Des vertus de la schizophrénie pour le prototypage d'applications à composants interopérables

Laurent Pautet* — **Fabrice Kordon** **

* *École Nationale Supérieure des Télécommunications*
46, rue Barrault
F-75013 Paris

** *LIP6, Université Pierre & Marie Curie,*
4, place Jussieu
F-75252 Paris cedex 05

RÉSUMÉ. *La schizophrénie dans le domaine des intergiciels vise à résoudre le problème d'interopérabilité entre modèles de répartition, plus connu sous l'acronyme M2M pour « Middleware To Middleware ». Si un intergiciel générique instancie une personnalité pour un modèle, un intergiciel schizophrène instancie simultanément plusieurs personnalités cohabitantes et interagissantes. Cette approche permet de construire des passerelles dynamiques entre intergiciels hétérogènes, solution souple au problème du M2M. L'architecture qui en découle facilite également le prototypage et le développement rapide d'applications réparties à composants. Notre plate-forme schizophrène PolyORB s'articule autour de composants adaptables fondés sur des gabarits de conception classiques ou spécifiques.*

ABSTRACT. *Schizophrenia in the middleware context intends to solve the distribution model interoperability issue (M2M or « Middleware To Middleware »). When a generic middleware instantiates a personality for a distribution model, a schizophrenic middleware includes several coexisting and collaborating personalities. This enables the construction of dynamic gateways between middleware in order to provide a flexible solution to the M2M problem. The resulting architecture also facilitates the rapid prototyping of component based distributed applications. Our schizophrenic platform PolyORB is composed of adaptive components based on standard or specific design patterns.*

MOTS-CLÉS : *intergiciels, prototypage, interopérabilité, généricité, configurabilité, composants.*

KEYWORDS: *middleware, prototyping, interoperability, genericity, configurability, components.*

RSTI - TSI. Volume 23 - n° 10/2004, pages 1301 à 1328

1. Introduction

Le développement intense de la répartition dans des domaines applicatifs aussi variés que le temps réel ou l'internet amplifie la définition de nouveaux modèles de répartition ou encore l'adaptation de modèles existants. Ces modèles s'articulent autour de combinaisons de plusieurs mécanismes accompagnés d'éventuelles extensions :

– l'envoi de messages constitue une réponse efficace aux problèmes de calcul scientifique (MPI, [FOR 95]) ou de systèmes d'information (MQSeries, [IBM 97]) ;

– l'appel de sous-programmes distants (RPC, [BIR 84]) ou de méthodes sur objets répartis (CORBA, [OMG 02]) fournissent des solutions plus lourdes mais plus structurantes du point de vue du génie logiciel ;

– les objets partagés ou localisés dans un espace de stockage commun éventuellement réparti (*distributed shared memory* ou DSM) se sont répandus à travers des environnements comme ThreadMarks [KEL 94].

En fonction des domaines et des besoins des applications, les modèles de répartition se spécialisent ou se généralisent autour de ces mécanismes fondamentaux. L'utilisateur peut vouloir évaluer un mécanisme spécifique de traitement de requête, comme celui de l'asynchronisme dans CORBA grâce à *CORBA Messaging* [OMG 98], ou encore intégrer dans un modèle de répartition une politique présente dans un autre modèle. Pour illustrer ce dernier cas, citons l'utilisation d'un *Portable Object Adapter* [OMG 02] au sein d'un intergiciel orienté messages (*Message Oriented Middleware* ou MOM) [BAN 99]. La multiplication des modèles de répartition rend alors nécessaire le développement d'intergiciels adaptés. Ce phénomène induit deux problématiques autour du développement et de la réutilisation de composants logiciels.

Le développement de nouveaux intergiciels permet de disposer rapidement d'une infrastructure d'exécution supportant un nouveau modèle de répartition. Lorsque le besoin d'expérimentation apparaît, notamment pour valider le modèle sous-jacent, on peut parler de conception et de développement par prototypage : le prototype révèle les forces et les faiblesses du modèle. Une nouvelle version est alors développée sur la base de cette expérimentation. Cependant, prototyper ou développer rapidement et à faible coût des plates-formes répondant aux besoins d'une application ou d'un modèle de répartition constitue une barrière technologique pour l'industrie de l'intergiciel.

Les logiciels répartis deviennent de plus en plus complexes et nécessitent la réutilisation de nombreux composants préexistants. Or, ces derniers sont parfois fondés sur des modèles de répartition hétérogènes ce qui, paradoxalement, compromet l'interopérabilité recherchée[1] ; le développement de passerelles statiques introduit un coût de développement important et donne lieu à des mises en œuvre peu efficaces. Dès lors, le développement rapide de passerelles dynamiques entre composants constitue un point crucial mais requiert des techniques de réalisation assistées bien particulières.

1. Le terme d'interopérabilité concerne ici l'interopérabilité entre modèles de répartition.

Pour permettre le développement rapide des intergiciels ainsi que celui de passerelles entre composants logiciels hétérogènes, les intergiciels disposent de différentes propriétés : la configurabilité, la généricité et l'interopérabilité.

La configurabilité permet d'adapter les plates-formes de communication aux besoins réels de l'application. Une approche classique consiste à définir une architecture dont les composants faiblement couplés sont configurables indépendamment. Ainsi, l'utilisateur ajoute ou retire des propriétés de l'intergiciel concerné en sélectionnant statiquement ou dynamiquement les composants appropriés.

La généricité traite de la configurabilité de l'intergiciel en fonction du modèle de répartition. Développer des plates-formes répondant efficacement à la multiplicité des modèles de répartition constitue une barrière technologique pour l'industrie de l'intergiciel. Une approche fréquemment adoptée vise à factoriser les composants pour les réutiliser, les surcharger et les intégrer dans une architecture générique instanciée ou personnalisée en fonction du modèle de répartition.

L'interopérabilité entre modèles de répartition constitue une problématique industrielle émergente, résumée par le concept de M2M pour *Middleware To Middleware* [BAK 01]. Elle est rendue essentielle par la réutilisation de composants préexistants. Le paradoxe de l'intergiciel provient de la contradiction entre son objectif d'interopérabilité au sens classique et l'hétérogénéité inhérente aux multiples modèles de répartition utilisés. Or, l'approche consistant à traduire statiquement une entité d'un modèle de répartition dans un autre ne supporte pas le passage à l'échelle.

L'objectif de la *schizophrénie* vise à unifier ces trois approches au sein d'un même intergiciel. Elle constitue ainsi une réponse aux deux problèmes que nous avons identifiés : le développement rapide d'intergiciels et le prototypage de composants applicatifs fondés sur des modèles de répartition nouveaux et hétérogènes.

La section 2 aborde, d'une part, l'adaptabilité des intergiciels existants, d'autre part, le prototypage d'applications à composants hétérogènes. La section 3 définit la schizophrénie des intergiciels comme une unification de la configurabilité, de la généricité et de l'interopérabilité. Elle constitue ainsi une solution globale aux problèmes d'industrie des intergiciels et d'interopérabilité des modèles de répartition. La section 4 décrit l'architecture et les composants d'un intergiciel schizophrène. Nous exposons en section 5 comment un tel intergiciel facilite le prototypage d'applications à composants hétérogènes. La section 6 présente une partie des résultats obtenus par notre intergiciel schizophrène PolyORB alors que la section 7 expose le développement rapide d'un nouvel intergiciel et le prototypage d'une application témoin.

2. Techniques propres à l'industrie de l'intergiciel et au prototypage d'applications à composants interopérables

Nous proposons d'étudier les approches généralement adoptées pour développer rapidement ou prototyper des applications à composants interopérables. Ces approches

relèvent de deux domaines que sont l'industrie de l'intergiciel et le développement rapide. Nous montrons tout d'abord les limites des approches retenues dans l'industrie des intergiciels pour enrichir ou adapter les plates-formes aux besoins d'une application. Parmi ces approches, nous présentons la configurabilité, la généricité et l'interopérabilité. Par la suite, nous abordons les stratégies adoptées pour satisfaire les besoins en prototypage et en développement rapide d'applications réparties. Enfin, nous présentons une application témoin à composants interopérables que nous exploiterons pour démontrer la pertinence de notre solution.

2.1. *Industrie de l'intergiciel*

Différentes solutions ont été proposées pour adapter, enrichir et faire interopérer les intergiciels. Celles-ci se fondent sur des techniques de configurabilité, de généricité et d'interopérabilité. Nous illustrons les principes d'implémentation choisis par quelques intergiciels reconnus. Nous concluons en soulignant les limites de ces approches et en mettant en évidence les éléments pertinents à retenir de ces techniques.

2.1.1. *Intergiciels fondés sur la configurabilité*

La configurabilité permet de répondre précisément aux besoins réels d'une application ou encore d'adapter un modèle de répartition à des variantes mineures. L'utilisateur sélectionne les composants utilisés dans un intergiciel afin de déterminer par exemple les politiques d'allocation de mémoire. Parmi les travaux examinés dans [PAU 01a], nous nous concentrons sur le logiciel libre GLADE, la première implémentation de l'annexe des systèmes répartis d'Ada (DSA) [ISO 95]. Cet intergiciel se caractérise par une configurabilité proche de celle de TAO [SCH 97].

