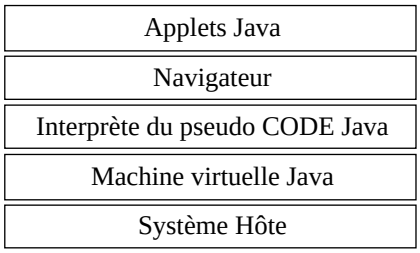

Figure 105 : Architecture Logicielle d'exécution d'un programme Java sur un navigateur

Du coté client, le mode d'exécution est le suivant (cf. Figure 105) : l'applet Java précompilé en Byte-code est téléchargé à partir d'un navigateur Web, puis exécuté sur la machine virtuelle s'appuyant sur la machine hôte. Le mode d'exécution est le même sur un serveur, à la seule différence que le programme Java n'est pas téléchargé.

Java Beans

Aujourd'hui, les applets Java fournissent un modèle de composant statique. Ainsi, les applets ne peuvent interagir avec d'autres applets sur une plate-forme Web, par exemple. Les Java Beans forment une architecture de composants réutilisables pour des plate-formes Java. Les Java Beans ont été créés pour que des interactions dynamiques puissent avoir lieu entre applets et sont fabriqués à partir d'une API générique, indépendante des plate-formes.

Sun Microsystems a publié les spécifications des Java Beans, qui définissent une série d'API découpés en cinq parties principales :

- **la gestion des évènements** (*Event Handling*) : API permettant à un composant de communiquer et d'envoyer des évènements à d'autres composants.
- **la gestion des propriétés** (*Properties*) : API représentant les attributs tels que la couleur de fond, ou la police de caractère à utiliser.
- **la gestion de la persistance** (*Persistence*) : API stockant un Java Bean comme une

partie de l'état d'un parent conteneur.

- **l'introspection** (*Introspection*) : API permettant que des outils de construction d'applications analysent les composants. Des API de bas niveaux laissent les outils connaître les méthodes, les interfaces, les variables associées à une classe Java. Les API de haut niveaux, quant à elles, déterminent quels évènements, propriétés ou méthodes sont exportés par un Java Bean donné.
- **la construction d'application** (*Applications Building*) : API permettant aux Java Beans de fournir des éditeurs de composants paramétrables (*Customizers*) pour les utiliser avec des outils de construction d'applications. Ces éditeurs de composants paramétrables sont séparés des Java Beans, c'est pourquoi, ils sont nécessaires uniquement durant le développement de l'application et ne doivent pas être téléchargés lorsque l'application créée est exécutée.

L'interopérabilité avec les modèles de composants déjà existant est résumé dans le Tableau 47.

Tableau 47: Interopérabilité d'un Java Bean avec Corba, ActiveX et Opendoc

Java Beans avec	Moyen d'interopérabilité
Opendoc	le composant OpenDoc (*parts*) est inclu dans un conteneur OpenDoc (*container*)
ActiveX / OLE / COM	une passerelle avec les OCX pour permettre l'exécution d'un Java Bean dans un conteneur ActiveX / OLE / COM (par exemple un Java Bean agira comme un serveur OLE automation)
CORBA	le support de corba IDL et du protocole IIOP pour intégrer les Java Beans avec les services CORBA.

Références bibliographiques

[Gosling & al. 95] J. Gosling & H. McGilton, «The Java Language Environment», White Paper, 1995.

[Gosling & al. 96] J. Gosling & K. Arnold, «The Java Programming Language», The java Series, Addison Wesley Eds, 1996, ISBN : 0-201-63455-4.

[Lindholm & al. 96] T. Lindholm & F. Yellin, «The Java Virtual Machine Specification», The java Series, Addison Wesley Eds, 1996, ISBN : 0-201-63451-1.

Site internet http://java.sun.com/

Mars/PM^2

PM^2 (*Parallel Multithreaded Machine*) est une plate-forme d'exécution portable d'utilisation de processus légers (ou activités) pour des environnements distribués hétérogènes. Plus généralement, c'est un environnement de programmation à grain fin sur un réseau de station de travail. Il supporte la migration d'activités, necessaire pour l'exécution et le placement dynamique d'applications irrégulières.

Le modèle de programmation est basée sur une double multiprogrammation : PVM est utilisé pour la gestion des processus et Marcel pour la gestion des activités. La bibliothèque Marcel a été créée pour étendre le modèle POSIX et offrir la migration et la gestion des piles de processus légers. La migration d'activité est un mécanisme transférant une activité d'un noeud vers un autre. Le modèle d'ordonnancement des processus légers est basé sur un mode préemptif avec priorités. Ainsi, tout processus léger perd le droit d'utilisation du processeur dès qu'il a consommé son quantum de temps. La vitesse d'exécution d'un processus léger dépend donc de sa propre priorité et de la somme des priorités des autres processus légers s'exécutant à l'intérieur d'un processus PVM. De plus, un contrôle de l'ensemble des priorités de tous les processus PVM garantit l'égalité sur tous les sites de la somme des priorités des processus légers locaux. Le maintien de cette égalité s'apparente à un équilibrage de charge.

L'invocation d'un service dans PM^2 est prise en charge à distance par un processus léger en mode synchrone ou asynchrone. Un nouveau type de RPC, les QuickLRPC, offrent une prise en charge de la requête directement par le processus de réception, sans création d'une nouveau processus léger. Par défaut, toute activité est considérée comme non migrante. La migration explicite est réalisée entre processus PVM identiques, c'est pourquoi seule la pile du processus léger est migrée. Dans PM^2, c'est donc au programmeur de garantir, lorsqu'il déclare une activité migrable, qu'elle n'accédera plus à des données globales à un processus. Pour aider le programmeur, PM^2 dispose d'une couche de régulation de charge basée sur les priorités et utilisant la migration des processus légers (LBMP) et d'une console graphique d'administration (Xpm2). PM^2 a été porté sur de nombreuses architectures (Sun/Solaris, Dec/OSF1, Linux, IBM-SP, etc.).

