Les middlewares réflexifs

No Author Given

Institut d'informatique des Facultés Universitaire Notre Dame de la Paix Namur

Résumé. Un système réflexif est un système possédant une auto-représentation qui décrit la connaissance qu'il a de lui-même, lui permettant de répondre à des questions le concernant et de modifier ses propres mécanismes de représentation et d'exécution. Ce papier décrit le contexte d'émergence de tels systèmes, les concepts liés à la réflexion ainsi que trois implémentations de middlewares réflexifs.

1 Introduction

Les middlewares classiques du type CORBA (*Common Object Request Broker Architecture*), java RMI (*Remote Methode Invocation*) et Microsoft .NET ont été développés ces dernières années dans le but de simplifier au maximum le travail d'implémentation d'architectures distribuées. Dans cette optique, ces technologies permettent cacher la complexité d'une série de mécanismes tels que les communications réseaux et l'invocation de méthodes distantes. D'autre part, elles facilitent l'élaboration de systèmes interopérables comprenant des modules implémentés dans différents langages et implantés sur différents systèmes d'exploitation. Ces plates-formes permettent en quelque sorte aux développeurs de travailler sur des applications distribuées un peu comme s'ils travaillaient sur des applications locales.

Aujourd'hui, les exigences en matières d'applications distribuées s'orientent vers l'*adaptabilité*, le *self-healing* ou encore le *self-organising*. toute ces notions font référence à la nécissité de l'adaptation dynamique des systèmes. Cependant, ces nouvelles exigences font défaut aux architectures basées sur les middlewares classiques dont la philosophie *black-box* est de masquer un maximum la complexité. En effet, même si les middlewares offrent une plénitude de services aux utilisateurs, ces services sont souvent trop rigides et ne permettent pas de voir ou de changer leur implémentation en fonction des exigences variant d'une application à l'autre.

Dans ce contexte, plusieurs approches ont été expérimentées afin de résoudre les exigences de plus en plus pressantes des applications distribuées. C'est ainsi que par exemple, l'*Object Management Group* (OMG), a introduit une série de spécifications qui inclut *real-time CORBA* et *Minimal CORBA*. Mais ces spécifications sont limitées à des domaines beaucoup trop spécifiques et ne résolvent donc pas tous les problèmes. Parmi les solutions les plus prometteuses mais aussi les plus ambitieuses, on retrouve différentes implémentations de middlewares dis *réfléxifs*. I.e. des middelwares ouverts (*openess*) qui possèdent une auto-représentation et une capacité d'adaptation.

La suite du document est organisée de la manière suivante : la section 2 présente les différents concepts sous-jacents à la technologies réflexive, la section 3 souligne les avantages d'une découpe en composants pour une architecture réflexive, la section 4 présente les différents challenges qui devraient conduire à la péremption des middlewares classiques, et enfin la section 5 présente trois implémentations abouties de middlewares réflexifs.

2 les concepts sous-jacents aux middlewares réflexifs

On peut définir simplement un système réflexif comme système possédant une auto-représentation qui décrit la connaissance qu'il a de lui-même. Un tel système peut dés lors répondre à des questions le concernant (*introspection*) mais aussi modifier ses propres mécanismes de représentation et d'exécution (*intercession*).[LED2]
Les architectures réflexives sont séparées de façon plus ou moins explicite (selon les implémentations) en deux niveaux permettant de distinguer ce que fait un objet (niveau de base) de comment il le fait (niveau méta). *Il existe donc une séparation clairement définie entre les fonctionnalités de l'application, programmées au niveau de base, et leurs représentations et contrôles, programmés au méta-niveau.*[LED1] Les objets se retrouvant dans le méta-niveau sont appelés *métaObjets* et les protocoles d'interaction entre ces métaObjets et les objets du niveau de base sont appelés *méta-object protocol* ou MOP.