GLADE est la première implémentation de DSA qui consiste en un sous-ensemble d'entités classiques d'Ada dotées de propriétés de répartition grâce à des directives de compilation. Celles-ci permettent à des sous-programmes et des objets du langage de se comporter comme des sous-programmes distants, des objets répartis et des objets partagés. Bien que dotée de fonctionnalités plus étendues, on peut considérer DSA comme étant à Ada ce que RMI est à Java.

GLADE se compose de GNAT, compilateur Ada de la famille de GCC, qui a été enrichi pour les besoins de la répartition ; de GARLIC, système de communication dont la norme impose l'interface ; et de GNATDIST, outil de déploiement et de configuration. Porté sur de nombreux systèmes d'exploitation, le logiciel libre GLADE a été validé auprès des organismes accrédités faisant de GNAT le seul compilateur proposant l'intégralité de la norme.

Grâce à son langage de description, GNATDIST [KER 96] permet de déployer une application en affectant ses modules à des nœuds logiques. Il permet surtout dans notre cas de configurer les composants du système de communication GARLIC et d'apporter des variantes mineures au modèle de répartition DSA [PAU 00].

GNATDIST permet de contrôler la configuration des ressources de GARLIC notamment les protocoles de communication, les supports de stockages partagés ou les processus légers traitant les requêtes distantes. Notamment, l'utilisateur peut configurer le système de communication dans une version allégée en supprimant la gestion concurrente des requêtes ; ce mécanisme est essentiel pour les systèmes embarqués.

2.1.2. *Intergiciels fondés sur la généricité*

La généricité étend le concept de configurabilité par la production d'une personnalité ou d'une instance de l'intergiciel en fonction d'un modèle de répartition. Cette approche permet de développer rapidement un environnement pour démontrer la pertinence d'un nouveau modèle de répartition. Parmi les intergiciels génériques, QuarterWare propose une architecture novatrice orientée autour des gabarits de conception [SIN 98]. Dans cette section, nous préférons présenter Jonathan, un intergiciel libre fondé sur les liaisons inspirées du modèle ODP [ODP 95].

Jonathan (http://www.objectweb.org/jonathan) propose un intergiciel générique bâti sur une architecture classique mais enrichie de composants internes originaux. Le subrogé (*Surrogate*) [DUM 98], constitue l'un d'entre eux. Il fournit l'implémentation des liaisons explicites ou implicites et ne se limite pas à un contrôle d'accès au serveur comme le fait un mandataire (*proxy*). La notion d'adaptateur d'objets se trouve étendue par rapport à celle de CORBA. Ce mécanisme ne concerne plus uniquement le serveur mais aussi le client, de sorte que le contrôle sur la liaison s'exerce de bout en bout. Le gabarit de conception *Fabrique Abstraite* utilisée pour les liaisons rend Jonathan configurable en l'occurrence en fonction des protocoles de communication.

`David` constitue une personnalité (ou instanciation) de Jonathan pour CORBA. Il offre les fonctionnalités de DII et de DSI et importe le Portable Object Adapter (POA) de JacORB [BRO 97]. `Jérémie` fournit également une personnalité pour RMI. Bien que les modèles de répartition de ces personnalités soient fort proches, le code provenant de la partie générique s'avère relativement réduit (moins de 10 %) comparé au code nécessaire à l'instanciation d'une personnalité. L'explication en est que la partie générique se compose essentiellement d'interfaces abstraites, ce qui induit la concrétisation de nombreux composants pour instancier une personnalité.

2.1.3. *Intergiciels fondés sur l'interopérabilité*

L'interopérabilité vise à produire des passerelles entre modèles de répartition, ce qui s'avère un processus délicat en l'absence de fonctionnalités adaptées au sein des intergiciels impliqués dans ce travail de traduction. Nous avons clairement mesuré cette difficulté lors du développement de CIAO [QUI 99], un générateur opérationnel de passerelles statiques entre DSA/GLADE et CORBA/AdaBroker (notre premier environnement CORBA libre pour Ada [AZA 99]). Dans cette section, nous nous concentrons sur CorbaWeb qui à la différence de CIAO produit des passerelles dynamiques unidirectionnelles du web vers CORBA.

CorbaWeb (http//corbaweb.lifl.fr/CorbaScript) propose des outils pour concevoir, déployer et utiliser des objets CORBA *via* le web. L'utilisateur accède aux services CORBA par l'intermédiaire d'un navigateur. Il les consulte comme des documents HTML et exécute des opérations sur ces objets à travers des scripts CGI. CorbaWeb se compose de métascripts développés en CorbaScript [MER 97]. Ils associent, à chaque objet CORBA, des interfaces HTML pour le représenter, ainsi que des scripts, pour appeler ses méthodes.

CorbaWeb [GEI 97] relie un client web à des objets CORBA sur leur serveur. Il effectue la liaison entre les deux environnements en recevant des requêtes HTTP pour les transformer et les exécuter dynamiquement sous forme de requêtes CORBA. Ces opérations se fondent sur le mécanisme d'invocation dynamique (DII) et sur le référentiel d'interfaces (IR). S'il n'adresse pas directement le problème de l'interopérabilité entre modèles de répartition, CorbaWeb propose une architecture novatrice de passerelle dynamique qui démontre la pertinence de la DII et de l'IR en la matière.

2.1.4. *Limites et analyses des intergiciels*

Aucun des intergiciels que nous avons étudiés ne propose simultanément des propriétés de configurabilité, de généricité et d'interopérabilité. Les gabarits de conception constituent un élément architectural commun car ils facilitent la mise en œuvre de la configurabilité (GLADE, TAO) et de la généricité (Jonathan, QuarterWare).

L'étude d'intergiciels génériques comme Jonathan (ou QuarterWare) et le développement de nombreux environnements répartis comme GLADE (ou AdaBroker) nous ont démontré qu'une plate-forme générique constitue une solution efficace au problème de l'industrie des intergiciels. Cependant, comme nous l'avons souligné, les intergiciels considérés partagent peu de code entre personnalités.

Si les systèmes d'exploitation s'articulent autour de modules désormais bien identifiés comme les gestionnaires de processus et de mémoire, aucune architecture classique accompagnée de ses composants fondamentaux n'apparaît clairement dans le domaine des intergiciels. Définir une telle architecture constitue également un prérequis à l'obtention d'un taux important de factorisation de code.

Nous avons retenu de CorbaWeb que le mécanisme d'activation de passerelles dynamiques fondé sur l'invocation et l'implémentation dynamiques d'interfaces DII et DSI peut s'appliquer au contexte de l'interopérabilité des modèles de répartition. Or, une telle production de passerelles dynamiques nécessite la mise en présence de plusieurs intergiciels sur un même nœud avec une inévitable inflation de ressources.

Notre première approche a consisté à fusionner le code de plusieurs personnalités d'un même intergiciel pour favoriser la production de passerelles dynamiques et limiter l'inflation de ressources due à la présence de multiples intergiciels. Or, les différentes personnalités des intergiciels génériques actuels cohabitent difficilement, ceux-ci étant structurellement conçus pour ne supporter qu'une seule instance à la fois. De plus, les personnalités concrétisent souvent l'architecture abstraite d'un intergiciel générique aux composants incompatibles entre instanciations.

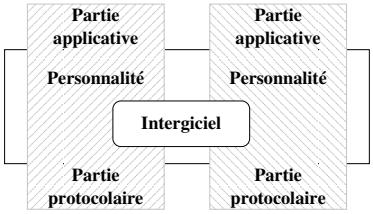

Figure 1. *Couplage fort*

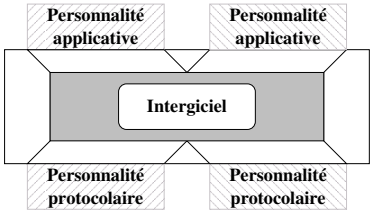

Figure 2. *Couplage faible*

Pour pallier ces limitations, notre seconde approche a consisté à rompre avec le caractère monolithique des intergiciels actuels et notamment avec le couplage fort des parties applicative et protocolaire illustré par la figure 1. Définir une couche neutre, indépendante des modèles de répartition, peut rendre effective la cohabitation et l'interaction de personnalités multiples au sein d'un même intergiciel comme l'illustre la figure 2. Ce couplage faible entre les interfaces applicatives présentées aux composants logiciels et les interfaces protocolaires présentées aux autres intergiciels peut s'obtenir en reprenant les mécanismes de DII, DSI et IR présentés précédemment.

2.2. *Prototypage et développement de logiciels complexes*

Le développement et la maintenance d'applications industrielles deviennent sans cesse plus délicats. Les systèmes concernés sont de plus en plus complexes, les technologies évoluent rapidement, la répartition devient plus courante et les intervalles entre la production et la commercialisation s'avèrent de plus en plus courts (*time to market*) [LEV 97]. On parle depuis maintenant presque 10 ans de « crise chronique du logiciel » [GIB 94].

2.2.1. *Développement d'applications réparties*

Définir des méthodes de génie logiciel outillées par des ateliers de génie logiciel constitue la première parade au problème du développement. Ces méthodes reposent sur des modèles décrits à l'aide de notations dédiées. À ce titre, UML [OMG 03] s'est imposé comme un standard de représentation dans l'industrie.

