

---

# Adaptation des connecteurs dans le CCM

**Mathieu Vadet<sup>\*,\*\*</sup> — Philippe Merle<sup>\*\*</sup>**

*\*THALES Communications  
1-5, Avenue Carnot  
91883 Massy CEDEX, France*

*\*\*Laboratoire d'Informatique Fondamentale de Lille  
UPRESA 8022 CNRS - U.F.R. I.E.E.A. - Bâtiment M3  
Université des Sciences et Technologies de Lille  
59655 Villeneuve d'Ascq CEDEX France  
{Mathieu.Vadet, Philippe.Merle}@lifl.fr*

---

*RÉSUMÉ. Les intergiciels à composants tels que le CORBA Component Model (CCM), les Enterprise Java Beans (EJB) ou .NET permettent l'abstraction et la prise en charge automatique par le conteneur des liaisons entre composants. Cependant, les possibilités d'adaptation de la technique de transport sous-jacente sont souvent limitées, voire inexistantes. Dans cet article, nous proposons une architecture dédiée pour adapter la technique de transport des événements dans la plate-forme OpenCCM. Celle-ci se base sur la réification du mécanisme de connexion pour intégrer de façon transparente les connecteurs. Ensuite, nous utilisons une approche descriptive par configuration pour automatiser l'instanciation des connecteurs. Finalement, nous montrons comment l'adaptation peut être réalisée au déploiement de l'application métier ou lors de connexions entre composants grâce à un protocole de propagation des configurations.*

*ABSTRACT. Component-based middleware such as CCM, EJB or .NET provides the abstraction and the automatic management of component binding by the container. Nevertheless, the adaptation possibilities of the underlying transport layer are often limited or even non-existent. In this paper, we propose a dedicated architecture to adapt the transport layer of events in the OpenCCM platform. This one is based on the reification of the connection mechanism to transparently integrate connectors. Then, we use a descriptive approach by configuration to automatically instantiate connectors. Finally, we show how this adaptation can be realized at deployment time or during component connections using a configuration propagation protocol.*

*MOTS-CLÉS : adaptation, connecteurs, CORBA Component Model, OpenCCM, propagation de configurations.*

*KEYWORDS: adaptation, connectors, CORBA Component Model, OpenCCM, configuration propagation.*

---

## 1. Introduction

La construction d'applications distribuées s'appuie de plus en plus sur l'utilisation d'intergiciels à composants, notamment les Enterprise Java Beans (EJB) de SUN Microsystems [SUN 02a], le CORBA Component Model (CCM) de Object Management Group (OMG) [OMG 02a] et .NET de Microsoft Corporation [ECM 00, RAM 02]. En effet, ils permettent la séparation du code métier et du code système au travers du paradigme composant/conteneur. Ici, le composant encapsule le code métier et le conteneur gère automatiquement les propriétés extra-fonctionnelles, c'est-à-dire les services (sécurité, transactions, etc), le cycle de vie du composant (session, entité, etc) ainsi que les interactions entre les composants (synchrone, événementielle, flux, etc). Néanmoins, les différentes approches ne se focalisent pas sur toutes ces extra-fonctionnalités. D'un côté, les normes CCM et EJB spécifient principalement l'intégration entre les services, le cycle de vie et les composants. De l'autre côté, la norme Reference Model of Open Distributed Processing (RM/ODP) de ISO/IEC [JTC 95] se focalise sur la définition, la représentation et la construction des liaisons (vue ingénierie des interactions). Les plates-formes .NET, Jonathan d'ObjectWeb [KRA 02] et JavaPOD du projet SIRAC de l'INRIA Rhône-Alpes [BRU 00] prennent en charge quant à elles l'intégration de liaisons propriétaires. Les *protocoles à méta-objets* (Meta-Object Protocol - MOP) [KIC 91] ou encore la *programmation par aspects* (Aspect Oriented Programming - AOP) [KIC 97] constituent des réponses académiques générales au problème d'adaptabilité des applications. Néanmoins, les MOPs se focalisent plus sur les moyens génériques pour intégrer de façon transparente les adaptations, alors que l'AOP se concentre sur la manière de décrire et de composer les adaptations (voir par exemple Aspect/J [KIC 01]). Ces approches ont aussi été utilisées dans le cadre plus spécifique de l'adaptabilité des extra-fonctionnalités. Ainsi, la plate-forme JavaPOD s'appuie sur la *réification* des appels de méthodes pour adapter les liaisons. Java Aspect Components (JAC) [PAW 01] supporte quant à lui le *tissage* d'aspects dynamique pour adapter à la fois les services et les liaisons.