Associé au concept d'auto-représentation sous-jacent au principe de réflexivité, on retrouve souvent dans la littérature les deux concepts suivants :

1. La *réification* qui consiste à extraire une représentation interne d'un système en terme d'entités qui peuvent être manipulées à l'exécution.
2. L'*absorption* qui consiste à effectuer les changements appliqués aux entités *réifiées* dans le système. C'est à dire le processus inverse de la *réification*.

Une distinction est encore souvent marquée [KON1] [BLA1] entre deux types de réflexion. On distingue la *réflexion structurelle*, c'est la capacité d'un system (ou d'un langage) à fournir une réification complète du système en marche en terme des interfaces supportées. Le programmeur peut dés lors inspecter ou (dans les systèmes plus évolués) changer les fonctionnalités du programme. Ce type de réflexion comprend les accès à l'architecture même[1] aussi appelée *réflexion architecturale*. Enfin, on distingue la *réfexion comportementale* : la capacité d'un système (ou d'un langage) à fournir une réification de la sémantique du système en terme d'aspects liés à son environnement d'exécution. Le programmeur peut alors inspecter ou changer la façon dont le programme s'exécute dans un environnement donné en fonction des propriétés dites non-fonctionnelles.

3 L'importance des composants

Selon le modèle de Darwin explicité dans [GEO], un composant peut être définit comme "Une unité de composition fournissant et nécessitant des services". Dans ce contexte, un service est fourni ou requis via un port typé dont le type correspond au type de l'interface utilisée pour accéder au service. Notons enfin que le typage supporte la relation d'héritage. La figure 1 représente un composant fournissant les deux services T0 et T1 et nécessitant le service T0.

Cette définition met en évidence l'existence de plusieurs avantages dans l'utilisation d'architectures basées sur des composants :
Tout d'abord on peut remarquer que les liens entre composants sont facilement explicitables, en terme de graphes par exemple, où les composants sont représentés par des noeuds et les liens entre composants par des arcs. Ceci facilite grandement la réification de la structure interne du système. D'autre part, selon cette définition, un composant n'est jamais lié sémantiquement à un autre. En effet, il est lié à une série de services fournis par un ou plusieurs autres composants. En terme d'adaptabilité, il devrait donc être possible de trouver les bons services pour les bons composants dans un contexte donné.

Enfin, notons encore que les composants de par leur indépendance peuvent être développés séparément.

4 De nouveaux défis

4.1 L'adaptabilité des systèmes

L'adaptabilité des systèmes est évidemment le défit majeur des middlewares réflexifs. Mon but ici n'est pas de rappeler les concepts déjà abor-

[1] On parle ici d'architecture de composants liés par des connecteurs. Une explication plus détaillée se retrouve dans la section 3.

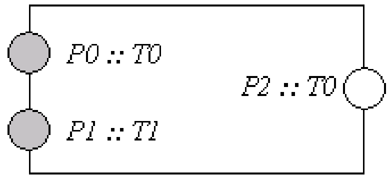

Fig. 1. Un composant

dés dans l'introduction et la section 2 mais de les illustrer par quelques exemples pour en définir la portée.

Tout d'abord, imaginons un système proposant un service de vidéo-conférence. Pour ce type de service, la bande passante est un facteur clé puisqu'elle va déterminer la qualité du flux vidéo (et audio) échangé entre les différents acteurs. Ce type de système trouvera donc un avantage certain dans la possibilité d'*adapter* les types de connexion (réseau local câblé, sans fil, ou réseau longue distance (Internet)) en fonction des disponibilités.
Imaginons ensuite un système pour lequel les accès aux services varient fortement. Plusieurs techniques existent déjà pour pallier la surcharge d'utilisateurs, parmi elles, citons la migration et la réplication des services qui sont les plus connues. Un système *adaptable* permettrait dans un premier temps au serveur d'adapter son service en fonction de la charge et ensuite au niveau du client de modifier le type d'invocation distante en fonction de la politique adaptée par le serveur.
Enfin, pour fournir toujours plus de portabilité et d'utilisabilité aux applications, on pourrait encore imaginer un système qui fournirait des interfaces graphiques *adaptées* aux différents utilisateurs en fonction de leurs différentes plates-formes (PC, PDA, GSM ...).