Mars, conçu au dessus de PM^2 est un environnement d'exécution multi-applications et multi-utilisateurs. Mars s'exécute en concurrence avec d'autres applications non Mars. Les objectifs de Mars sont :

- la portabilité : aucune modification n'est réalisée au niveau du système d'exploitation ;
- la protection : aucun accès ne se fait dans un mode super-utilisateur ;
- la transparence : la disponibilité et la répartition de charge ne sont pas gérées par l'utilisateur ;
- l'extensibilité : Mars dispose d'une structure hiérarchique à deux niveaux où un ordonnanceur global (GMS) contrôle et assure la coopération entre un ensemble de serveurs de groupes (GS). Chaque GS maintient un pool de noeuds (sélectionnés selon des critères prédéfinis) pris en charge par un gestionnaire de noeud (NM). Les NM informent les GS, ce qui permet au GS d'ordonnancer les applications Mars sur leurs propres noeuds ou sur ceux d'autres GS grâce au GMS.

Le modèle de programmation de Mars est SPMD. L'une des caractéristiques de Mars est qu'outre les fonctions de répartition de charge, son ordonnanceur ajuste de manière adaptative le nombre de processus d'une application donnée en fonction des noeuds disponibles.

Références bibliographiques

[Denneulin & al. 96] Y. Denneulin, R. Namyst, J.-M. Geib & J.F. Méhaut, «LBMP: Load Balancing with Migration Directed by Priorities», Proceedings of the ESPPE'96, Alpes d'Huez, Avril 1996.

[Geib 96] J.-M. Geib, «Processus Légers Distribués et Régulation de Charge», Actes de l'école thématique CNRS placement dynamique et répartition de charge, Presqu'île de Giens, pp. 89-102, Juillet 1996.

[Hafidi & al. 96] Z. Hafidi, E.-G. Talbi & J.-M. Geib, «Parallèlisme adaptatif dans un environnement multi-utilisateurs hétérogène», Actes de l'école thématique CNRS placement dynamique et répartition de charge, Presqu'île de Giens, pp. 237-243, Juillet 1996.

[Namyst & al. 95] R. Namyst & J.F. Méhaut, «PM^2: Parallel Multithreaded Machine : a Multithread Environment on Top of PVM», Proceedings of the 2nd EuroPVM, Lyon, France, September 1995.

MIA

MIA *(Multivendor Integration Architecture)* [MIA 92] a été mis en chantier par NTT *(Nippon Telegraph and Telephone corporation)*. C'est une architecture conçue pour rendre interopérable des systèmes basés sur un modèle client-serveur, en spécifiant un ensemble d'interfaces de services. Le but final étant de pouvoir vendre des systèmes clefs en main construits avec des composants en provenance de constructeurs différents.

Mise en oeuvre de l'architecture

Quatre interfaces ont été définies :

1) l'interface de programmation d'application qui permet la portabilité d'applications écrites dans différents langages (C, Fortran, Cobol);
2) l'interface d'interconnexion de systèmes définit des protocoles de communication et s'appuie à la fois sur les protocoles de l'Internet et de l'OSI.
3) l'interface homme-machine qui caractérise les formats d'affichage et les opérations sur les stations clientes.
4) l'interface d'échange d'information inter-environnement qui facilite le transfert d'informations entre l'environnement de développement et celui d'exécution. Cette interface définit aussi les tables de transcodage de caractères permettant l'échange des données et des programmes. Trois types d'information peuvent être échangés : le code source des programmes, les définitions des bases de données (avec SQL en utilisant le langage de description de données DDL) et la définition des écrans.

Référence

[MIA 92] «Multivendor Integration Architecture : Concept and Design Philosophy», White Paper, Nippon Telegraph and Telecom Corporation, Tokyo, 1992.

ROSA

ROSA *(RACE Open Service Architecture)* [Rosa 92a & 92b] est une architecture orientée objet pour les services de communications à hauts débits *(Integrated Broadband Communication Services)*. Ses objectifs sont de proposer un ensemble de concepts, de règles et de recettes pour la spécification, la planification et l'implémentation de services ouverts.

Le modèle d'architecture

L'intégration se fait au niveau services et doit être indépendante des technologies de réseaux.

Services	
SSF	Sémantiques des services Exigences sur l'infrastructure
RSF	Mécanismes supportant l'ajout de services aux ressources de télécommunication
	Ressources de télécommunication

Figure 106 : Les squelettes d'implantations de RSA

ROSA utilise deux squelettes d'implantation SSF et RSF

1) SSF (*Service Specification Framework*) pour la spécification de services;

   Si on utilise la terminologie ODP, SSF couvre le point de vue des traitements (*Computational Model*). Les concepts du SSF sont mieux décrits par les types d'objets suivants :

   - **contrôle de service** (*Service Control*) qui laisse un utilisateur rejoindre, suspendre, reprendre et quitter des activités dans un service donné et négocier les paramètres du dit service ;

- **sessions** (*Session*) qui laisse le service de contrôle d'objet ajouter, changer et supprimer les informations d'état de l'utilisateur ;
- **charge** (*Charge*) qui enregistre et manipule les informations de charges induites par la session utilisateur ;
- **contrôle de transport** (*Transport Control*) qui maintient des information d'états des connections de transport.

2) RSF (*Resource Specification Framework*) pour la spécification du système, à savoir entre autres :

- définir des abstractions pour les ressources de télécommunications ;
- définir les composants requis sur les systèmes cibles ;
- définir des règles pour l'extension de ses concepts.

Mise en oeuvre de l'architecture

SSF et RSF sont liés entre eux par des relations qui permettent ensuite de faire une correspondance entre les squelettes d'implantation (cf. Figure 106). Ainsi, les besoins implicites ou explicites de SSF sont incorporés dans l'infrastructure créée à travers les spécification de services.

Les nouveaux services sont spécifiés avec des composants SSF qui aident à s'abstraire des technologies sous jacentes. Les nouvelles infrastructures de réseaux sont spécifiées grâce à des composants RSF (et aux règles attachées) et donc n'influent pas sur les SSF.