Il est reconnu depuis longtemps que, même menée scrupuleusement, l'implémentation d'un système n'est pas en général une image exacte de sa spécification. L'une des raisons majeures provient des choix de réalisation effectués par les équipes de développement qui contredisent souvent les hypothèses implicites de la phase de conception [MUR 91]. Ce problème est crucial dans le cas d'applications réparties à composants. La conception au moyen d'un modèle (objet) de haut niveau est nécessaire mais ne permet pas de décrire de manière suffisamment précise les besoins propres aux systèmes répartis [MED 00]. Des langages de description d'architecture (ADL) viennent

souvent compléter les spécifications UML afin d'en faciliter l'implémentation, notamment sur les aspects de communication et de déploiement [EGY 99].

Pour avoir une meilleure correspondance entre modèle et implémentation, une solution consiste à utiliser un modèle de répartition de haut niveau (par exemple, le modèle d'objets répartis) et à s'appuyer sur des intergiciels qui facilitent l'implémentation en adaptant le modèle de développement à une architecture répartie. Dans certains cas, des outils de compilation supportent l'écriture d'une partie de l'application (les souches et les squelettes). CORBA [OMG 02] peut être vu comme un précurseur dans ce domaine. Mais sa démarche trop explicite oblige le développeur à considérer des détails qui méritent d'être masqués. Bien qu'offrant une vision centrée sur un langage de programmation, l'annexe des systèmes répartis d'Ada (DSA) constitue un autre exemple qui toutefois masque mieux les détails de réalisations que CORBA.

L'objectif initial d'un intergiciel est d'unifier et de faciliter le développement audessus d'un modèle de répartition abstrait le plus généraliste possible. Cependant, le prix à payer est une couche logicielle complexe et coûteuse en ressources et en temps d'exécution. Comme les applications n'utilisent souvent qu'un sous-ensemble des fonctions offertes, les développeurs envisagent d'élaborer des versions légères adaptées aux besoins précis de l'application, ce qui est en contradiction avec les objectifs initiaux. C'est le « paradoxe de l'intergiciel » introduit dans [PAU 01a].

2.2.2. *Prototypage d'applications réparties*

L'IEEE [KUR 93] définit le prototypage comme une approche « privilégiant le développement de prototypes dès les premières étapes du cycle de vie du logiciel afin d'obtenir des réactions et des analyses utiles pour la suite du processus de développement » [2]. Il apporte une solution au problème de la dérive entre spécification et produit en proposant de relier étroitement ces notions.

Cette définition est diversement interprétée tout au long du cycle de vie du logiciel [KOR 95]. Il devient maquettage (prototype jetable) ou génération de programmes (le générateur étant vu comme un moyen de lier, par un processus de production déterministe, une spécification et son implémentation). Il est aussi vu comme une approche de conception/développement/maintenance (RAD-Cycle, [KER 95]). L'idée consiste à regrouper les techniques à base de maquettage et celles à base de génération de programmes dans un modèle de développement [KOR 02] basé sur des raffinements successifs d'un prototype qui tend à devenir la version finale du système.

Des mesures démontrent qu'une démarche de prototypage intégrée dans un processus de développement, comme souvent dans le domaine du matériel, réduisent considérablement les phases de test [SHT 98]. Pour les systèmes répartis, des expériences ont été menées avec succès dans des projets comme Proteus [GOL 96] (calcul massivement parallèle) ou Trapper [SCH 95] (systèmes répartis embarqués). Les applica-

2. La citation originale est « A type of development in which emphasis is placed on developping prototypes early in the development process to permit early feedback and analysis in support of the development process ».

tions réparties produites automatiquement grâce à chacun de ces projets peuvent être modifiées et reconfigurées au niveau du modèle. Les prototypes produits à l'aide de l'environnement Proteus ont constitué les versions finales de ces applications.

2.3. *Prototypage d'une application témoin répartie à composants interopérables*

Nous considérons l'application issue de [QUI 03] pour illustrer les problèmes posés par une démarche de développement par prototypage. Cette application, dont l'architecture est décrite en figure 3 permet d'échanger et de distribuer des messages courts entre plusieurs utilisateurs. Elle comporte deux types d'objets :

– un *tableau d'affichage* sur lequel chaque utilisateur peut déposer un message public destiné à l'ensemble des utilisateurs ;

– des *correspondants* représentant les utilisateurs. Ces derniers émettent des messages (à destination d'un ou de tout utilisateur) et en reçoivent. Certains correspondants peuvent être préexistants.

Considérons que l'on associe des correspondants réalisés avec Ada/DSA à des correspondants s'exécutant sur CORBA. On se heurte à un problème d'interopérabilité : les liaisons entre les différentes configurations des correspondants et le tableau d'affichage doivent être préservées. Le développement d'une passerelle *ad hoc* permet de résoudre ce problème mais est incompatible avec une démarche de prototypage. Son coût de développement est trop élevé, d'autant plus que la passerelle peut être remplacée lors d'une prochaine itération (pour obtenir une nouvelle version du prototype).

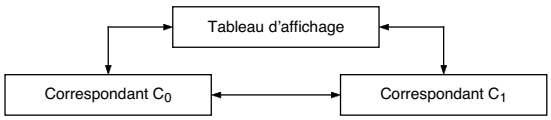

Figure 3. *Architecture d'une application à composants*

De même, le déploiement de certains composants de cette application sur des nœuds possédant des contraintes particulières (consommation mémoire ou stratégie de parallélisme contrainte) ou le besoin d'expérimenter de nouveaux modèles de répartition sont un frein à l'étude d'un prototype tôt dans le cycle du développement. Les coûts de développement d'exécutifs dédiés pour ces composants sont trop importants pour s'intégrer dans une démarche de prototypage.

3. Définition, architecture et propriétés de la schizophrénie

Dans [PAU 01a], nous définissons la schizophrénie comme caractérisant chez un intergiciel sa capacité à disposer, simultanément, de plusieurs personnalités afin de les faire interagir efficacement. Un tel intergiciel dispose des caractéristiques suivantes :

– cohabitation de personnalités multiples,

– partage du code et des données entre les personnalités,

– découplage des personnalités applicatives et protocolaires.

3.1. *Architecture*

Obtenir ces caractéristiques nécessite la définition d'une architecture spécifique bâtie autour d'une couche indépendante des personnalités comme l'illustre la figure 4. Cette architecture ainsi que ses composants fondamentaux constituent les éléments cruciaux conduisant à un fort taux de réutilisation de code. Si l'architecture s'adapte mal à la construction d'une personnalité, l'essentiel de son code s'avérera spécifique.

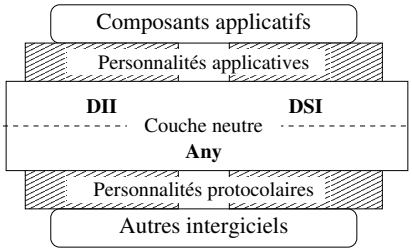

Figure 4. *Architecture schizophrène*

Au même titre que les systèmes d'exploitation sont bâtis autour de composants désormais bien identifiés (processeur, mémoire, réseau, etc.), nous proposons d'organiser un intergiciel autour de sept composants fondamentaux que nous présenterons dans la section 4. Défini indépendamment des modèles de répartition, chacun de ces composants fournit des éléments de code réutilisables d'une personnalité à l'autre que nous nommons *couche neutre*.

Notons que plus les modules sont éloignés des personnalités applicative et protocolaire, moins ils sont influencés par les modèles de répartition et plus ils contribuent à la couche neutre. Celle-ci occupe ainsi une place centrale dans l'architecture de l'intergiciel schizophrène. Le principe de la couche neutre s'inspire de celui du langage intermédiaire qui sépare le dorsal et le frontal d'un compilateur.

Globalement, la couche neutre s'articule autour de structures particulières inspirées du type autodescriptif Any de CORBA qui stocke une valeur de type quelconque et une description de ce dernier. De même, des structures proches de l'invocation et de l'implémentation dynamiques de CORBA permettent à la couche neutre d'interagir indépendamment avec les parties applicative et protocolaire. Ces types de données sont adaptés afin de supporter des personnalités différentes de celle de CORBA.

3.2. *Propriétés*

L'intergiciel schizophrène contribue à l'industrie de l'intergiciel. Il propose une architecture et des composants simplifiant le travail de prototypage et de développement de tout intergiciel. Plus sa couche neutre s'enrichit de composants concrets, plus le code se trouve factorisé entre personnalités. Un tel intergiciel favorise aussi le déploiement des personnalités applicative et protocolaire. En effet, en l'absence de son équivalent protocolaire, développer une personnalité applicative peut s'effectuer grâce à l'instanciation protocolaire d'un autre modèle de répartition.

L'intergiciel schizophrène contribue à l'interopérabilité entre modèles de répartition en facilitant la mise en œuvre de passerelles dynamiques entre composants hétérogènes. En effet, l'existence de personnalités protocolaires et applicatives découplées par l'intermédiaire de la couche neutre introduit un élément d'architecture essentiel pour la réalisation de passerelles dynamiques. Plus les personnalités applicatives et protocolaires utilisent la représentation commune de la couche neutre, plus les conversions entre modèles sont facilitées. Plus la couche neutre s'enrichit de composants concrets, plus l'interaction entre personnalités se trouve favorisée.