On voit dés lors que les possibilités de l'adaptabilité sont aussi nombreuses que variées. On peut aussi remarquer que les besoins et bénéfices en matière d'adaptabilité sont spécifiables par les développeurs d'application. Les nombreuses implémentations des middlewares réflexifs répondent en générale bien à ces besoins grâce au niveau méta. Cependant d'autres défis sont beaucoup plus complexes et nécessitent des mécanismes supplémentaires pour les relever.

4.2 La gestion des QoS

Bien que les applications distribuées se voient offrir de plus en plus de services par les middlewares classiques, elles introduisent aussi des nouveaux besoins en terme d'exigences non fonctionnelles relatives aux QoS (*Quality Of Service*). Dans son acceptation générale, la notion classique de QoS reprend des mesures de performance telle que le débit d'un réseau ou la latence d'un service. Cela dit, une vue plus générale des QoS reprend d'autres aspects plus complexes [TRI] tels que : la fiabilité, la disponibilité, la sécurité, les garanties temps-réel et la synchronisation de flux de données dans la distribution multimédia. Les nouveaux middlewares devraient permettre l'implémentation de *policies* complexes pour la gestion des QoS. Selon [TRI] les méchanismes basés sur la réflexion méta-niveau sont nécessaires mais non suffisants pour cette gestion dynamique des exigences de QoS. Selon lui, les middlewares *Next Generation* devraient aussi supporter la gestion indépendante des aspects ainsi que des mécanismes permettant une allocation, basée sur les QoS, des couches réseaux ainsi que des ressources au niveau des systèmes d'exploitation. [ELI] apporte une formulation plus précise des exigences QoS :

- **Dynamic QoS support** : les middlewares devraient pouvoir fournir les QoS demandés par les applications et être capable de s'adapter en fonction de ceux-ci.
- **Evolution of QoS requirements** : de nouveaux types de QoS peuvent apparaitre dans les nouvelles applications, la définition des QoS doit donc pouvoir être extensible.
- **Transparency versus fine-grained control** : les middlewares devrait pouvoir supporter la définition des QoS en terme de spécifications de haut-niveau (application) et de bas niveaux (paramètres systèmes, allocation de ressources, ...)
- **Policy control** : les middlewares devraient permettre aux différents acteurs (gestionnaire de système, développeur d'application et utilisateur final) de spécifier des *policies* pour gérer les QoS.
- **Automatic support for compatibility control** : les middlewares devraient pouvoir détecter les incompatibilités dans les exigences des utilisateurs et les résoudre.
- **Support for seamless system evolution** : les middlewares devraient pouvoir supporter l'intégration de nouveaux composants sans recompilation du système et sans modification des composants existants.

4.3 La composition sécurisée des services

Le défit le plus ambitieux arrive cependant avec la difficulté d'obtenir une composabilité sécurisée [VEN](*safe 'composability'*) des composants sous une exécution concurrente d'un certain nombre de services.

Pour mieux comprendre l'enjeu de la concurrence des services prenons l'exemple d'un service de migration s'exécutant en même temps qu'un service de "garbage collection". On devine alors aisément un risque de non terminaison pour le processus de "garbage collection" et un risque d'inconsistance pour les données migrées par le service de migration. Plus généralement, les problèmes liés à la concurrence sont les deadlocks, les livelocks, la non terminaison de processus, la perte d'informations et les problèmes d'inconsistance.
Le procédé le plus connu pour éviter les problèmes de concurrence, est la sérialisation. Cependant ce procédé peut entraîner des chutes de performance considérables totalement incompatibles avec les exigences QoS des applications.