Références bibliographiques

[Rosa 92a] RACE Open Services Architecture, «ROSA Handbook», Release Two, Technical Report D.WPQ..2 93/FTL/DNR/DS/A/013/b1, RACE Project, European Commission, Brussels, May 1992

[Rosa 92b] RACE Open Services Architecture, «ROSA Architecture:», Release Two, Technical Report D.WPY..2 93/BTL/DNR/DS/A/005/b1, RACE Project, European Commission, Brussels, May 1992

SOM

La solution d'IBM aux problèmes d'interopérabilité de classes d'objets écrites dans des langages différents est SOM [SOM 94a et 94b] (*System Object Model*) et sa version distribuée DSOM (*Distributed SOM*). L'interopérabilité se fait au niveau du code binaire résultant de la compilation de ces classes. Il en résulte que les objets sont utilisables quel que soit le langage de développement et modifiables sans recompilation complète du programme.

SOM une extension du modèle objet de l'OMG. On y retrouve donc la notion de classe, d'héritage (simple et multiple) et de polymorphisme (grâce à un mécanisme de liaison dynamique). Les interfaces de classes sont décrites en IDL Corba et les types de données de l'OMA sont supportés. La classe *SomObject* est la racine de toutes les classes de la hiérarchie. La gestion et la modification dynamique des classes se fait à l'aide de métaclasses. Ainsi, les descripteurs de classes SOM sont des objets instances d'une métaclasse SOM. Autrement dit, toute classe est décrite par la métaclasse dont elle est une instance.

SOM dispose de trois méthodes de recherche du code d'une méthode appelée (classée par ordre d'efficacité et de transparence décroissante) :

1) **la liaison par déplacement** : le compilateur remplace statiquement le nom de la méthode par un déplacement dans une table associée à l'objet.
2) **la liaison par nom** (*lookup*) : lors de l'appel, une recherche dynamique est lancée dans la hiérarchie des classes pour trouver le code de la méthode en fonction de

son nom.

3) **la liaison par répartition** (*dispatch*) : le programmeur fournit une fonction qui sait retrouver où se trouve la méthode.

Chaque service disponible dans SOM est un ensemble de classes dépendantes, appelé infrastructure (*Framework*). Il en existe cinq principales pour :

1) la gestion de la persistance (*Persistence Framework*) ;
2) la gestion de la réplication (*Replication Framework*) ;
3) la génération d'émetteurs (*Emitter Framework*) ;
4) la gestion de la répartition (*Distribution Framework*) aussi appelé DSOM ;
5) la gestion du référentiel d'interface (*Interface Repository Framework*).

SOM est disponible sur les systèmes suivants : OS/2, AIX, Windows 95, MacOS 7.

Références bibliographiques

[Campagnoni 95] M. Potel & J. Grimes, «IBM's System Object Model», Dr Dobb's Special Report On The Interoperable Object Revolution, Vol 19 (16), pp. 24-29, March 1995.

[Orfali & al. 96] R. Orfali, D. Harkey & J. Edwards, «The Essential Distributed Objects Survival Guide», Wiley Ed., pp. 343-352, 1996.

[SOM 94a] IBM Corp., «SOMobjects Users Guide», SC23-2680, IBM Ed. , 1994.

[SOM 94b] IBM Corp., «SOMobjects Program Reference», SC23-2681, IBM Ed. , 1994.

Stardust

Stardust [Cabillic & al. 96a et 96b] est un environnement d'exécution pour applications parallèles s'exécutant sur une architecture matérielle hybride composée de stations de travail et de machines parallèles à mémoire distribuée. L'architecture matérielle actuelle est composée d'une machine Intel Paragon de 56 processeurs (le système d'exploitation étant l'Unix Paragon/OSF1) et d'un réseau de PC Pentium (le système d'exploitation étant l'Unix chorus/MiX). Toutes les stations de travail sont connectées via un commutateur ATM à 155 Mbit/s. Le modèle de programmation est SPMD et les applications communiquent à la fois par envoi de messages et par mémoire partagée répartie.

En ce qui concerne la communication par mémoire partagée, les problèmes de représentation et de transfert des données entre machines hétérogènes ont été résolus par la création de la notion de page mémoire hétérogène. Pour les deux systèmes, les communications par mémoire virtuelle partagée sont réalisées par construction d'un serveur de pagination externe au noyau.

La cohérence des données partagées est alors gérée par un système à deux niveaux. Le premier niveau gère la cohérence des données entre des machines homogènes (*intra-architectrure*) grâce à un gestionnaire qui se trouve sur chaque machine. Au second niveau, au sein de chaque groupe de machines homogènes, la cohérence des données avec des machines hétérogènes est gérée par un gestionnaire de mémoire partagée hétérogène. Les pages hétérogènes sont échangées entre gestionnaire via le protocole de représentation des données XDR et le protocole de communication TCP/IP.

Stardust inclut un mécanisme de capture, à la demande, de l'état des applications. Ces demandes de création de points de reprise sont insérés **par le programmeur** dans la boucle principale du code de son application. Ces points de reprise sont éventuellement sauvegardés sur disque dans le but soit de gérer les fautes, soit de réaliser de la migration d'applications.

La migration ou la reprise d'une application, entraîne sa réexécution complète depuis le dernier point de reprise. En cas de placement, le choix des machines dépend à la fois de la priorité de l'application, de la charge des machines et des choix du programmeur. Ce dernier établit alors un fichier de configuration contenant une liste de machines (liste et

type) trié par ordre décroissant de préférence. La gestion des informations de charge est rudimentaire (machine chargée ou non chargée) et est centralisée par type d'architecture.

L'accès concurrent aux ressources par différentes applications est résolue par des priorités, associées à chaque application. Ces priorités sont utilisées pour ordonnancer les exécutions. En cas de surcharge d'une machine, les applications les moins prioritaires migrent ou sont suspendues momentanément.