Dans cet article, nous présentons une approche architecturale pour adapter les interactions dans le cadre de la norme CCM et de la plate-forme OpenCCM<sup>1</sup>. Cette dernière est un projet OpenSource du consortium ObjectWeb hébergé par l'INRIA et dont le principal acteur est l'équipe Génie des Objets et composAnts Logiciels (GOAL) du Laboratoire d'Informatique Fondamentale de Lille (LIFL). Ici, la contribution principale des travaux présentés est la définition d'une architecture dédiée permettant l'intégration transparente de connecteurs (vue architecturale des interactions) au déploiement ou lors de l'exécution des applications métiers.

Le plan de cet article est le suivant. La section 2 identifie nos objectifs spécifiques. Ensuite, la section 3 présente l'architecture du support d'exécution ainsi que deux expérimentations sur le déploiement du connecteur. Finalement, la section 4 conclut l'article et donne quelques perspectives en rapport à l'expérimentation et à l'adaptabilité dans le CCM.

---

1. Disponible à l'adresse «<http://www.objectweb.org/openccm/index.html>».

## 2. Objectifs

La norme CCM spécifie le langage OMG IDL3 pour décrire les interfaces requises et fournies des composants, appelées *ports*. Ceux-ci utilisent une sémantique soit d'appels synchrones entre réceptacles et facettes soit événementielle entre sources et puits d'événements. Cependant, la norme impose l'utilisation du service de notification de CORBA (Notification Service [OMG 00]) pour le transport des événements. Ensuite, elle spécifie l'intégration des services, du cycle de vie et des composants au travers d'interfaces fournies par le conteneur et par la mise en œuvre du composant. Finalement, elle spécifie le déploiement d'assemblages de composants à partir d'archives contenant des versions binaires des composants et des descripteurs eXtended Markup Language (XML) contenant notamment une description des composants à instancier, leur configuration et leurs connexions initiales. Le but de nos travaux est d'offrir la possibilité d'adapter la technique de transport pour les événements, c'est-à-dire d'utiliser d'autres services que celui de notification. Cette étude se porte sur les événements plutôt que sur les appels bloquants car le cadre CORBA offre plus d'alternatives pour les techniques de transport dans le cas des événements. Plus précisément, nous dégageons six objectifs :

- **Transparence pour les composants métiers.** L'utilisation de connecteurs adaptables ne doit pas avoir d'impacts sur le cycle de développement standard des composants. En particulier, la gestion des opérations liées à la connectique doit être indépendante du connecteur utilisé ;

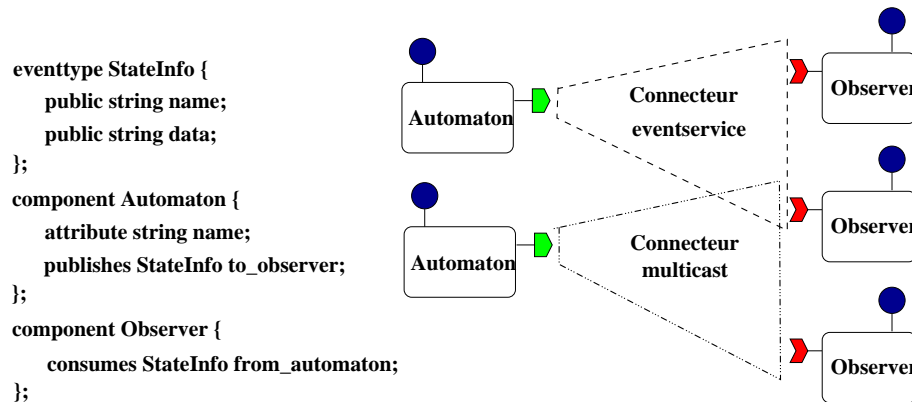
- **Abstraction des connecteurs.** De façon à permettre l'adaptation des connecteurs, il est nécessaire d'isoler celui-ci de la plate-forme. En particulier, cela impose la définition d'interfaces permettant la réalisation d'un grand nombre de connecteurs ;

- **Réalisation/prototypage de plusieurs connecteurs.** Dans le but de valider l'abstraction établie précédemment, nous nous imposons la mise en œuvre de connecteurs variés. Ceux-ci doivent reposer sur des sémantiques différentes (transport, groupe, etc) et des techniques différentes (par exemple TCP/IP et multicast pour le transport) ;

- **Programmation des connecteurs.** Nous souhaitons offrir la possibilité de programmer les connecteurs. De plus, une attention particulière doit être mise sur la simplicité de cette programmation ;

- **Interopérabilité des connecteurs.** L'utilisation de connecteurs adaptables doit rester possible en présence d'autres mises en œuvre du CCM ne supportant pas ces adaptations ;

- **Déploiement des connecteurs.** Dans le cadre du CCM, il est intéressant d'envisager un déploiement des connecteurs. Celui-ci doit pouvoir être réalisé à n'importe quel moment entre le déploiement de l'application et sa désinstallation.