La solution proposée par [VEN] est de définir dans l'espace méta du middleware des services de base et non-interférable entre eux. Ces services permettent alors de garantir les contraintes de "composabilité sécurisée" et peuvent être réutilisés par les concepteurs de système pour développer des librairies de services et de protocoles plus évolués. On retrouve trois services de base :

- Remote creation : gère la re-création de services ou de données sur un site distant ;
- Distributed snapshot : gère la capture d'informations liées sur plusieurs noeuds ou sites ;
- Directory services : gère les interactions avec un *repository* global.

5 Trois implémentations

Cette section présente trois implémentations d'ORB réflexif : DynamicTAO, OpenORB et OpenCORBA. Ces trois implémentations sont basées sur la technologie CORBA mais diffèrent sensiblement dans la façon d'utiliser le principe de réification. Ces trois implémentations ne représentent évidemment pas un ensemble exhaustif des middlewares réflexifs existant mais sont parmi les plus abordées et les plus abouties.

5.1 DynamicTAO

DynamicTAO est une extension *réflexive* de la version C++ de l'ORB TAO (University of Illinois).
TAO *est un ORB portable, flexible, extensible et configurable basé sur des design patterns orienté objet.*[KON2] Il a comme principale caractéristique de représenter des différents aspects configurables de l'ORB sous la forme

de designs pattern *strategy*. La configuration du système est alors possible via un fichier qui permet de spécifier les différentes stratégies à appliquer par l'ORB avant le lancement du système. Cependant cette plate-forme souffre, comme tous les middlewares classiques, d'une incapacité de reconfigurer l'ORB pendant son exécution. C'est pourquoi TAO à laissé place à dynamicTAO.

Le principe de réification dans dynamicTAO est basé sur l'utilisation de configurateurs (*ComponentConfigurators*) pour gérer chaque composant. Il s'agit d'objets C++ qui stockent les dépendances entre chaque composant de l'ORB ainsi que les dépendances entre l'ORB et les composants d'application. A chaque processus tournant sur l'ORB est associé un type de configurateurs (*DomainConfigurator*) qui maintient des références vers les instances de l'ORB qu'il utilise ainsi que vers les servants tournant dans ce processus. De plus, à chaque instance de l'ORB est associé un autre type de configurateurs (*TAOConfigurateur*) qui contient des crochets[2] (*hooks*) vers les stratégies TAO utilisées dans une implémentation. Les associations entre les crochets et les composants peuvent alors être reconfigurées à l'exécution moyennant certaines contraintes de composition. Enfin, les configurateurs peuvent aussi servir d'une part pour exprimer des relations de dépendance entre les différentes stratégies et d'autre part pour stocker des informations sur les connexions des clients pouvant influencer le choix des stratégies.
Ces objets configurateurs ont la particularité de pouvoir être spécialisés par technique d'héritage. Ce qui permet de fournir une implémentation adaptée à chaque type de composant. Cette spécialisation est particulièrement utile dans un contexte dynamique car elle permet de définir des contraintes de consistance sur la structure interne de l'ORB. DynamicTAO permet ainsi de traiter des aspects comme la concurence, la sécurité et le monitoring. La figure 2 extraite de [KON1] représente la structure de dépendance entre les différents types de configurateurs.

DynamicTao fournit un ensemble d'interfaces permettant aux utilisateurs de reconfigurer sa structure. Il fournit une méta interface permettant le chargement et le déchargement dynamiques des composants. Ceux-ci sont d'abord transformés sous forme de DDLs (*Dynamically Loadable Library*) qui facilitent leur liaison au système en exécution et sont organisés en *catégories* reprenant les différentes stratégies nécessaires à leur exécution. De plus, pour faciliter la reconfiguration de composants distribués et

[2] Un crochet est définit comme un "point montage". En d'autres termes, ils répondent à la question : à quel aspect correspond quelle stratégie à un moment donné de l'exécution du système ?