Références bibliographiques

[Cabillic & al. 96a] G. Cabillic & I. Puaut, «Répartition de charge dans Stardust: un environnement pour l'exécution d'applications parallèles en milieu hétérogène», Actes de l'école thématique CNRS placement dynamique et répartition de charge, Presqu'île de Giens, pp. 167-174, Juillet 1996.

[Cabillic & al. 96b] G. Cabillic & I. Puaut, «Stardust: an environment for parallel programming on network of heterogeneous workstations», Technical Report No 1006, IRISA, Avril 1996.

Systèmes répartis et systèmes agents

AEGIS/EXOKERNEL

Aegis [Engler & al. 95a] est un système d'exploitation qui offre une gestion des ressources physiques au niveau de l'application. Son architecture, appelée exokernel, est basée sur un noyau minimal exportant de manière sécurisée toutes les ressources matérielles (à travers des interfaces de bas niveau) directement vers les applications. *Un exokernel rompt avec l'idée commune qu'un système d'exploitation doit offrir des abstractions sur lesquelles les applications sont construites (telles qu'un processus, un fichier, un espace d'adressage).* Ces abstractions définissent une machine virtuelle sur laquelle l'application s'exécute, restreignant par la même les types d'applications possibles. C'est pourquoi l'exokernel ne définit pas d'abstractions autres que celles nécessaires au matériel [Engler & al 95b]. Les services «standards» du système d'exploitation tels que la mémoire virtuelle et l'ordonnanceur sont implémentés comme des librairies s'exécutant dans l'espace d'adressage de l'application. Le code des services systèmes s'exécutant dans les librairies peut être modifié par l'application, suivant ses besoins, pour implémenter des objets systèmes et les politiques d'accès associées.

Références bibliographiques

[Engler & al. 95a] D.R. Engler, M.F. Kaashoek & J.O'Toole, «Exokernel: An Operating System Architecture for Application-Level Ressource Management», Proceedings of the Fifteenth ACM Symposium On Operating Systems Principles, December 1995.

[Engler & al. 95b] D.R. Engler & M.F. Kaashoek, «Exterminate All Operating System Abstractions», Proceedings of the Fith Hot topics in Operating Systems, May 1995, pp. 78-83.

[Kaashoek & al. 97] M.F. Kaashoek, D.R. Engler, & al. «Application Performance and flexibility on exokernel systems», Proceedings of the 16th ACM Symposium on Operating Systems principles, Saint Malo, France, 1997.

Aglets Workbench

IBM Aglets Workbench est un environnement de programmation d'agents mobiles totalement écrit en Java (pour offrir une portabilité maximale). Un aglet est un agent mobile (ou agile) écrit en Java. L'utilisation de la classe Aglet permet d'hériter des propriétés et des fonctions nécessaires à un agent mobile tels que :

- un système de nommage unique pour chaque agent ;
- un itinéraire de voyage pour indiquer les destinations multiples où l'agent doit se rendre et la gestion automatique en cas d'impossibilité de mouvement ;
- un mécanisme de tableau noir permettant à plusieurs agents de collaborer et de partager des informations de manière asynchrone ;
- un schéma de communication par passage de messages qui supporte les communications asynchrones et synchrones ;
- un chargeur de classes d'agents qui permet le mouvement à travers le réseau à la fois du code (Bytecode Java), de l'état de l'agent et du contexte d'exécution. Ce contexte d'exécution étant indépendant de la machine hôte.

Le paradigme des Aglets unifie la gestion des objets fixes et mobiles, des objets passifs et autonomes et des objets locaux et distants. Il supporte les opérations en mode connecté et déconnecté.

Les aglets disposent d'un système de sécurité en couches. La première couche est offerte par Java et ses mécanismes de sécurité. La couche suivante est constituée du gestionnaire de sécurité qui permet aux utilisateurs d'implémenter leurs propres mécanismes de pro-

tection. La troisième et dernière couche est composée des API de sécurité Java permettant au programmeur d'inclure des fonctionnalités de sécurité dans leur agent.

Aglet Workbench inclut les services et les outils suivants :

- **un protocole de transfert d'agent** (*ATP*). ATP est utilisé, comme son nom l'indique, pour transférer, de manière uniforme et quelle que soit la plate-forme, des agents à travers le réseau. C'est un protocole standard de niveau applicatif, adapté à la communication sur Internet. ATP gère ainsi les URL (*Universal Resource Locator*) pour localiser les ressources et les agents. ATP supporte le transfert d'agents écrit dans des langages différents et fournit un mécanisme de transport et d'interrogations d'agents uniforme à travers le réseau. La première version d'ATP, nommée ATP/0.1, est implémentée de manière indépendante à la plate-forme des Aglets, tout en y étant intégré. Totalement écrit en Java, il fournit des classes de gestion des démons ATP et de génération de requêtes et de réponses vers des sites ATP.
- **des gabarits de conception.** Ils sont disponibles pour le programmeur, pour lui faciliter la mise en oeuvre de canevas d'interactions, tels que le canevas maître-esclave, client-serveur et notifieur/notifié. Ces gabarits sont disponibles sous la forme de classes réutilisables.
- **Tahiti : un gestionnaire d'agents visuel**. Tahiti utilise une interface graphique unique pour suivre et contrôler l'exécution des aglets. Il est possible en utilisant le glisser-déposer de faire communiquer deux agents ou de les faire migrer vers un site particulier. Tahiti dispose d'un gestionnaire de sécurité paramétrable qui détecte toute opération non autorisée et empêche l'agent de la réaliser.
- **Fiji - un lanceur d'agent sur le Web**. Fiji est un applet Java capable de créer ou de détruire un aglet sur un navigateur Web. Fiji utilise comme unique paramètre l'URL de l'agent, qui est intégré directement dans le code HTML d'une page Web. Comme pour un applet, l'exécution d'un aglet commencera par le téléchargement du code, puis par son lancement grâce à Fiji. Si le serveur Web est complété par un démon ATP, il devient très facile de répartir des aglets sur les sites Web.