**Figure 1.** *OMG IDL3 et exemple d'application déployée.*

### 3. Expérimentation

Pour illustrer l'expérimentation menée sur les connecteurs adaptables, nous allons utiliser l'exemple donné à la figure 1. Dans cet exemple, un ou plusieurs automates (composant `Automaton`) envoient des messages `StateInfo` reflétant leur état courant à un ou plusieurs observateurs (composant `Observer`). La figure 1 montre aussi une vue schématique d'une application déployée. Celle-ci utilise un connecteur `eventservice` et un connecteur `multicast` pour réaliser les liaisons entre les instances. Nous remarquons que ces connecteurs se chevauchent sur une des instances. Nous désirons en effet permettre à un port de participer à différents connecteurs en même temps. Nous allons maintenant présenter l'architecture mise en place ainsi que les deux expériences effectuées sur le déploiement du connecteur.

#### 3.1. Architecture

L'architecture du support d'exécution définit les entités suivantes :

- Les exécuteurs de ports réifient le mécanisme de connexion et participent ainsi à l'intégration transparente des connecteurs ;
- Les connecteurs de transport sont des extensions propriétaires et programmables de la plate-forme. Ils offrent une abstraction de la technique de transport des événements ;
- La configuration définit de manière descriptive l'information nécessaire au déploiement des connecteurs ;
- Le gestionnaire de connecteurs réalise le déploiement des connecteurs en instanciant la configuration. Pour cela, il définit un mécanisme générique de connexion

utilisé par les exécuteurs de ports ainsi qu'un mécanisme de propagation de la configuration.

### Exécuteurs de ports

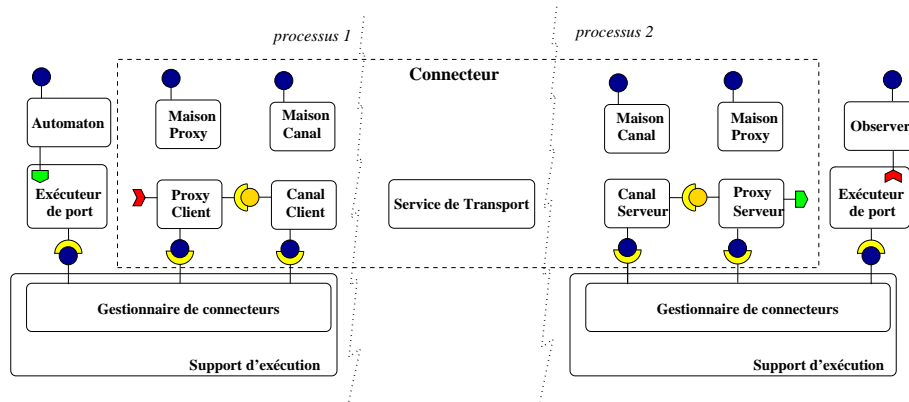
L'exécuteur de ports constitue le point de jonction entre le composant et le support d'exécution. En particulier, il implante les opérations non métier du port provenant de la projection de l'OMG IDL3 vers l'OMG IDL (pour plus de détails sur cette projection, voir [OMG 02c]). Pour réaliser cette implantation, l'exécuteur se base sur les opérations génériques fournies par le gestionnaire de connecteurs. En fait, l'exécuteur ne fait principalement qu'une traduction d'opérations typées en opérations génériques. En particulier, il est complètement indépendant des connecteurs instanciés pour le port. En ce sens, nous pouvons voir l'exécuteur comme l'artefact technique réalisant la réification du mécanisme de connexion.