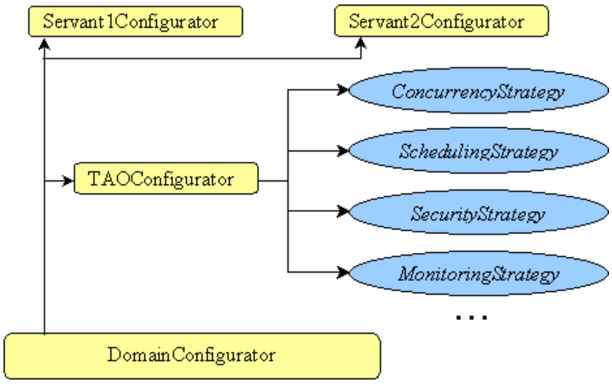

Fig. 2. Les configurateurs dans dynamicTAO

interconnectés, dynamicTAO fournit aussi une méta interface permettant la gestion d'agents mobiles. Cette interface permet à l'utilisateur de spécifier dans un premier temps sous forme de graphe les noeuds du réseau sur lesquels les agents agiront, et ensuite de spécifier une liste d'actions de reconfiguration qui seront appliquées localement à chaque noeud du réseau.

On retrouve encore dans l'implémentation de DynamicTAO la possibilité d'insérer des *intercepteurs* entre différents composants ou différentes applications. Ces *intercepteurs* peuvent remplir diverses fonctionnalités comme la cryptographie, la compression, le contrôle d'accès, le monitoring, etc.

Enfin, pour ce qui est du traitement des ressources, DynamicTAO utilise le *Dynamic Soft Real-Time Scheduler* (DSRT). DSRT fonctionne comme processus utilisateur dans les systèmes d'exploitation et permet de satisfaire les exigences en matière de QoS (*Quality of Service*) des applications. I.e. : négociation de ressources, réservation, programmation temps réelle, etc.

5.2 Open ORB

Contrairement à DynamicTAO, Open ORB (Lancaster university) est une implémentation *from scratch* d'un ORB CORBA réflexif basé sur le langage Python [PYT]. Le caractère réflexif est ici beaucoup plus marqué par la nette distinction entre le niveau de base (*base-level*) qui comprend les composants implémentant les sevices classiques du middleware et le niveau méta (*meta level*) qui comprend les composants permettant de (re)configurer le niveau de base. L'approche est particulièrement intéressante dans le sens où le niveau méta est structuré de la même manière que le niveau de base, ce qui permet une (re)configuration via un autre niveau (*méta méta niveau*) qui est lui-même (re)configurable et ainsi de suite. A chaque niveau correspond donc un environnement qui est configurable par une méta interface.[3]

L'espace méta est partitionné en quatre classes de modèles dont le but avoué [BLA1] est "*de simplifier l'interface que le méta espace offre en séparant les sujets en différents aspects du système*". Ces quatre classes de modèles sont regroupées par deux dans une logique de séparation entre les deux types de réflexion : la réflexion structurelle et la réflexion comportementale. Pour le premier type de réflexion (réflexion structurelle), on retrouve :

- **le méta modèle d'interfaces** : qui fournit un accès à la représentation externe des composants en terme d'interfaces requises et fournies (qui rappelle le modèle de Darwin exposé dans la section 3). Le MOP (*meta object protocol*) associé permet alors de définir les services fournis par un composant et de trouver des composants pour les interfaces requises. Notons tout de même qu'il n'est pas possible d'accéder à l'implémentation interne d'une interface en terme de méthodes et d'attributs. Cette restriction est cependant guidée par la volonté de se restreindre à une programmation orientée composant.
- **le méta modèle d'architectures** : qui définit la partie interne des composants en terme d'architecture logiciel. Il comprend deux éléments : un graphe explicitant la structure du composant en terme de sous-composants et un ensemble de contraintes architecturales. Les sous-composants sont reliés entre eux par des liens locaux (i.e. une correspondance entre une interface requise et une interface fournie dans un même espace d'addressage) ou des liens distribués tels que l'invocation distante, le publish-suscribe, les liens de flux continu, etc. Le MOP associé permet donc alors de découvrir et de changer cette structure plus précise. L'ensemble de contraintes permet de spécifier certaines limitations dans les modifications en temps réel du système. Avant qu'une modification soit réellement appliquée au système, elle est validée par les ensembles de contraintes des composants concernés.