4. Composants fondamentaux d'un intergiciel schizophrène

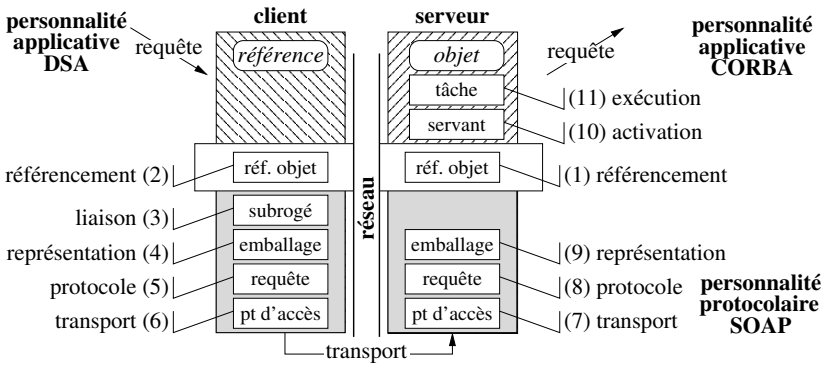

Figure 5. *Traitement d'une requête entre intergiciels schizophrènes*

Afin de mettre en œuvre un intergiciel schizophrène, nous avons formalisé son architecture autour de sept composants, chacun d'entre eux pouvant être indépendamment adaptés ou étendus selon le modèle de répartition [QUI 01]. Les composants des couches applicative (*Activation*, *Exécution*), neutre (*Référencement*) ou protocolaire (*Transport*, *Liaison*, *Représentation*, *Protocole*) se distinguent des composants externes à l'intergiciel (*Nommage*) par leur caractère fondamental dans la mise en œuvre de la plate-forme. Nous donnons dans la suite les définitions, les propriétés ainsi que les objectifs de ces composants fondamentaux. La figure 5 permettra d'illus-

trer leur articulation au travers du traitement d'une requête d'un client DSA vers un serveur CORBA par l'intermédiaire du protocole SOAP.

4.1. *Référencement*

Chaque objet se voit attribuer une référence composée d'informations transmises auprès de nœuds logiques. Elle désigne sans ambiguïté l'objet au cours de son existence. La référence d'un objet hébergé par l'intergiciel consiste en l'association :

– d'un identifiant local de l'objet ;

– de l'adresse d'un point d'accès du service de transport.

Côté serveur (étape 1), la couche applicative, responsable de la gestion de l'espace d'adressage local, fournit l'identifiant local de l'objet. Une personnalité protocolaire fournit l'adresse d'un point de transport ; celle-ci s'accompagne de la désignation du protocole à utiliser sur ce point d'accès. La réunion de l'identifiant local et de l'information de transport nécessite d'avoir visibilité sur les couches applicative et protocolaire. Elle doit donc être effectuée par un module situé dans la couche neutre.

Cette structure de données est utilisée comme une capsule opaque par la couche applicative du côté client (étape 2) : celle-ci ne prend pas connaissance des détails de sa structure interne et du contenu des profils. Elle n'utilise les références d'objets que pour les communiquer à d'autres nœuds en les passant en paramètres d'appels distants ou pour demander à l'intergiciel l'établissement d'une liaison avec un objet.

4.2. *Liaison*

Étant donné une référence désignant un objet, la fonction de liaison a pour rôle de créer une structure qui représente l'objet désigné et la concrétisation de la liaison avec lui. Cet objet de liaison correspondant à un objet distant comporte :

– **côté client :** un point de terminaison du service de transport et une pile de protocole ;

– **côté serveur :** également un point de terminaison du service de transport et une pile de protocoles, créés lors d'une connexion par un point d'accès.

Un objet de liaison permet de traiter des appels de méthode sur l'objet distant désigné, en les lui transmettant sous forme de messages. Cette structure constitue donc un *subrogé* de l'objet réel (étape 3). Comme celui-ci, ce subrogé est susceptible d'être désigné par une référence, moyennant l'attribution d'un identifiant par la fonction de référencement de l'intergiciel client. Une telle référence permet l'accès à l'objet initialement désigné, le nœud hébergeant le subrogé jouant le rôle de passerelle. Ces passerelles contribuent considérablement à l'interopérabilité entre modèles de répartition dans le cadre de la schizophrénie.

Le mécanisme d'établissement d'une liaison peut être :

– **implicite** lorsque l'établissement d'une connexion de transport et son association avec une référence sont effectuées de façon transparente pour l'utilisateur, au cours d'un appel de méthode (c'est le cas par défaut dans CORBA) ;

– **explicite** lorsque l'utilisateur souhaite contrôler certains aspects de l'établissement de la connexion, par exemple pour choisir des paramètres de qualité de service à négocier avec le réseau.

Lorsqu'une liaison est demandée pour une référence désignant un objet local, c'est le servant réalisant cet objet qui est retourné par la fonction de liaison (et non un subrogé de l'objet). Dans ce cas, la détermination du servant à partir de la référence est réalisée par un appel à la fonction d'activation, à travers la couche neutre.

4.3. *Représentation*

Les données reçues ou émises par un objet applicatif sont définies par leur type, au sein d'un modèle de types fixé par la personnalité applicative. Les modèles de types sont construits autour de types élémentaires (nombres, caractères) et de types composés formés par agrégation de types élémentaires ou d'autres types composés (tableaux, enregistrements). La couche neutre est elle-même munie d'un modèle de types. Chaque personnalité applicative définit une projection de son modèle de types vers celui de la couche neutre afin de construire les requêtes confiées à l'intergiciel.

La fonction de représentation intervient pour la transmission des requêtes. Elle convertit une donnée D d'un type T du modèle de la couche neutre en un message conforme à la représentation imposée par le protocole de communication utilisé. La fonction de représentation définit, pour tout type du modèle de la couche neutre, une transformation injective projetant toute valeur du type sur une valeur d'un type du modèle de la personnalité protocolaire (étape 4).

$$< D >_T \xmapsto{Representation} msg$$

Cette transformation peut s'effectuer avec une perte d'information : le message résultant msg doit contenir suffisamment d'information pour reconstituer la valeur D connaissant T et msg, mais pas nécessairement pour reconstituer $< D >_T$ en connaissant seulement msg. Ainsi, dans la représentation CDR de CORBA, une suite de quatre octets peut correspondre à un entier long ou bien à deux entiers courts.

La transformation réciproque de la fonction de représentation doit donc prendre non seulement msg mais aussi T en entrée. En l'absence de T, elle ne pourrait pas « magiquement » reconstituer cette information perdue avant transmission. La personnalité applicative doit fournir *a priori* à la fonction de représentation le type des

données qu'il souhaite recevoir (étape 9). Si $< \cdot >_T$ représente un conteneur vide possédant l'information de type T, nous définissons la transformation réciproque comme :

$$(msg, < \cdot >_T) \xmapsto{Representation^{-1}} < D >_T$$

4.4. *Protocole*

La fonction de protocole d'invocation de méthodes est complémentaire de la fonction de représentation : elle transforme la description des interactions entre les objets applicatifs d'une représentation locale définie par la couche neutre vers une forme transmissible au sein d'un message, et réciproquement (étapes 5 et 8).

Cette fonction orchestre aussi le déroulement d'un appel de méthode : à partir d'une demande d'invocation, un message est préparé puis émis. L'intergiciel client est mis en attente d'une réponse ; lorsque celle-ci est reçue, elle est retraduite sous forme neutre et signalée à la couche neutre, qui la transmet à la personnalité applicative.

Sur un serveur, à la réception d'une demande d'invocation, l'identité de l'objet et de la méthode concernés est extraite, ainsi que les paramètres de l'appel. Une requête sous forme neutre est construite et confiée à la couche neutre, afin que celle-ci la traite grâce à un servant local ou la retransmette vers un objet de liaison lorsqu'une passerelle est établie. Une fois la requête traitée, la couche neutre signale à la couche protocolaire que la réponse éventuelle peut être retournée à l'intergiciel client.

La fonction de protocole se charge donc de la transformation d'une requête en message et de la transformation réciproque. Dans ce dernier cas, l'adapteur d'objet lui fournit les informations nécessaires au déballage du message et notamment le conteneur typé $< \cdot >_T$ attendu par la fonction de représentation.

4.5. *Transport*

La fonction de transport consiste à transférer une information d'un point à un autre et offre deux abstractions principales : les points d'accès du service de transport et les points de terminaison du service de transport. Les points d'accès du service de transport représentent les entités créées par le système d'exploitation à la demande de l'intergiciel afin de recevoir des connexions de la part de correspondants distants à travers un réseau de communication. Les points de terminaison du service de transport représentent les entités du système au moyen desquelles les données sont échangées. La connexion établie, les deux intergiciels peuvent dialoguer par échange de données au moyen des points de terminaison ainsi créés (étapes 6 et 7).

Points d'accès et points de terminaison du service de transport représentent des entités permettant à un intergiciel d'interagir avec le monde extérieur. Les instants où des événements se produisent sur ces entités et requièrent l'attention de l'intergiciel

ne sont pas connus *a priori*. Ces objets constituent donc des sources d'événements asynchrones qui doivent être régulièrement scrutées par l'intergiciel afin que ces événements externes soient pris en compte.

4.6. *Activation*

A la réception d'une requête par la couche protocolaire, l'intergiciel grâce à un adaptateur d'objets associe l'objet logique désigné par la référence sur laquelle l'appel est effectué à un objet concret local (servant) chargée d'exécuter la méthode (étape 10).