### Connecteurs

Les connecteurs sont les extensions propriétaires (programmables) de la plateforme. Dans la suite, nous nous focalisons sur les connecteurs de transport. Par connecteur de transport, nous entendons les connecteurs encapsulant un service de transport (CORBA ou non) et permettant ainsi la réalisation d'appels inter-processus (ou inter-machines). Le connecteur de transport possède les caractéristiques suivantes :

- Il est constitué d'un canal et d'un proxy. Ceux-ci possèdent une vue cliente et une vue serveur. Le canal est l'abstraction du service de transport. Le proxy établit le lien entre l'interface typée de l'exécuteur et le canal. En particulier, du côté client, il convertit les appels typés en appels propriétaires sur le canal. Du côté serveur, il convertit les appels du canal en appels typés ;
- La liaison entre le canal et le proxy est non spécifiée et doit être si nécessaire définie par le fournisseur du connecteur ;
- Le fournisseur peut aussi définir de l'information contextuelle qui sera passée du canal serveur au canal client lors des connexions entre ports ;
- Le canal et le proxy sont créés à partir d'une maison. Celle-ci est instanciée à partir d'un point d'entrée (méthode statique en Java) localisé dans une archive chargée dynamiquement ;
- Du côté serveur, le canal et le proxy sont créés à l'instanciation du connecteur. Par contre, du côté client, seul le canal est créé à ce moment. Le proxy est créé lors de connexions avec le serveur.

La figure 2 présente un exemple de connecteur instancié entre le port `to_observer` du composant `Automaton` et le port `from_automaton` du composant `Observer`. L'instanciation de ce connecteur est présentée plus tard.



**Figure 2.** Exemple d'un connecteur de transport instancié.

### Configuration

Dans l'idée de se rapprocher du déploiement du CCM, nous avons opté pour une approche descriptive pour spécifier le déploiement des connecteurs. Celle-ci repose sur des configurations associées à chaque port du composant. Les différents éléments présents dans une configuration sont les suivants :

- La configuration initiale contient pour chaque connecteur la localisation physique de l'archive, les points d'entrées des maisons et de l'information propriétaire au connecteur ;
- Le déploiement statique référence les connecteurs à instancier lors de la création du composant auquel est rattaché le port ;
- Le déploiement dynamique définit les actions à effectuer lors des connexions du port. En particulier, il permet la propagation de configuration initiale de connecteurs.

L'exemple suivant illustre la configuration pour le port `to_observer` du composant `Automaton`.

```

1  "IDL:example/Automaton/to_observer:1.0"={
2      connectors={
3          static=[eventservice];
4          dynamic=[];
5
6          eventservice={
7              location=http://egmont.lifl.fr/eventservice.jar;
8              channel_home=EventServiceChannelHome.create_home;
9              proxy_home=EventServiceProxyHome.create_home;
10             address=corbaloc::localhost:8000/DefaultEventChannel;
11         };
12     };

```

```

13     synchronous={
14         location=http://egmont.lifl.fr/synchronous.jar;
15         channel_home=SynchronousChannelHome.create_home;
16         proxy_home=SynchronousProxyHome.create_home;
17     };
18
19     multicast={
20         location=http://egmont.lifl.fr/multicast.jar;
21         channel_home=MulticastChannelHome.create_home;
22         proxy_home=MulticastProxyHome.create_home;
23         ip=224.0.0.100;
24         port=9999;
25     };
26 };
27 };

```

Plus précisément, la configuration est structurée en une arborescence de propriétés. Celles-ci peuvent associer un identifiant à une valeur (ligne 7), à un ensemble de valeurs (ligne 3) ou à un ensemble de propriétés (ligne 2). Les identifiants et les valeurs sont des chaînes de caractères. Comme nous l'avons vu précédemment, un connecteur est constitué d'un canal et d'un proxy. Ceux-ci sont instanciés séparément à partir d'une maison. De plus, le code du canal, du proxy et de leur maison est contenu dans une archive chargée dynamiquement. La configuration initiale de chaque connecteur doit donc contenir la propriété `location` définissant la localisation physique de l'archive (ligne 7), les propriétés `channel_home` et `proxy_home` définissant respectivement le point d'entrée dans l'archive pour la maison du canal (ligne 8) et celle du proxy (ligne 9). De plus, cette configuration contient des informations propriétaires au connecteur et interprétables seulement par celui-ci (ligne 10, 23 et 24). En fait, ces informations servent à la configuration interne du canal. Ensuite, la configuration définit l'ensemble des connecteurs à instancier lors de la création du composant auquel est attaché le port. Ceci est réalisé grâce à la propriété `static` (ligne 3) qui référence les identifiants de configuration des connecteurs à instancier. De même, l'ensemble des connecteurs à instancier lors des connexions du port est défini au travers de la propriété `dynamic` (ligne 4). Nous détaillerons cette propriété à la section 3.3. Dans cet exemple, la configuration du port `to_observer` définit trois connecteurs :