Et pour le second (réflexion comportementale) :

[3] Pour éviter le problème de récursion infinie de méta-niveaux, le modèle instantie seulement les métaComposants sur demande.

- **les modèles d'interception** : Un MOP permet ici de gérer des exigences non fonctionnelles via la manipulation dynamique (ajout, retrait) d'*intercepteurs* entre les différents composants. Tout comme dans DynamicTAO, il est ainsi possible d'adapter le comportement pré/post d'un service lié à un composant.
- **les méta modèles de ressources** : Ce dernier espace de modèles permet la gestion dynamique des ressources requises par les différentes tâches du système. Les tâches du système sont "*les unités logiques des activités du système dont la granularité dépend d'une configuration à une autre*"[BLA1]. Ce qui implique que les tâches peuvent s'organiser sur plusieurs composants et que leur structuration doit être définie orthogonalement par rapport aux modèles d'interface et d'architecture. Il y a un méta modèle de ressources par espace d'adresse, ce qui signifie que tous les composants d'un espace d'adresse partagent le même modèle de ressources. Ce méta modèle est encore représenté sous forme de graphe d'interconnexion qui permet encore l'introspection et l'adaptation de la structure. Les noeuds de ce graphe sont alors ici : les ressources, les tâches et les gestionnaires de ressources. Plus précisément, cette structure est définie dans un ADL (*Architecture Description Language*) appelé Xelha qui permet notamment la définition de contraintes de QoS associée aux tâches.

La figure 3 extraite de [BLA2] reprend ces différents méta modèles et leurs intéractions avec les composant du niveau de base.

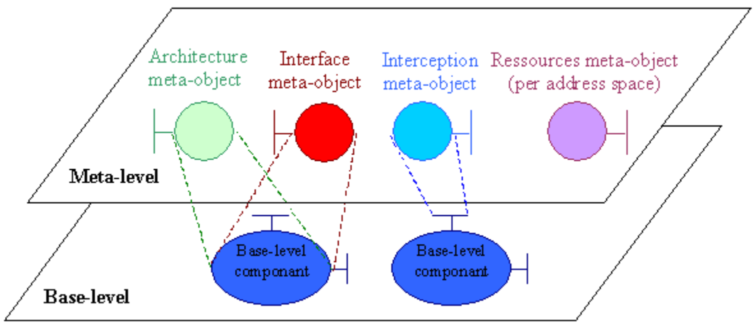

Fig. 3. les méta modèles de OpenORB.

5.3 OpenCorba

OpenCorba (Ecole des Mines de Nantes) est une autre implémentation d'un bus CORBA réflexif. Il est basé sur le langage réflexif NeoSmalltalk qui est le résultat de l'implémentation d'un MOP (*Méta Object Protocol*) dans le langage Smalltalk. On retrouve encore dans OpenCorba la volonté de distinguer les propriétés fonctionnelles d'une classe des propriétés non fonctionnelles (aussi appelées *propriétés de classe* dans [LED1]) en deux niveaux distincts : le niveau de base et le niveau méta.

L'idée originale de OpenCorba consiste en l'utilisation du *changement dynamique de méta classe* à l'exécution qui est une des principales caractéristiques du MOP de NeoSmalltalk. L'implémentation de OpenCorba comprend une série de méta classes représentant différents aspects pour les services classiques définis par l'OMG pour CORBA. Il est alors possible de définir des alternatives dans le choix du comportement d'un service en implémentant différentes méta interfaces pour ce service.
NeoSmalltalk comprend une méta classe par défaut (*StandardClass*) qui est la racine de toute les méta classe NeoSmalltalk et qui définit le comportement standard des classes. Le *changement dynamique de méta classe* permet alors d'adapter les comportements des services en fonction du contexte d'exécution en basculant d'une méta classe standard vers une méta classe modélisant un aspect particulier et en rebasculant vers la méta classe standard ensuite.
Par exemple, le mécanisme d'invocation distante sera représenté par une méta classe car il correspond à une *propriété de classe* pour les classes de type Proxy. Pour éviter les embouteillages au niveau du serveur, on utilise parfois des techniques de migration de composants. Pour supporter cette technique, une autre méta classe "local invocation" pourra être implémentée afin de fournir un accès local aux composants serveurs migrés. De plus, le MOP NeoSmalltalk permettra alors à la classe Proxy de changer de méta classe en fonction du contexte.