Ce servant peut être un objet déjà existant, qui a été préalablement enregistré de façon explicite auprès de l'intergiciel. Dans ce cas, celui-ci maintient une table associant les objets actifs à leurs identificateurs.

Il est également possible que le servant soit incarné par une structure créée à la volée par une usine d'objets, et qui pourra, ou non, persister pour recevoir des appels ultérieurs. La référence peut également être utilisée comme une clé pour rechercher un objet existant parmi une collection, suivant un critère quelconque.

Enfin, un servant peut servir de servant « par défaut » pour recevoir les appels de méthodes pour les références ne correspondant à aucun objet explicitement enregistré.

4.7. *Exécution*

Un appel de méthode correspondant à une requête reçue doit être affecté à une tâche, et orienté vers le sous-programme applicatif approprié (étape 11). La tâche utilisée peut être soit une tâche prêtée temporairement à l'intergiciel par l'application, soit une tâche propre de l'intergiciel. Dans ce dernier cas, la tâche peut être banalisée, ou bien dédiée à l'exécution des requêtes liées à une entité.

L'ordonnanceur de la fonction d'exécution se charge d'affecter une tâche à une requête. Les différentes variations de son comportement se font par délégation d'une partie de ses fonctionnalités à des objets déterminant la politique de parallélisme de l'intergiciel. La détermination du sous-programme applicatif correspondant est réalisée sous le contrôle de la personnalité applicative, en fonction de l'interface que lui présentent les objets applicatifs.

5. Prototypage d'applications réparties fondé sur la schizophrénie

À l'heure où le développement et la maintenance d'applications deviennent de plus en plus délicats, le prototypage autrefois vu comme l'élaboration de maquettes sans lendemain, devient une approche de développement à part entière. L'apparition de la démarche *Model Driven Architecture* (ou MDA) [OMG 01] qui prône une nouvelle

manière de développer des applications réparties à partir d'un modèle de répartition indépendant de toute plate-forme d'exécution ne fait que renforcer cette tendance.

Ainsi, l'utilisation de technologies de conception et de développement rapides et malléables comme la schizophrénie constitue un atout important pour le génie logiciel des systèmes répartis. En particulier, ce concept nous permet d'instrumenter une démarche de développement de type *prototypage par raffinements* (*evolutionary prototyping*) [ASU 93].

5.1. *Prototypage par raffinements*

Le prototypage par raffinements répond à deux besoins [KOR 02] : non seulement les techniques de développement doivent être rapides mais elles doivent également aboutir à l'élaboration d'une partie du produit final ou de son intégralité. Aussi l'objectif consiste-t-il à assister la phase de développement d'une application à l'aide d'une panoplie d'outils permettant notamment de produire automatiquement les parties critiques du code. Parmi celles-ci, citons tout ce qui relève du contrôle et de l'adaptation des composants dans les applications réparties.

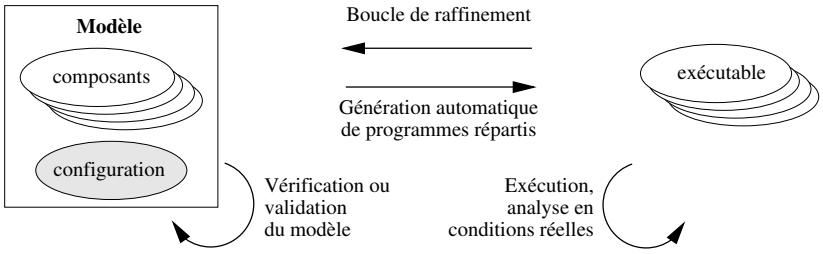

Figure 6. *Éléments d'une démarche de développement par prototypage*

La figure 6 illustre ce type de démarche. L'application est décrite au moyen d'un *modèle* qui sert de base à sa construction par génération automatique de programmes exécutables. Ce modèle identifie des composants (existants ou nouveaux), leurs interactions et les paramètres de configuration pour une architecture cible.

La génération de programmes peut être effectuée à différents niveaux :

1) sur l'application elle-même, par production, soit d'un squelette de contrôle garantissant les interactions entre composants, soit de l'application dans son intégralité (typiquement par insertion d'annotations de code séquentiel dans le squelette d'application ce qui se fait depuis longtemps dans les environnements HOOD) ;

2) sur l'exécutif dont le paramétrage permet, soit de faire coexister plusieurs modèles de répartition (en exploitant la généricité inhérente à la schizophrénie), soit d'optimiser l'exécution pour certaines cibles (en exploitant les possibilités de configuration offertes par la schizophrénie).

Ces deux niveaux sont complémentaires. Leur usage conjoint s'avère nécessaire lorsqu'un système intègre des composants préexistants issus de différents modèles de répartition (voir section 5.2). L'intérêt de cette approche, outre la rapidité de codage, réside dans la possible validation d'un modèle (formellement ou par simulation).

Une application exécutable dans son environnement cible peut ensuite être produite à faible coût. Cette application peut être évaluée afin d'optimiser certains critères (performance, empreinte mémoire, etc.), d'étudier l'impact des choix de déploiement ou d'analyser les stratégies du générateur de programmes. Les observations sur le prototype sont ensuite ré-intégrées sur le modèle au niveau de la description des composants, de la configuration ou plus simplement comme paramètres d'invocation des générateurs de programmes. Son aspect cyclique vaut à cette démarche l'appellation de « prototypage par raffinements ».

5.2. *Utilisation de la schizophrénie dans le prototypage*

Reprenons l'exemple présenté en section 2.3 en considérant que le comportement du tableau d'affichage et des correspondants soit décrit à l'aide d'un modèle exprimant la dynamique du comportement de manière non ambiguë.

Les outils de génération automatique de programme évoqués en figure 6 agissent sur cette spécification à la manière d'un compilateur et produisent une application répartie respectant les contraintes définies dans le modèle. Le tableau d'affichage et le correspondant seront automatiquement implémentés en deux composants distincts.

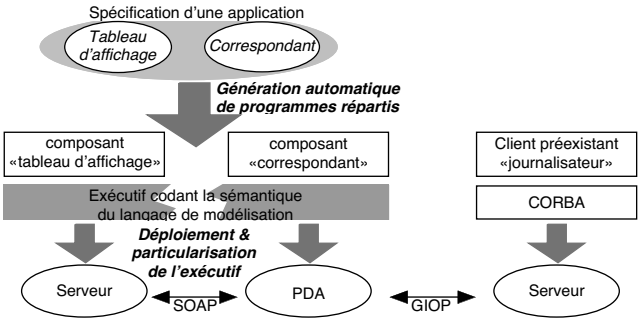

Figure 7. *Démarche de prototypage de l'application de la figure 3*

Comme un compilateur, la génération automatique de programmes utilise les primitives d'un exécutif réparti codant le modèle de répartition associé à la notation [GIL 02]. Mais, pour qu'un tel outil soit opérationnel, il faut implémenter l'exécutif associé au langage de modélisation. La schizophrénie nous permet de le faire, sous forme d'une personnalité applicative. Une telle opération s'effectue la première fois ;

on peut ensuite raffiner l'exécutif au fur et à mesure des évolutions apportées au langage de modélisation. Cette approche réduit avantageusement le coût de construction d'un générateur de programmes pour un langage de description donné (et donc pour un modèle de répartition donné).

Considérons maintenant que les différents éléments de cette application doivent être déployés sur différents types d'architecture. Par exemple, le correspondant s'exécute sur DSA dans un environnement embarqué tandis que le tableau d'affichage fonctionne sur un environnement CORBA. Il faut alors configurer les différentes instances de l'exécutif qui mettent en œuvre les services nécessaires à l'interaction entre les composants. Les propriétés de la schizophrénie facilitent cette phase de configuration.

– La personnalité applicative implémentant l'exécutif du langage de modélisation se décline en fonction des contraintes imposées par la cible sur laquelle réside chaque composant (empreinte mémoire, parallélisme, etc.).

– Pour assurer les communications entre les deux composants, il suffit de sélectionner une personnalité protocolaire (ici, SOAP).

Considérons maintenant que l'on souhaite ajouter un mécanisme de journalisation des actions du tableau d'affichage. Ce service est offert par un composant CORBA natif, s'exécutant sur une plate-forme développée « traditionnellement ». La réutilisation de ce composant est possible en reconfigurant l'exécutif du nœud qui exécute le tableau d'affichage. Ce dernier se comportera alors comme une passerelle. Cette reconfiguration ne nécessite aucun développement supplémentaire.

5.3. *Synthèse*

La notion de développement par prototypage (*model based development*) devient une alternative crédible au développement traditionnel, en particulier dans des domaines comme celui des applications à composants interopérables. Les propriétés de la schizophrénie facilitent la mise en œuvre de cette démarche :

– la *généricité* pour élaborer rapidement (à l'aide de personnalités) de nouveaux exécutifs spécialisés : cela permet d'envisager l'élaboration de nouveaux modèles de description adaptés à un domaine d'application et supportant un processus de génération de code (compilation de haut niveau),

– la *configuration* pour paramétrer finement un ou plusieurs exécutifs sur une architecture répartie ; cela est particulièrement utile lorsque ce dernier est partiellement embarqué,

– l'*intéropérabilité* : pour associer des modèles de répartition dans une même application : cela rend possible l'intégration à faible coût de composants préexistants.