- Un connecteur `eventservice` (ligne 6) basé sur le service d'événements de CORBA (Event Service [OMG 01]). Il requiert la propriété spécifique `address` utilisée par le canal pour trouver l'instance du service d'événements à utiliser pour le transport des événements ;
- Un connecteur `synchronous` (ligne 13) basé sur les appels CORBA synchrones et ne nécessitant pas d'informations particulières ;
- Un connecteur `multicast` (ligne 19) basé sur le plugin OCI multicast d'ORBacus [ION 01]. Il requiert les propriétés `ip` et `port` qui sont utilisées pour déterminer le groupe multicast à utiliser.

Pour ces trois connecteurs, l'archive contenant le code est téléchargée à partir d'un serveur Web. Ensuite, nous avons spécifié le déploiement du connecteur `eventservice` lors de la création du composant (ligne 3).

### Gestionnaire de connecteurs

Le gestionnaire de connecteurs est l'entité centrale du support d'exécution. En effet, il est responsable de la gestion du cycle de vie des connecteurs. Pour cela, il supporte les fonctionnalités suivantes :

- L'instanciation des connecteurs à partir de leur configuration ;
- La gestion des opérations de connexions ;
- La propagation de la configuration pour le déploiement dynamique (voir section 3.3).

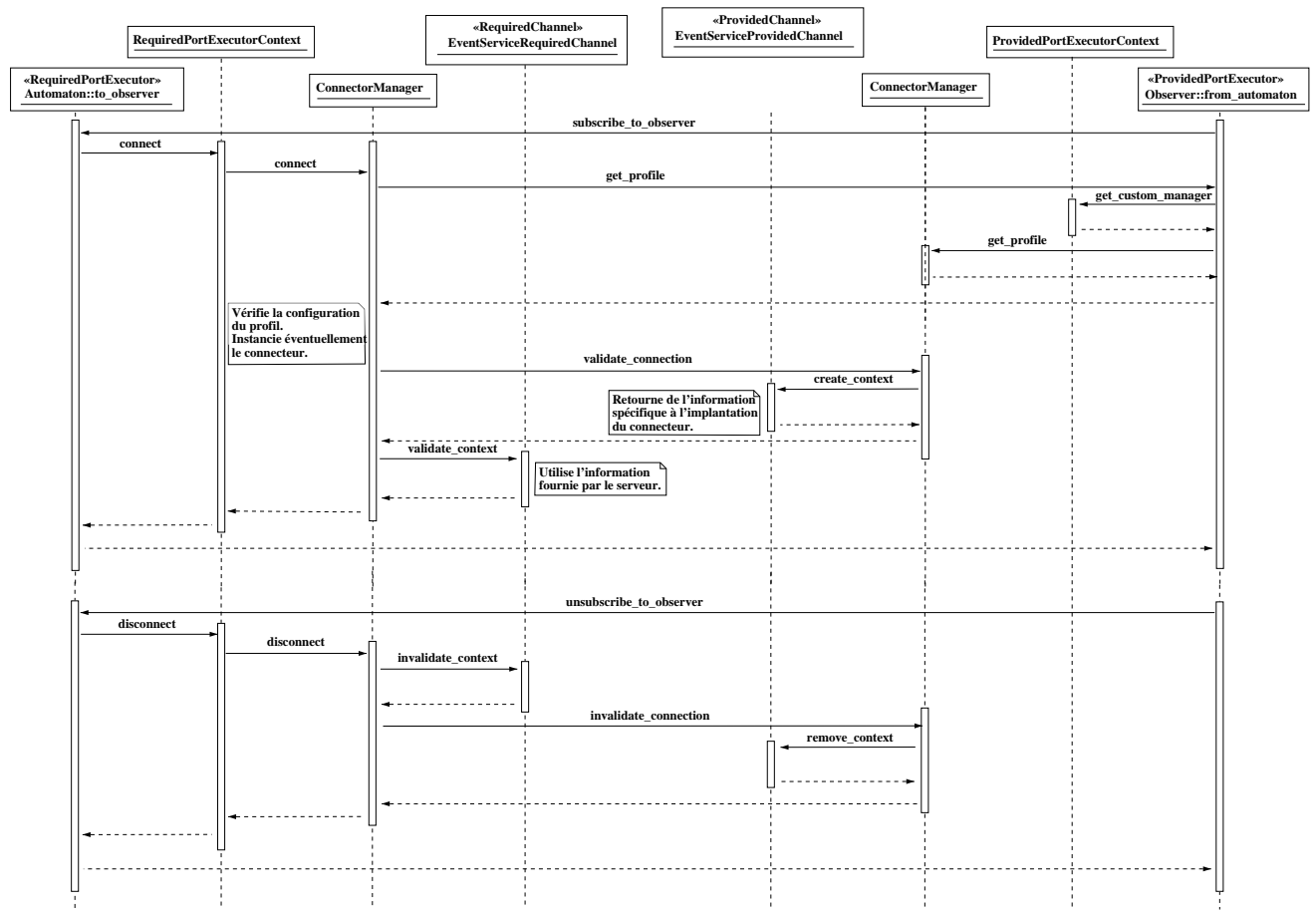
Plus précisément, l'instanciation du connecteur suit le schéma suivant. Tout d'abord, le gestionnaire récupère des instances de maisons à partir des points d'entrées définis dans la configuration. Ensuite, il vérifie le type de port. Du côté serveur (puits d'événements), il crée une instance de canal et de proxy. Du côté client (source d'événements), il ne crée que le canal. Puis, il configure le canal avec l'information propriétaire de la configuration et le notifie de la fin du processus. A partir de ce moment, les ports peuvent être connectés.