Références bibliographiques

Site Internet http://www.trl.ibm.co.jp/aglets/ et aglets@yamato.ibm.co.jp

ATLAS

ATLAS [ATLAS 91 & Schuelke 91] est une architecture orientée-objet pour des systèmes d'exploitations répartis. Son objectif central était de cacher la complexité de l'architecture en fournissant un système réparti ouvert, robuste et administrable. Mis en oeuvre par Unix International, un consortium à but non lucratif, ATLAS :

1) offre une architecture basée sur l'intégration des technologies et l'interopérabilité des différents systèmes existant de l'industrie ;
2) met en oeuvre cette architecture au moindre coût possible ;
3) donne à l'administrateur et à l'utilisateur la vision d'un système unique.

Le modèle d'architecture

Cette architecture s'appuie sur la notion d'objet. Chaque objet accède à une ressource par l'intermédiaire d'un agent qui lui indique sa localisation.

L'architecture proposée est découpée en cinq couches distinctes (cf. Figure 107) :

1) **la couche de gestion et d'intégration des services de base.** On y regroupe le noyau des systèmes UNIX, les différentes commandes, les entrées-sorties, les types de fichiers (etc.) ;

2) **la couche des services de communications.** Elle gére les protocoles de communication de l'Internet, les couches de communication OSI, les multiples types de RPC et les services tolérants aux fautes ;

3) **la couche de services distribués**. On y trouve les services de gestion des objets, du temps, de la sécurité, du nommage et de l'administration ;

4) **la couche des services d'application.** Elle offre la gestion des fichiers, des types de données, des transactions, de la messagerie et de l'interface utilisateur ;

5) **la couche des outils d'application.** Elle se trouve au niveau le plus haut de l'architecture et permet la spécification, le développement, le test et la documentation des applications distribuées.

Mise en oeuvre de l'architecture

La mise en concordance de cette architecture en couches avec les systèmes propriétaires monolithiques *(legacy system)* est difficile à mettre en oeuvre. Elle passe par la création au sein de l'architecture d'un composant unique pour la gestion et l'interopérabilité des systèmes propriétaires entre eux et avec les systèmes répartis. L'intégration est réalisée avec le protocole LU 6.2, des logiciels d'émulation de terminaux et des services transactionnels puissants.

ATLAS intègre les produits Open Network Computing de Sun Microsystem et de l'OSF/DCE au sein d'ONP (*Open Network Platform*), fournie dans Unix System V Release 4.

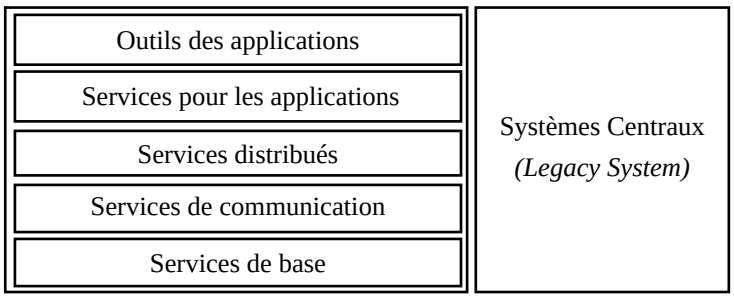

Figure 107 : L'architecture fonctionnelle de UI-ATLAS

Références bibliographiques

[ATLAS 91] «Unix Consortium Building Distributed Computing Standards», Database Reviews (USA), Vol. 3 (5), Octobre 1991, pp. 4-7.

[Schuelke 91] A. Schuelke, «Unix International's Atlas Distributed Computing Framework», EurOpen Unix Conference, May 1991.

CHOICES

Choices [Campbell & al. 92 et 93] est le premier exemple d'application de la programmation par objets à la conception d'un système d'exploitation avec utilisation d'une méthodologie de conception. Choices est un système réparti orienté objet, complètement écrit en C++, qui dispose de plus de 400 classes et 150 000 lignes de code source. Les objets sont utilisés pour modéliser à la fois l'interface matérielle, l'interface de l'application et toutes les ressources systèmes. L'interface d'application orientée objet est définie par l'invocation de méthodes à partir d'objets situés dans l'espace utilisateur vers des objets situés dans l'espace du noyau. Dans l'espace utilisateur, un objet du noyau est représenté par un proxy appelé *ObjectProxy.* Choices supporte plusieurs systèmes de gestion de fichiers (formats et interfaces) et plusieurs systèmes de communications (par messages, par mémoire virtuelle partagée et répartie).

L'architecture de Choices utilise des squelettes d'applications (*Framework*) qui décrivent les composants objets du système et la façon dont ils interagissent. Dans Choices, le comportement du système est défini par des classes. Des squelettes d'applications généraux ont été créés et sont ensuite, par raffinements successifs, spécialisés en *sous- squelettes* d'applications pour finalement obtenir un prototype exécutable. Les squelettes d'application sont plus intéressants que les traditionnels modèles en couches, car ils permettent par héritage et polymorphisme de réutiliser des composants systèmes par instanciation, sans avoir à les modifier en fonction de telle ou telle architecture matérielle cible.

Choices fonctionne sur une machine virtuelle Unix et a été porté sur des plate-formes SUN, PC et Encore Multimax NS32332. Un exécutable réparti est réalisable à tout moment par l'utilisation d'un outil de prototypage, nommé Virtual Choices. Cet outil de prototypage génère du code pour des plate-formes matérielles données, par encapsulation du code spécifique pour chaque processeur dans des classes. Ainsi, les appels systèmes Unix sont-ils traités par des machines virtuelles équivalentes, mais sur des machines physiques hétérogènes.

Références bibliographiques

[Campbell & al. 92] R. Campbell, N. Islam & P. Madany, «Choices, framework and refinment», Computer Systems, Vol. 5 (3), 1992, pp. 217-257.

[Campbell & al. 93] R. H. Campbell & N. Islam, «CHOICES: a parallel Object Oriented Operating Systems», Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, A. Yonezawa Ed., The MIT Press, 1993, Chapter 13, pp. 393-451.