6. Réalisations

PolyORB est un intergiciel schizophrène pour Ada réalisé sur la base de nos développements précédents, GLADE et AdaBroker, destinés à être remplacés par des personnalités de PolyORB. Nous leur avons emprunté des gabarits de conception et des éléments de configuration pour les étendre et les intégrer dans PolyORB.

Nous avons choisi Ada car ce langage intègre à la fois la programmation objet, des mécanismes sophistiqués de concurrence (tâches, types protégés) et la généricité. Une autre raison est l'existence d'un profil pour le temps réel (Ravenscar [DOB 98]). Nous pensons que ces choix ont grandement facilité le développement de PolyORB.

6.1. *Services de base*

PolyORB se compose de plusieurs services de base utilisés aussi bien dans la couche neutre que dans les personnalités protocolaires ou applicatives. Parmi ceux-ci, nous pouvons citer :

– une bibliothèque de parallélisme configurable (parallélisme complet, parallélisme profilé pour le temps réel [DOB 98], pas de parallélisme),

– des outils de génération de fonctions de hachages parfaites statiques [CZE 92] ou dynamiques [DIE 94],

– des gabarits de conception repris d'autres projets ou de notre propre confection.

Parmi les gabarits de conception utilisés, nous avons repris des gabarits comportementaux tels que *Wrapper Facade* ou *Acceptor - Connector* ainsi que des gabarits de concurrence *Half-async - Half-sync* ou *Reactor* [SCH 96]. Ces gabarits avaient déjà trouvé leur place dans des projets précédents comme GLADE et AdaBroker.

Nous avons introduit de nouveaux gabarits : *Component* et *Annotation* [QUI 03]. *Component* enrichit dynamiquement les méthodes offertes par un objet et rend explicite l'équivalence entre un appel de méthode et l'envoi d'un message [ING 78]. *Annotation* fournit une fonctionnalité équivalente afin d'enrichir dynamiquement un objet de nouveaux attributs ou données. Grâce à ces gabarits, nous définissons des relations entre entités sans imposer plus de connaissances sur celles-ci que nécessaires.

6.2. *Couche neutre*

Les composants des couches applicatives (*référencement*, *activation*, *exécution*) ou protocolaires (*référencement*, *transport*, *liaison*, *représentation*, *protocole*) s'appuient sur une couche neutre qui s'articule autour de divers éléments :

– un noyau neutre composé d'usines de références, d'usines de liaisons proches de celles proposées par Jonathan et du gestionnaire d'exécution et d'ordonnancement de requêtes (utilisé dans les composants de *référencement* et de *liaison*),

– un ensemble d'adaptateurs d'objets génériques inspirés de ceux de CORBA (utilisé dans les composants d'*activation* et d'*exécution*),

– une infrastructure de transport fondée sur des filtres inspirés de x-Kernel et de gabarits de conception comme ceux de TAO présentés précédemment (utilisée dans le composant de *transport*).

Nous avons réutilisé les aspects réflexifs des types `Request` et `ServerRequest` présents dans l'implémentation de DII et de DSI d'AdaBroker pour réaliser le gestionnaire d'exécution et d'ordonnancement de requêtes. Notons que nous avons adapté ces éléments afin qu'ils n'attirent pas la personnalité CORBA dans la couche neutre. Cette approche ainsi que l'utilisation de structures autodescriptives de type Any indiquée en 3.1 assure une forte implication de la couche neutre dans la réalisation d'une personnalité puisqu'elle n'impose pas plus de connaissances que nécessaires sur les entités qu'elle manipule.

6.3. *Personnalités applicatives*

CORBA : AdaBroker nous a facilité le développement de la personnalité CORBA. Celle-ci consiste en un compilateur d'IDL et un système de communication mettant en œuvre les mécanismes d'invocation et d'implémentation statiques et dynamiques. Hormis les réels à virgule fixe (pas encore implémentés), la personnalité applicative CORBA respecte la projection des constructions IDL vers le langage Ada. Le composant d'`Activation` consiste en un POA complet. Le composant d'`Exécution` reprend des travaux effectués dans GLADE et AdaBroker afin d'implanter les allocations de processus légers lors de requêtes distantes. Par ailleurs, la personnalité applicative comporte les services communs les plus courants comme le service de nommage ou d'événements ainsi qu'un référentiel d'interface.

DSA : la mise en œuvre de la personnalité DSA a été facilitée en raison du savoir-faire acquis lors du développement de GLADE et de l'étude d'interopérabilité faite autour de CIAO (voir section 2.1.3). Ce travail important a porté sur la réécriture du compilateur GNAT afin qu'il interagisse avec le système de communication de la personnalité DSA. Il faut noter que le compilateur met en œuvre des mécanismes plus riches et plus complexes que ceux de CORBA. Au travers de ce travail, nous nous sommes attachés à démontrer l'interopérabilité entre DSA et CORBA. À ce titre, le composant de *Liaison* a été enrichi afin de faciliter la mise en œuvre de passerelles dynamiques entre les modèles DSA et CORBA. Grâce à la suite de tests de l'*Ada Conformity Assessment Authority*, nous avons pu vérifier la conformité de notre mise en œuvre. Notons que les tests relatifs à la conformité des aspects désormais obsolètes de DSA ont été écartés de cette suite.

6.4. *Personnalités protocolaires*

GIOP : AdaBroker a facilité le développement de la personnalité protocolaire GIOP. Les composants de *Représentation* et de *Protocole*, inspirés d'AdaBroker, implantent les versions de GIOP allant de 1.0 à 1.2. Nous avons pu vérifier la bonne interopérabilité de ces composants lors de tests avec omniORB, Jonathan et OpenORB. Ceux-ci ont consisté en des appels de méthodes comportant des transferts de données des divers types prévus par le modèle de types d'OMG IDL.

7. Expérimentations et mesures

Nous avons utilisé PolyORB pour réaliser deux expériences de prototypage :

– la mise en œuvre de modèles de répartition n'étant pas fondée sur l'appel de méthodes à distance afin de valider pleinement notre architecture,

– l'exploitation des caractéristiques d'un intergiciel schizophrène pour le prototypage de l'application témoin présentée en section 2.3.

Nous avons évalué par des mesures les avantages de la schizophrénie.

7.1. *Prototypage de personnalités*

MOM : nous avons instancié PolyORB pour obtenir un MOM, nommé MOMA (*Message Oriented Middleware for Ada*). Nous avons adapté la spécification de JMS [SUN 99] pour Ada. La personnalité applicative consiste en un client réalisant cette spécification et un serveur de gestion des files de messages. La mise en œuvre d'un MOM aux fonctionnalités très distinctes de celles présentes dans un ORB n'a nécessité que peu d'ajouts au sein de la couche neutre. Le seul enrichissement significatif de PolyORB a porté sur l'asynchronisme dont le traitement initial comportait certaines lacunes. Cette étape nous a permis de constater que PolyORB fournit les composants essentiels pour la production de personnalités bien différentes de celles des ORB.

SOAP : la personnalité SOAP [W3C00] nous a conduit à développer le composant de *Protocole* pour HTTP et celui de *Représentation* pour XML. La fonction protocole pour SOAP réutilise et étend certains composants du projet AWS (*Ada Web Server*) [OBR 00]. Nous confirmons ainsi que le modèle de types et le formalisme de représentation des données que nous avons retenus pour la couche neutre (inspirés de CORBA) sont compatibles avec d'autres personnalités de répartition.

Lors de ces deux expérimentations, les fonctions offertes par la couche neutre ont permis la réalisation rapide des fonctions demandées. Ainsi, le temps de développement a été considérablement réduit par rapport à celui d'un intergiciel d'architecture classique. Un tel résultat démontre les perspectives intéressantes offertes par la schizophrénie en matière de prototypage de modèles de répartition.

Par ailleurs, le découplage entre les personnalités protocolaires et applicatives nous a permis de développer indépendamment les unes des autres. Notamment, la personnalité applicative fondée sur la spécification JMS a pu être élaborée en l'absence de la personnalité protocolaire SOAP. Cette propriété permet d'expérimenter des personnalités applicatives en réutilisant des protocoles déjà existants. Par exemple, les aspects protocolaires de GLADE n'ont pas été développés pour l'instant, GIOP et SOAP suffisant largement à assurer le transport des requêtes DSA.

7.2. *Prototypage d'applications à composants hétérogènes*

La problématique évoquée dans l'exemple de la section 5.2 nous permet de valider les fonctionnalités de PolyORB tout en démontrant que le prototypage d'une application répartie à composants hétérogènes est réalisable à très faible coût. Il s'agit d'ajouter à une application construite par génération de programme un composant préexistant : le journalisateur, posé sur une implémentation « classique » de CORBA.

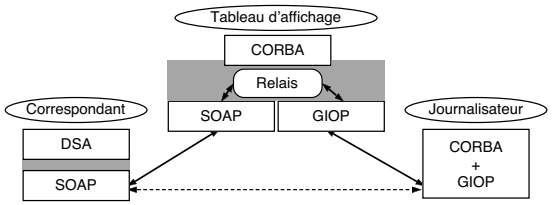

Figure 8. *Détail du prototypage de l'exemple de la figure 3*

La figure 8 illustre les différents composants de l'application enrichie. Pour faire interagir les composants originaux sachant que le tableau d'affichage s'exécute sous CORBA et que le correspondant est posé sous DSA, il faut configurer comme suit :

– *nœud du correspondant :* personnalité applicative DSA et personnalité protocolaire SOAP (nous ne disposons pas de personnalité protocolaire DSA),

– *nœud du tableau d'affichage :* personnalité applicative CORBA et personnalité protocolaire SOAP.