La figure 3 présente le diagramme de séquence d'une connexion et d'une déconnexion. Lors d'une connexion, le gestionnaire client récupère d'abord la configuration supportée par le serveur (opération `get_profile`). Il effectue ensuite la vérification par rapport à sa configuration locale et valide la connexion (opération `validate_connection`). Cette validation permet de récupérer de l'information propriétaire au connecteur et contextuelle à la connexion (opération `create_context`). Finalement, cette information est passée au canal client (opération `validate_context`). Lors d'une déconnexion, l'information contextuelle est invalidée du côté client (opération `invalidate_context`). Ensuite, la connexion est invalidée (opération `invalidate_connection`), ce qui permet de revenir à l'état précédent la connexion. En particulier, le canal côté serveur est notifié de la déconnexion (opération `remove_context`).

### 3.2. Déploiement statique

Dans un premier temps, nous souhaitons supporter la configuration et l'utilisation d'un connecteur reliant plusieurs ports de composants appartenant à la partie statique de l'application métier. Par exemple, nous voulons utiliser un connecteur basé sur le service d'événements de CORBA entre le port `to_observer` d'une instance du composant `Automaton` et le port `from_automaton` de deux instances du composant `Observer` s'exécutant sur des nœuds différents. Pour cela, l'approche utilisée est la suivante :

Figure 3. Diagramme de séquence d'une connexion et d'une déconnexion.



- Chaque port possède la même configuration initiale du connecteur `eventService`. De plus, chaque port référence cette propriété dans la propriété `static` de sa configuration ;

- Lors de la création du composant, le support d'exécution contacte le gestionnaire de connecteurs et lui demande pour chaque port d'instancier tous les connecteurs référencés dans la propriété `static` de leur configuration.

- Lors des connexions, le gestionnaire vérifie la consistance des configurations et crée le proxy client.

Le déploiement statique est une bonne solution pour spécifier un connecteur avant ou lors du déploiement de l'application. De plus, ce connecteur peut relier potentiellement un port de  $n$  instances d'un type de composant avec un port de  $m$  instances d'un autre type de composant. Cependant, ce type de déploiement requiert la présence d'une configuration consistante sur tous les ports et une connaissance *a priori* des instances de composants. Cette solution convient bien pour le déploiement de la partie statique de l'application métier où toutes les liaisons sont connues. Cependant, dans le cas de liaisons dynamiques, c'est-à-dire établies lorsque l'application est déjà déployée, elle suppose que les nouveaux composants soient eux aussi configurés. Pourtant, au moment de leur création, ceux-ci ne possèdent pas forcément cette information. Par exemple, si de nouvelles instances du composant `Observer` sont créées et que l'on souhaite les connecter à des instances préexistantes de composants `Automation`, il devient nécessaire de configurer le port du composant `Automation` seulement au moment de l'établissement de la connexion. De plus, en cas de reconfiguration du connecteur, il est nécessaire de reconfigurer tous les composants. Cette approche rend donc assez complexe l'administration des connecteurs.