Exokernel

voir AEGIS

Oasis

Oasis [Cheong 93] est un système réparti basé sur le modèle agent-objet. Dans ce système, les processus sont des entités distribuées et autonomes respectant les propriétés d'un agent définit par [Agha 86]. Les agents sont implémentés sous la forme d'un ou plusieurs processus légers qui peuvent être dans trois états : libre (*free*), prêt (*ready*) et en attente (*wait*). Un processus léger est libre, lorsqu'il n'est assigné à aucun appel de méthode et est utilisé pour former un groupe dynamique (*pool*) de processus. Ainsi, lors d'une invocation de méthode, la création d'un nouveau processus léger n'est réalisée que s'il n'y a pas de processus léger libre. Les processus légers dans l'état prêt sont ceux qui sont en attente d'exécution. L'exécution des processus prêts est pseudo parallèle et l'ordonnancement est réalisé non pas par le système mais par l'agent. La gestion de la file d'attente des processus prêt est non préemptive, c'est donc à chaque processus d'indiquer qu'il a fini de s'exécuter. Un processus en mode attente est bloqué jusqu'à la réception d'un évènement. Oasis gère la hiérarchie et définit un Meta agent comme un regroupement d'agent au sein d'un agent de niveau supérieur.

Oasis gère la migration d'objets par passage de références entre espaces mémoire dans un environnement hétérogène. Un agent peut migrer sur n'importe quelle machine. Pour se faire, le système Oasis compile les applications dans un langage intermédiaire permettant une cross-compilation lorsque l'objet arrive sur une machine hétérogène par rapport à celle d'origine.

L'environnement hétérogène est découpé en grappes nominatives redéfinissables par l'utilisateur. Le système Oasis est une surcouche au système. Un processus démon doit donc s'exécuter sur chaque machine de la grappe. La communication est basée sur des RPC asynchrones, ce qui permet à un agent de créer plusieurs activités.

Références bibliographiques

[Agha 86] G. Agha, «An overview of Actor languages», Sigplan Notices, Vol. 21 (10), October 1986, pp. 58-67.

[Cheong 93] F.-C. Cheong, «OASIS: An Agent-Oriented Programming language for Heterogeneous Distributed Environment», Phd-Thesis, University of Michigan, 1993.

Odyssey

L'environnement Telescript a été porté en Java dans Odyssey [GenMagic 97]. Odyssey permet de définir des agents logiciels mobiles dans un environnement organisé en places. Ainsi un agent mobile se déplace de places en places. Les places et les agents sont des classes Java et sont implémentés par des processus légers. Comme Java ne permet pas de capturer l'état d'exécution d'une processus léger, chaque fois qu'un agent est transporté d'une place vers une autre, le processus léger est relancé.

Un travailleur (*Worker*) est structuré comme un ensemble de tâches et de destinations. A chaque tâche est associée une destination. Ainsi, arrivée à sa destination une tâche s'exécute jusqu'à la fin, puis le travailleur passe à la tâche suivante sur sa liste.

Odyssey fonctionne sur toutes les plate-formes supportant le JDK 1.1 de Sun, sans modification de la machine virtuelle. Odyssey utilise la méthode d'invocation distante (RMI) de Java et utilise la librairie graphique AWT et les entrées sorties du JDK 1.1. Des travaux sont en cours pour intégrer les protocoles IIOP de Corba et DCOM de Microsoft.

Certaines fonctionnalités de Telescript n'ont pas été implémentées à cause des limitations du langage Java. Ainsi, la fonction GO de Telescript a disparu, du fait qu'en Java, il n'est pas possible d'enregistrer l'état d'exécution de l'agent avant sa migration.

Référence

[GenMagic 97] http://www.genmagic.com/agents

Spin

SPIN [Bershad & al. 95] est un système d'exploitation fonctionnant sur Dec Alpha, qui peut être dynamiquement et de manière sécurisée, spécialisé pour atteindre les performances et les fonctionnalités requises par des applications. Programmé en Modula-3 [Nelson 91], SPIN implémente les processus légers, la mémoire virtuelle et la gestion des périphériques systèmes. Des services d'extensions sont utilisés pour que les services spécifiques d'une application puissent être intégrés dans le système.

Ainsi, les applications définissent des extensions en Modula-3, qui sont dynamiquement liées à l'espace d'adressage virtuel du noyau du système. Les appels systèmes spécifiques de l'application sont alors réalisés avec une latence faible, ce qui permet des interactions à faible grain entre le système et l'application. Cette intégration est réalisée grâce à deux outils clefs : le linker dynamique SPIN [Sirer & al. 96] et le routeur dynamique d'évènements. Le premier résoud tous les symboles non définis et les inclus dans un domaine de protection logique, accessible sans risque pour le reste du système. Le second fait remonter à l'application les évènements qui lui sont spécifiques, après que cette dernière se soit faite enregistrée et ait indiquée les évènements qui la concernaient.

Plexus [Fiuczynski & al. 96] est une implémentation intéressante au dessus de SPIN, d'un système qui laisse les applications définir ou redéfinir elles mêmes les protocoles de communications optimisées dont elles ont besoin.

Références bibliographiques

[Bershad & al. 95] B.N. Bershad, S. Savage, P. Pardyak, E.M. Sirer, M.E. Fiuczynski & al., «Extensibility, Safety and Performance in the SPIN Operating System», Proceedings of the Fifteenth ACM Symposium on Operating System Principles, December 1995.

[Fiuczynski & al. 96] M.E. Fiuczynski & B.N Bershad, «An Extensible Protocol Architecture for Application-Specific Networking», Proceedings of Usenix Winter, 1996.

[Sirer & al. 96] E.G. Sirer, S. Savage, P. Pardyak & B.N. Bershad, «Safe Dynamic Linking in Extensible operating System», First Workshop on Compiler Support for System Software, February 1996.