Cependant, cette approche souffre encore par la difficulté de composer différents aspects modèlisés par différentes méta classes. Le problème survient lorsqu'il y a chevauchements de comportement. Afin de pallier ce problème, des travaux récents autour de OpenCorba ont mené à la définition d'un *modèle de compatibilité* de méta classes afin d'offrir un cadre fiable pour la composition des méta classe [LED1].

6 Conclusion

Les middlewares réflexifs ont été développés dans le but de répondre aux exigences d'adaptation dynamique des applications distribuées. Peu importe les différences apparentes dans l'implémentation du principe de réification, les travaux existants répondent en général correctement à cet objectif. Cela dit, les exigences des applications sont en constante évolution et de nouveaux défis sont confrontés à ces plates-formes. Rappelons ici la difficulté de composer des services concurrents que l'on retrouve notamment dans OpenCorba. Notons aussi ici que dynamicTAO n'existe quasi plus que dans les articles qui en parle. Il a en effet laissé place à UIC-CORBA, un nouveau type d'ORB réflexif dont le but est de tenir compte des machines ayant des ressources limitées. OpenORB est aussi en constante évolution et a donné lieu à plusieurs versions : récemment, poussé par le même type d'objectif que UIC-CORBA, des modifications ont été apportées notamment en incluant du code C++ pour plus de performance.

Références

[BLA1] Gordon Blair, Geoff Coulson, Lynne Blair : **Reflexion, Self-Awareness and Self-Healing in OpenOrb**. 2002. ACM Special Interest Group on Software Engineering.

[BLA2] Gordon S. Blair, Geoff Coulson, et al. : **The Design and Implementation of Open ORB version 2**. 2001. IEEE Distributed Systems Online Journal 2(6).

[ELI] Frank Eliassen, Thomas Plagemann et al. : **QoS management in the MULTE-ORB**. http ://dsonline.computer.org/middleware/articles/dsonline-MULTE-ORB.html. IEEE Distributed Systems Online Journal.

[GEO] Ioannis Georgiadis, Jeff Magee, Jeff Kramer : **Self-Organising Software Architectures for Distributed Systems**. 2002. ACM Special Interest Group on Software Engineering.

[KON1] Fabio KON1, Fabio Costa, Gordon Blair, Roy H. Campbell : **The case for reflective Middleware**. 2002. ACM special adaptative Middleware.

[KON2] Fabio Kon, Manuel Román et al. : **Monitoring, Security and Dynamic reconfiguration with the dynamicTAO reflective ORB**. 2000. http ://choices.cs.uiuc.edu/2k/papers/middleware2000.pdf

[LED1] Thomas Ledoux : **OpenCorba : a reflective open broker**. http ://www.emn.fr/x-info/ledoux/Publis/reflection99.pdf

[LED2] Thomas Ledoux : **Adaptabilité dynamique des aspects pour la construction d'applications réparties ouvertes**. http ://www.iro.umontreal.ca/NOTERE/Articles/notere-ledoux.pdf

[PYT] Le site officiel pour le langage Python : http ://www.python.org/

[TRI] Anand Tripathi : **Challenges Designing Next-Generation Middleware Systems**. 2002. ACM Special Interest Group on Software Engineering.

[VEN] Nali Venkatasubramanian : **Safe 'composability' of Middleware services**. 2002. ACM special adaptative Middleware.