### 3.3. Déploiement dynamique

L'objectif principal du déploiement dynamique est de pallier aux limites du déploiement statique et en particulier de permettre la découverte à l'exécution du connecteur à utiliser. L'approche est la suivante : lors de connexions entre composants, le gestionnaire de connecteurs vérifie du côté client l'adéquation entre les connecteurs déjà déployés sur le client et sur le serveur. Dans le cas où aucun connecteur identique n'est présent des deux côtés, le gestionnaire utilise l'information définie dans la propriété `dynamic` des configurations. Pour l'instant, deux actions complémentaires sont possibles :

- Propagation de la configuration initiale d'un ou plusieurs connecteurs vers l'autre composant ;

- Demande d'envoi de la configuration initiale d'un ou plusieurs connecteurs par l'autre composant.

L'exemple suivant illustre ces actions :

```
1 // Nous modifions l'exemple de la section précédente.
```

```

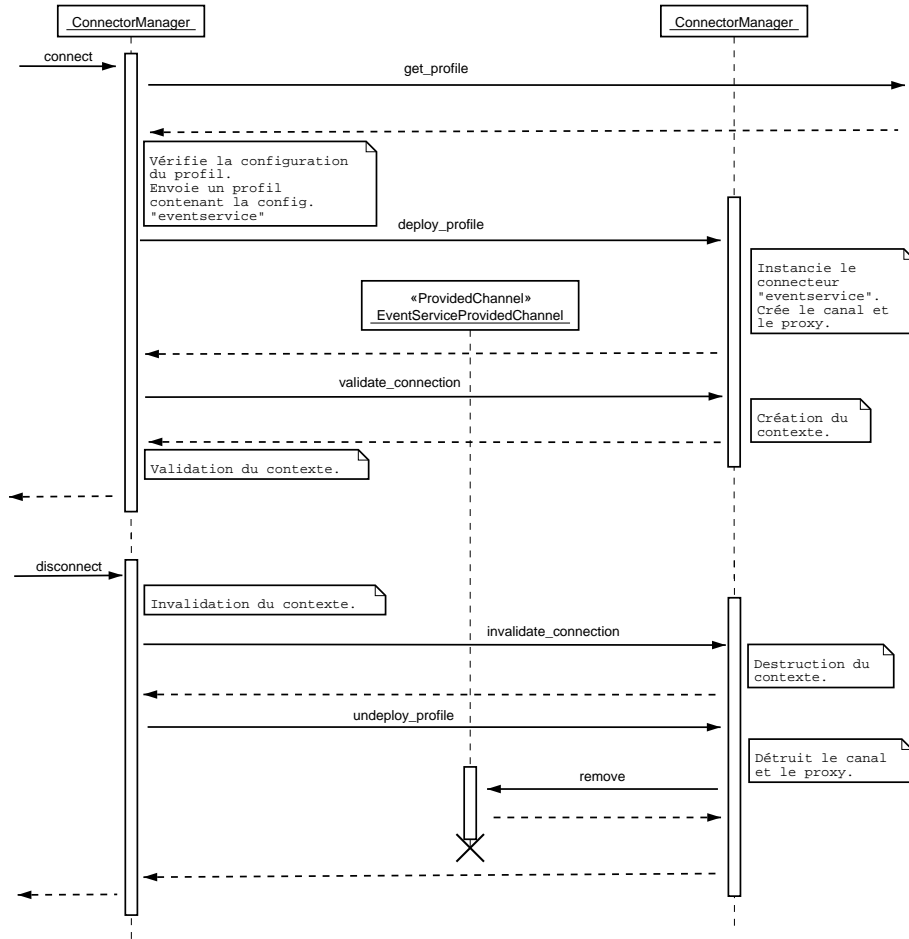
2  "IDL:example/Automaton/to_observer:1.0"={
3      connectors={
4          static=[eventservice];
5          // Spécifie la propagation de la configuration eventservice.
6          dynamic=[send:eventservice];
7          // Mêmes configurations initiales de connecteurs.
8      };
9  };
10
11 "IDL:example/Observer/from_automaton:1.0"={
12     connectors={
13         static=[];
14         // Nécessite la propagation du connecteur à utiliser.
15         dynamic=[remote];
16     };
17 };

```

Ici, le composant `Observer` ne définit pas de déploiement statique (ligne 13). De ce fait, il y a nécessairement propagation de configurations lors des connexions. Dans ce cas, le composant `Automaton` spécifie l'envoi de la configuration initiale du connecteur `eventservice` (ligne 6). Pour être consistant, le composant `Observer` doit spécifier le besoin d'une configuration (ligne 15).