L'ajout du journalisateur entraîne la modification de configuration suivante :

– *nœud du journalisateur :* l'intergiciel monolithique CORBA,

– *nœud du tableau d'affichage :* ajout de la personnalité protocolaire GIOP pour assurer l'interopérabilité attendue par le composant réutilisé.

Cette modification de la configuration se fait sans développement supplémentaire. Les deux interlocuteurs du tableau d'affichage peuvent donc interagir avec ce dernier. Ils peuvent de plus dialoguer entre eux *via* le relais constitué par les structures de données de la couche neutre. Cette dernière se comporte comme une passerelle dynamique assurant, à l'aide d'un référentiel d'interfaces, la correspondance entre les appels DSA

et les appels CORBA. L'évolution de l'application n'implique aucun développement sur les exécutifs des nœuds de l'application, elle peut donc être envisagée dans le cadre d'une démarche de prototypage. Notons que l'utilisation de GIOP comme unique protocole de communication entre les nœuds est parfaitement légitime (bien que moins intéressante du point de vue de l'illustration).

7.3. *Factorisation du code*

Le taux de réutilisation du code mesure le degré de généricité qu'apporte une architecture schizophrène. Dans ce but, nous avons comparé le nombre de lignes de code entre les parties génériques et spécifiques dans PolyORB et dans Jonathan, deux intergiciels génériques dont les sources sont disponibles.

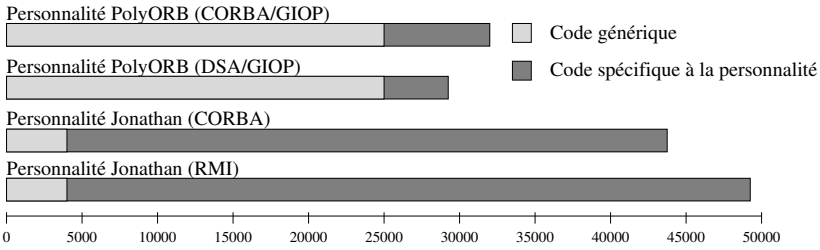

Figure 9. *Factorisation de code entre PolyORB et Jonathan*

La figure 9 fournit un comparatif entre deux configurations distinctes. La première consiste en une personnalité complète, protocolaire et applicative, pour CORBA. La seconde concerne la personnalité de RMI pour Jonathan et celle de DSA pour PolyORB, DSA étant à Ada ce que RMI est à Java. Une analyse des résultats montre que la couche neutre de PolyORB représente une partie significative comprise entre 75 % et 85 % de l'infrastructure de répartition. Le faible taux de réutilisation de Jonathan s'explique par la présence de nombreuses interfaces abstraites qui nécessitent une concrétisation.

7.4. *Performances*

Les mesures que nous indiquons dans le tableau 1 concernent le temps d'exécution en seconde pour 10 000 appels de méthodes prenant un entier long comme paramètre et retournant sa valeur. Les performances de PolyORB sont meilleures que celles de Jonathan alors que celui-ci utilise des souches et des squelettes statiques. N'ayant pas pu compiler Jonathan avec GCC pour Java, nous n'en avons évalué qu'une mise en œuvre interprétée. Les performances de PolyORB sont actuellement inférieures à celles d'un intergiciel comme omniORB réputé pour sa rapidité. L'écart mesuré est surtout important côté serveur, notamment lorsque l'intergiciel est configuré pour utiliser plusieurs tâches.

Client \ Serveur	omniORB	Jonathan	PolyORB sans parallélisme	PolyORB avec parallélisme
omniORB	1,382	3,713	1,785	3,340
Jonathan	4,168	7,090	4,577	8,042
PolyORB	1,870	4,390	2,056	3,602

Tableau 1. *Temps d'exécution de 10 000 appels de méthodes*

Cet écart avec omniORB demeure acceptable d'autant plus que de nombreuses optimisations sont actuellement envisagées dans la bibliothèque de parallélisme, ainsi que dans l'adaptateur d'objets, afin de les rendre plus efficaces. Par ailleurs, les fonctionnalités apportées (interopérabilité entre intergiciels et entre modèles de répartition) représentent un avantage par rapport à l'ensemble des intergiciels existants.

8. Conclusion et perspectives

Un *intergiciel schizophrène* est un intergiciel générique disposant de plusieurs personnalités cohabitantes et interagissantes. Nous avons proposé une architecture pour de tels intergiciels. Elle découple la partie applicative de la partie protocolaire grâce à une couche neutre du point de vue du modèle de répartition.

Grâce à cette couche neutre, un intergiciel schizophrène fournit automatiquement un mécanisme de passerelle et jugule l'explosion combinatoire qu'introduit l'interopérabilité entre modèles de répartition. Par ailleurs, il accorde une place importante à la factorisation de composants fondamentaux que nous avons identifiés. La personnalisation ne nécessite donc que la concrétisation d'un nombre limité de composants et le développement d'un volume réduit de code spécifique.

PolyORB [PAU 01b] constitue le premier intergiciel schizophrène : configurable, personnalisable et interopérable entre modèles de répartition. Techniquement, sa couche neutre généralise les mécanismes d'invocation et d'implémentation dynamique d'interfaces présents dans CORBA. PolyORB met en œuvre le principe de schizophrénie afin de résoudre un problème de génie logiciel synthétisé par le concept de M2M (*Middleware To Middleware*) : l'hétérogénéité des modèles de répartition. Aussi, cet intergiciel nous a-t-il permis de valider notre approche sur des exemples complexes en faisant interopérer de manière effective des services répartis conçus dans des modèles de répartition différents comme CORBA, DSA ou MOM/JMS. Les analyses que nous avons menées sur PolyORB ont démontré qu'un intergiciel schizophrène induit un taux moyen de factorisation du code de 80 % et possède également une forte configurabilité (notamment par le choix d'un exécutif sans parallélisme, avec parallélisme complet ou profilé pour le temps réel).

Le prototypage d'intergiciels exige la définition préalable d'une architecture canonique comme celle généralement adoptée dans les systèmes d'exploitation autour de la gestion des processus, de la mémoire ou encore des entrées/sorties. L'architecture

schizophrène apporte en cela certains éléments de réponse et peut donner lieu selon nous à la définition d'une interface portable permettant de contrôler le comportement des intergiciels en général (à la manière de POSIX). Nous montrons également que notre intergiciel schizophrène a non seulement permis de développer rapidement une nouvelle personnalité comme celle d'un MOM mais que par ailleurs, celle-ci a donné lieu à un outil tout à fait opérationnel. Nos travaux actuels s'orientent vers la mise en place d'une démarche permettant de guider les concepteurs de nouvelles personnalités, tant applicatives que protocolaires.

Si l'architecture schizophrène fournit une organisation claire favorable au développement rapide de nouveaux intergiciels, elle permet également la mise en œuvre de passerelles dynamiques essentielles dans le cadre d'interopérabilité entre modèles de répartition. Cette propriété de la schizophrénie facilite le prototypage de systèmes répartis notamment lorsque certains composants existants sont issus de modèles de répartition différents. Cette approche limite les difficultés liées au développement et la maintenance de passerelles statiques (ou construites pour coupler deux modèles de répartition définis). Même si l'interopérabilité sémantique reste une problématique ouverte, nous considérons qu'en matière d'interopérabilité technique, les intergiciels schizophrènes permettent d'accélérer le processus de synthèse de programmes pour systèmes répartis à composants.

L'intégration des mécanismes de la schizophrénie dans une démarche de développement par prototypage apporte également de nombreuses perspectives pour l'optimisation du déploiement d'une application répartie. En effet, il est possible de configurer un à un les exécutifs de chacun des nœuds de l'application mais également de choisir les protocoles les plus adaptés aux interactions entre nœuds. L'intégration de mécanismes de mesures configurables dans la couche neutre de PolyORB constitue à ce titre une direction potentielle de travail.

9. Bibliographie

[ASU 93] ASUR S., HUFNAGEL S., « Taxonomy of Rapid-Prototyping Methods and Tools », *4th Int'l Workshop on Rapid System Prototyping*, North Carolina, USA, juin 1993, IEEE Computer Society Press, p. 42-56.

[AZA 99] AZAVANT F., COTTIN J.-M., NIEBEL V., PAUTET L., PONCE S., QUINOT T., TARDIEU S., « CORBA and CORBA Services for DSA », *Proceedings of SigAda'99*, Redondo Beach, CA, USA, octobre 1999, ACM Press.

[BAK 01] BAKER S., « A2A, B2B — Now We Need M2M (Middleware to Middleware) Technology », *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, septembre 2001, IONA Technologies, Keynote speech.

[BAN 99] BANAVAR G., CHANDRA T., STROM R., STURMAN D., « A Case for Message Oriented Middleware », *13th International Symposium on Distributed Computing*, Bratislava, Slavak Republic, 1999, p. 1-18.

[BIR 84] BIRRELL A. D., NELSON B. J., « Implementing Remote Procedure Calls », *ACM Transactions on Computer Systems*, vol. 2, n° 1, 1984, p. 39-59, ACM Press.