Spring

Spring [Hamilton 93a] est un système distribué supportant une plate-forme pour des applications constituées d'un ensemble d'objets. Le système est organisé autour d'un micro-noyau (cf. Figure 108) structuré pour gérer des invocations rapides d'objets entre espaces d'adressage et utilise une mémoire répartie [Khalidi & al. 93]. Ce micro-noyau gère des domaines contenant des objets actifs et des portes comme mécanismes de communication entre objets. L'implémentation des applications se fait dans un langage quelconque et les interfaces des objets sont décrits avec un langage d'interface (IDL). L'héritage des interfaces est possible, mais pas l'héritage des implémentations.

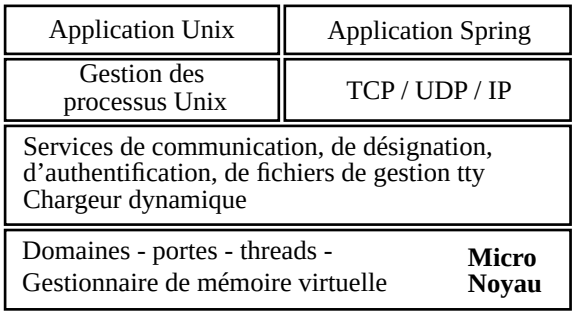

Figure 108 : Architecture fonctionnelle de Spring

La distribution est complètement transparente et les communications sont optimisées en fonction de la situation (on peut alors passer de RPC à des RPC légers moins coûteux). Dans Spring, le sous contrat [Hamilton 93b] défini comme une classe, précise la sémantique de l'appel de méthode. Ainsi, différentes implémentations d'un même type peuvent avoir des sous contrats différents. Le sous contrat est très proche de l'objet protocole d'ODP. Supposons qu'un client communique avec un serveur qui est répliqué N fois, le sous contrat servira à envoyer pour une requête client, N messages aux serveurs répliqués (de manière transparente). L'invocation d'un objet à distance se fait à l'aide d'un proxy qui connecte les noyaux des différentes machines de manière transparente.

Références bibliographiques

[Hamilton & al. 93a] G. Hamilton & P. Kougiouris, «The Spring Nucleus: a Microkernel for Objects», In Proc. of the Summer Usenix Conference, pp. 147-160, June 1993.

[Hamilton & al. 93b] G. Hamilton, J. Mitchell & M. Powell, «Subcontract: A Flexible Base For Distributed Programming», In Proc. of 14th Symposium on Operating Systems Principle, December 1993.

[Khalidi & al. 93] Y. A. Khalidi & M.N. Nelson, «The Spring Virtual Memory System», Sun Microsystem s laboratories, Technical report SMLI-93-9, March 1993.

SR

Un programme SR [Andrews & al. 88 et 93, Atkins & al. 88] est constitué d'une ou plusieurs ressources. Chaque ressource contient deux parties, une partie spécification qui définit l'interface et une partie corps qui contient le code décrivant l'objet abstrait. Dans la partie spécification, il est possible d'importer ou d'exporter des ressources et de déclarer l'interface des opérations invocables par une autre ressource.

L'instanciation dynamique d'une ressource se fait avec la fonction *create*, qui nécessite un paramètre de localisation sur une machine virtuelle. Une machine virtuelle procure un espace d'adressage unique sur une machine. Si aucune machine virtuelle n'est spécifiée, alors les ressources sont créés par défaut sur la machine locale.

Les processus ne peuvent communiquer qu'à travers des opérations. Il existe deux types d'opérations : *proc* et *in*. Une opération *proc* ressemble à une procédure, car elle possède un nom et peut passer et recevoir des paramètres. Une opération de type *in* ressemble à la clause *accept* en ADA et est bloquante. Une opération parmi toutes celles possibles est alors choisie de manière indéterministe.

Une opération appelée par un *call* est synchrone, c'est à dire que l'appel se termine lorsque le résultat est retourné. De l'autre coté, la primitive d'envoi *send* se termine quand le processus cible a été créé ou que l'opération demandée a été mémorisée en attente d'être traitée (opération semi-synchrone). Les combinaisons d'un type d'invocations (*call* ou *send*) avec un type de service (*proc* ou *in*) changent le type d'appel (cf. Tableau 48).

Tableau 48: Les primitives d'envoi de messages dans SR

Invocation	Service	Résultat
call	proc	Appel de procédure (à distance ou locale)
call	in	Rendez-vous
send	proc	Migration dynamique de processus
send	in	Passage asynchrone de messages

SR a été implémenté au dessus d'Unix BSD 4.x. Le lieur (*linker*) agglomère les ressources compilées au sein d'un module de chargement, associé à un processus Unix communiquant par *sockets*. Au sein de ce dernier, le parallélisme est simulé par le noyau d'exécution de SR.Références bibliographiques

[Andrews & al. 88] G.R. Andrews & al., «An overview of the SR language and implementation», ACM transaction on Programming Language and Systems, Vol. 10 (51), Jan 1988.

[Andrews & al. 93] G.R. Andrews & R. Olsson, «The SR Programming Language: Concurrency In Practice», Benjamin / Cummings Eds, 345 pages, 1993.

[Atkins & al. 88] M.S. Atkins & R.A. Olson, «Performance of Multi-tasking and Synchronization in the Programming Language SR», Software Practice and Experience, Vol. 18 (9), September 1988, pp. 879-895.

Taligent

Taligent est une société fondée en 1992 par Apple, Hewlett-Packard et IBM. Taligent a mis au point un environnement de développement, de maintenance et de réutilisation de logiciels orienté objet, portable et de nouvelle génération. Le concept clé introduit, comme dans Choices, est le Framework [Taligent 93]. Un framework est une collection d'objets qui fournissent un service ajustable par le programmeur. Deux interfaces sont disponibles pour l'accès aux framework sous forme d'API. La première, appelée API client, indique uniquement les interfaces d'appels des services et présente les frameworks comme des boites noires. La seconde, appelée API des framework, est destinée aux utilisateur désirant personnaliser les frameworks disponibles.