[BRO 97] BROSE G., « JacORB : Implementation and Design of a Java ORB », Cottbus , Germany, septembre 1997.

[CZE 92] CZECH Z. J., HAVAS G., MAJEWSKI B. S., « An optimal algorithm for generating minimal perfect hash functions », *Information Processing Letters*, vol. 43, n° 5, 1992, p. 257–264.

[DIE 94] DIETZFELBINGER M., KARLIN A., MEHLHORN K., MEYER AUF DER HEIDE F., ROHNERT H., TARJAN R. E., « Dynamic perfect hashing : upper and lower bounds », *SIAM Journal on Computing*, vol. 23, n° 4, 1994, p. 738-761.

[DOB 98] DOBBING B., BURNS A., « The Ravenscar tasking profile for high integrity real-time programs », *Proceedings of SigAda'98*, Washington, DC, USA, novembre 1998.

[DUM 98] DUMANT B., HORN F., TRAN F. D., STEFANI J.-B., « Jonathan : an Open Distributed Processing environment in Java », *IFIP Int'l Conference on Distributed Systems Platforms and Open Distributed Processing*, Londres, 1998, Springer Verlag, p. 175-190.

[EGY 99] EGYED A., MEDVIDOVIC N., « Extending Architectural Representation in UML with View Integration », *Proceedings of the 2nd International Conference on the Unified Modeling Language*, Fort Collins, CO, USA, octobre 1999, p. 2-16.

[FOR 95] FORUM M. P. I., « MPI : A Message-Passing Interface Standard », juin 1995, http://www.mpi-forum.org/docs/mpi-11.ps.Z.

[GEI 97] GEIB J., GRANSART C., MERLE P., *CORBA : des concepts à la pratique*, Masson, Collection InterEditions, 1997.

[GIB 94] GIBBS W., « Software's Chronic Crisis », *Scientific American*, vol. 271, n° 3, 1994, p. 72-81.

[GIL 02] GILLIERS F., KORDON F., REGEP D., « Proposal for a Model Based Development of Distributed Embedded Systems », *Monterey Workshop on Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, octobre 2002.

[GOL 96] GOLDBERG A., PRINS J., REIF J., FAITH R., Z. LI P. M., NYLAND L., PALMER D., RIELY J., WESTFOLD S., « *The Proteus System for the Development of Parallel Applications* », p. 151–190, Springer-Verlag, 1996.

[IBM 97] IBM, « MQ Series », 1997, http://www-3.ibm.com/software/ts/mqseries.

[ING 78] INGALLS D., « The Smalltalk-76 Programming System Design and Implementation », *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, janvier 1978, p. 9-16.

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

[KEL 94] KELEHER P., COX A., S., DWARKADAS, ZWAENEPOEL W., « ThreadMarks : Distributed Shared Memory on standard workstations », *1994 Winter Usenix Conference*, janvier 1994, p. 115-132.

[KER 95] KERR J., HUNTER R., *Inside RAD*, McGraw Hill, 1995.

[KER 96] KERMARREC Y., NANA L., PAUTET L., « GNATDIST : A Configuration Language for Distributed Ada 95 Applications », *Proceedings of TRI-Ada'96*, ACM Press, 1996, p. 63-72, http://www.infres.enst.fr/~pautet/papers/pautet96gnatdist.ps.

[KOR 95] KORDON F., « Prototypage automatique : de la maquette au système final », *Colloque sur la Conception de Systèmes*, Paris, France, avril 1995.

[KOR 02] KORDON F., LUQI, « An introduction to Rapid System Prototyping », *IEEE Transaction on Software Engineering*, vol. 28, n° 9, 2002, p. 817-821.

[KUR 93] KURPIS G., BOOTH C., « The new IEEE Standard Dictionnary of Electrical and Electronic Terms », rapport, 1993, IEEE.

[LEV 97] LEVESON N., « Software Engineering : Stretching the Limits of Complexity », *Communications of the ACM*, vol. 40, n° 2, 1997, p. 129-131.

[MED 00] MEDVIDOVIC N., TAYLOR R., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, 2000, p. 70-93.

[MER 97] MERLE P., « CorbaScript - CorbaWeb : propositions pour l'accès à des objets et services distribués », Thèse de doctorat, Université des Sciences et Technologies de Lille (USTL), janvier 1997.

[MUR 91] MURPHY S., GUNNINGBERG P., KELLY J.-P., « Experiences with Estelle, LOTOS and SDL : A Protocol Implementation Experiment », *Computer Networks and ISDN Systems*, vol. 22, n° 1, 1991, p. 51-59.

[OBR 00] OBRY P., « AWS : Ada Web Server », *Actes d'Ada-France, session spéciale Ada et Internet*, Ada France, novembre 2000.

[ODP 95] ODP, « ODP Reference Model : Overview, ITU-T -- ISO/IEC Recommendation X.901 -- International Standard 10746-1 », 1995.

[OMG 98] OMG, *CORBA Messaging Specification*, OMG, mai 1998, OMG orbos/98-05-05.

[OMG 01] OMG, « Model Driven Architecture (MDA), Document number ormsc/2001-07-01 », rapport, 2001, OMG.

[OMG 02] OMG, *The Common Object Request Broker : Architecture and Specification, revision 3.0.2*, OMG, décembre 2002, OMG formal/2002-12-02.

[OMG 03] OMG, *OMG Unified Modeling Language Specification, version 1.5*, mars 2003, formal/2003-03-01.

[PAU 00] PAUTET L., TARDIEU S., « GLADE : a Framework for Building Large Object-Oriented Real-Time Distributed Systems », *Proceedings of ISORC'00*, Newport Beach, California, USA, juin 2000, IEEE Computer Society Press.

[PAU 01a] PAUTET L., « Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition », Habilitation à Diriger des Recherches, Université Pierre et Marie Curie – Paris VI, décembre 2001, http://www.infres.enst.fr/~pautet/papers/pautet01hdr.ps.

[PAU 01b] PAUTET L., QUINOT T., KORDON F., TARDIEU S., AZAVANT F., NIEBEL V., PONCE S., GINGOLD T., « PolyORB », 2001, http://libre.act-europe.fr.

[QUI 99] QUINOT T., KORDON F., PAUTET L., « CIAO », 1999, http://adabroker.eu.org/ciao.

[QUI 01] QUINOT T., PAUTET L., KORDON F., « Architecture for a reuseable object-oriented polymorphic middleware », *Proceedings of PDPTA'2001*, Las Vegas, Nevada, Etats-Unis, juin 2001, http://www.infres.enst.fr/~pautet/papers/pautet01architecture.ps.

[QUI 03] QUINOT T., « Conception et réalisation d'un intergiciel schizophrène pour la mise en oeuvre de systèmes répartis interopérables », PhD thesis, University P. & M. Curie, 2003.

[SCH 95] SCHÄFERS L., SCHEIDLER C., KRÄMER-FUHRMANN O., « Software Engineering for Parallel Systems : The TRAPPER Approach », *28th Hawaiian International Conference on System Sciences*, janvier 1995.

[SCH 96] SCHMIDT D. C., STAL M., ROHNERT H., BUSCHMANN F., *Patterns for Concurrent and Networked Objects*, vol. 2 de *Pattern-Oriented Software Architecture*, John Wiley & Sons, 1996.

[SCH 97] SCHMIDT D., CLEELAND C., « Applying patterns to develop extensible and maintainable ORB middleware », *CACM*, vol. 40, n° 12, 1997, ACM Press.

[SHT 98] SHTIL S., BHATIA V., « Rapid Prototyping Technology Accelerates Software Development for Complex Network Systems », *9th Int'l Workshop on Rapid System Prototyping*, Leuven, Belgium, juin 1998, IEEE Computer Society Press, p. 113–115.

[SIN 98] SINGHAI A., SANE A., CAMPBELL R., « Quarterware for Middleware », *Proceedings of ICDCS'98*, Amsterdam, The Netherlands, mai 1998, IEEE, http://choices.cs.uiuc.edu/singhai/Papers/icdcs98.pdf.

[SUN 99] SUN, « Java Message Service », 1999.

[W3C00] W3C, « Simple Object Access Protocol (SOAP) 1.1 », mai 2000, W3C note, http://www.w3.org/TR/SOAP/.

Laurent Pautet *est maître de conférence, habilité à diriger des recherches, à l'Ecole Nationale Supérieure des Télécommunications (ENST). Ses recherches portent sur les plates-formes logicielles pour les systèmes répartis. Elles concernent les modèles, les architectures et les composants logiciels ou algorithmiques nécessaires à la réalisation et au déploiement d'infrastructures réparties pour les systèmes embarqués, temps réel ou mobiles.*

Fabrice Kordon *est professeur d'informatique à l'Université Pierre & Marie Curie (Paris VI) et responsable du thème "Systèmes Répartis et Coopératifs" au Laboratoire d'Informatique de Paris 6 (LIP6). Ses recherches sont centrées sur le prototypage de systèmes répartis. Elles concernent l'aide au développement de composants logiciels et des infrastructures dédiées à la répartition. Il s'intéresse en particulier aux techniques (modélisation, méthodes de vérification formelles, génération automatique de code, etc.) permettant d'assister les ingénieurs et leur permettre de maîtriser la complexité du développement de systèmes répartis.*

Article reçu le 5 décembre 2002
Version révisée le 24 novembre 2003
Rédacteur responsable : PHILIPPE MERLE