L'environnement Taligent est divisé en trois parties [Potel & al. 95] :

1) TalAE *(Taligent Application Environment)* est l'implémentation du modèle orienté-objet de programmation des applications. TalAE contient plus de 100 frameworks couvrant les domaines du multimedia, du graphisme, de la programmation distribuée, du réseau informatique, de l'interface utilisateur et de l'accès aux bases de données. TalAE est distribué et portable sur différents systèmes d'exploi-

tation.

2) TalDE *(Taligent Development Environment)* est un ensemble d'outils de développement orienté framework complétant TalAE. La programmation se fait alors en C ou en C++.

3) TalOS *(Taligent Object Services)* a été conçue pour être la plate-forme système orienté-objet native pour TalAE. TalOS est basée sur le micro-noyau Mach v 3.0, étendu pour la gestion des objets. TalOS fournit des framework systèmes pour la création de gestionnaires de bas niveau (entrées-sorties, fichiers, systèmes, réseaux et communications).

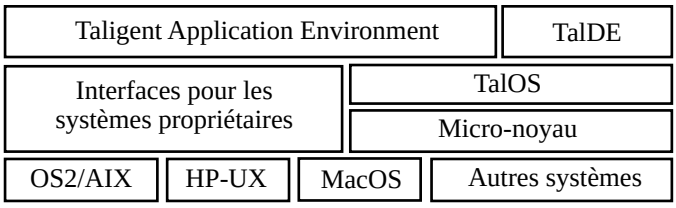

Figure 109 : L'architecture fonctionnelle du système Taligent

TalAE a été porté sur diverses plate-formes matérielles (Intel, PA-Risc, Power-PC), en attendant que TalOS deviennent la couche système commune à toutes ces architectures (cf. Figure 109). La migration se fera donc en douceur. Enfin, Taligent est un environnement ouvert conforme aux spécifications CORBA et a été proposé pour adoption à l'X-Open.

Références bibliographiques

[Potel & al. 95] M. Potel & S. Cotter, «Inside Taligent Technology », Addison-Wesley Ed., 1995.

[Taligent 93] Taligent white papers, «Leveraging Object-Oriented Frameworks», «Building Object-Oriented Frameworks», site: http://www.taligent.com.

Site Internet http://www.taligent.com

Telescript

Le produit de General Magic est le plus ancien et le plus connu du marché des langages de génération de code mobile. Il est composé de Telescript [White 95], un langage de programmation et de Magic Cap, une interface utilisateur.

Telescript est un langage orienté objet pour la création d'applications distribuées sur un réseau informatique. Chaque machine sur le réseau doit disposer d'une serveur Telescript (appelé *Engine*) capable d'interpréter le langage. La machine virtuelle dispose de la gestion de processus légers.

Les concepts de base de ce langage sont la place et l'agent. Avec Telescript, un agent est représenté par du code mobile à durée de vie limité capable de se transporter lui même entre deux localisations différentes (appelées places). La place accueille les agents, vérifie leurs droits d'accès avant de les activer et offre des services prédéfinis à l'agent.

Deux instructions du langage caractérisent les propriétés de l'agent :

- *la mobilité* avec la commande **GO**. L'agent qui fait appel à cette méthode suspend son activité et la reprend une fois qu'il a atteint sa nouvelle place.
- *la coopération* avec la commande **MEET**. Les agents peuvent alors se rencontrer et éventuellement collaborer à un but commun.

Telescript est fortement orienté vers le monde des télécommunications et dispose d'une gestion de la sécurité évoluée [Magic 96].

Références bibliographiques

[Magic 96]	General Magic, «An Introduction To Safety And Security In Telescript», White Paper, February 1996,
[White 95]	J. E. White, «Mobile Agents», General Magic White Paper, October 1995.
Site Internet	http://www.genmagic.com

Time Warp

Il existe plusieurs implémentations au dessus du système Time Warp, celle ci s'attache à faire du placement de processus sur une architecture de stations de travail multi-utilisateurs en réseaux. Ce qui est original dans cette approche, c'est l'utilisation des horloges logiques pour faire de l'équilibrage de charge. Une phase statique permet de lancer les processus et ensuite la répartition de charge se fait dynamiquement. La décision est centralisée sur une machine qui reçoit périodiquement toutes les horloges des processus sur chaque station. La migration de processus est offerte et le système tourne sur station Unix.

Référence

[Burdof 93]	C. Burdof, J. Marti, "Load Balancing Strategies for Time Warp on Multi-User Workstations", The computer Journal, Vol. 36 (2), 1993, pp. 168-176.

Voyager

Voyager est un environnement de création d'agents mobiles complètement écrit en Java qui a été mis au point par la société ObjectSpace. Le composant de base de Voyager est l'objet. Les objets résident et s'exécutent dans des applications Voyager. Chaque application est responsable de l'infrastructure fournie aux objets pour les communications distantes, le mouvement et d'autres services de base. Au démarrage d'une application Voyager, des processus légers sont créés pour gérer les services de gestion du temps (*Timing Services*), le ramasse miettes réparti (*garbage collector*) et la gestion du trafic TCP/IP. En général, les services d'une application Voyager sont lancés lors de la création d'un objet.

Un objet distant est un objet qui existe en dehors de l'espace d'adressage de l'application Voyager. Une application communique avec un objet distant en construisant une version virtuelle de l'objet localement. La notion d'objet mandataire est alors appelé objet virtuel. Il est possible de déplacer des objets d'une application Voyager vers une autre, en envoyant un message *move()* à un objet via son interface virtuelle (l'adresse de l'application destination doit être comprise dans le message). L'objet attend alors de traiter tous les messages en attente et se déplace à son nouveau lieu de résidence, en laissant un secrétaire sur place. Le rôle du secrétaire est de lui renvoyer les messages susceptibles d'arriver après son départ. Voyager offre aussi la gestion des agents, qui ne sont en fait que des objets mobiles et autonomes.

Référence

Site Internet	http://www.objectspace.com/voyager/