Cette approche requiert donc un mécanisme de propagation de configurations. Toutefois, deux contraintes doivent être respectées. D'abord l'intégration de ce mécanisme ne doit pas perturber le mécanisme de connexion déjà mis en place et utilisé dans le cas du déploiement statique. Ensuite, ce mécanisme doit aussi permettre le retour à l'état précédent la connexion en cas de déconnexion. L'approche prise ici est d'ajouter un protocole de propagation de la configuration entre les gestionnaires de connecteurs. Celui-ci s'insère dans le protocole déjà présenté à la figure 3. La figure 4 présente la diagramme de séquence de cette propagation lors de la connexion et de la déconnexion du port `to_observer` du composant `Automaton` avec le port `from_automaton` du composant `Observer`. Ici, l'action `send :eventservice` utilise l'opération `deploy_profile` pour envoyer la configuration initiale du connecteur `eventservice` au gestionnaire du composant `Observer`. Ensuite, ce dernier instancie le connecteur de son côté. Finalement, le reste de la séquence de connexion est inchangée. De même, à la fin de la séquence de déconnexion, le gestionnaire du composant `Automaton` utilise l'opération `undeploy_profile` sur l'autre gestionnaire pour le notifier de la déconnexion. Ce dernier détruit donc le canal et le proxy.

Le déploiement dynamique du connecteur permet donc la découverte à l'exécution du connecteur à utiliser. Par exemple, il devient possible de ne spécifier la configuration initiale que sur une instance du composant `Automaton` puis de se reposer sur la propagation pour mettre en place le connecteur sur les instances du composant `Observer`. Cependant, ce mécanisme de propagation est limité et en particulier ne prend pas en compte les contraintes liées aux contextes d'exécution de chaque composant. Il serait en effet préférable de négocier le connecteur à utiliser ainsi que certaines



**Figure 4.** Propagation de la configuration pour le déploiement dynamique.

propriétés de sa configuration initiale de façon à satisfaire au mieux les contraintes de chaque composant. De plus, contrairement au déploiement statique, il n'est pas possible de définir un connecteur (n,m) en utilisant seulement ce mécanisme.

#### 4. Conclusion et perspectives

Dans cet article, nous avons présenté une architecture dédiée pour intégrer/adapter de façon transparente des connecteurs dans le CCM. Celle-ci se base sur la réification du mécanisme de connexion par l'exécuteur de ports pour intégrer les connecteurs. Ensuite, le gestionnaire de connecteurs réalise le mécanisme de connexion et instancie

les connecteurs en utilisant la configuration. Finalement, nous avons montré comment cette architecture permet le déploiement statique du connecteur, puis le déploiement dynamique en intégrant un protocole de propagation de la configuration. Ici, l'originalité de l'approche se situe sur la configuration, qui permet de décrire l'adaptation en dehors du support d'exécution, et sur le protocole de propagation, qui permet de déclencher automatiquement l'adaptation à distance.

A ce point de l'expérimentation, deux perspectives s'offrent à nous. Tout d'abord, cette expérience sur les connecteurs n'est pas terminée. Nous sommes partis sur une approche principalement pragmatique pour spécifier l'interface entre le connecteur et la plate-forme. De manière à stabiliser cette interface et à la valider concrètement, nous allons continuer la mise en œuvre de connecteurs diversifiés. Pour l'instant, des connecteurs basés sur des appels CORBA synchrones, sur le service d'événements de CORBA et sur le multicast sont réalisés. Ici, des connecteurs basés sur d'autres techniques asynchrones, comme CORBA Messaging [OMG 02b] ou Java Messaging Service (JMS) de SUN [SUN 02b] sont de futures expérimentations. Ceci nous amènera aussi à enrichir le protocole de propagation de la configuration pour arriver à définir un protocole de négociation de la configuration. De plus, le cadre CCM nous offre quelques directions prometteuses. Dans un premier temps, nous désirons intégrer le déploiement des connecteurs dans le déploiement standard du CCM. L'idée est de réutiliser les mêmes formalismes XML de façon à pouvoir se servir des outils fournis par OpenCCM. Dans un deuxième temps, nous remonterons la configuration des connecteurs vers les outils de modélisation. L'idée est de s'engager dans une approche Model Driven Architecture (MDA) dans laquelle les connecteurs sont modélisés dans le Platform Specific Model (PSM) associé à OpenCCM.

Ensuite, nous allons étudier d'autres points d'adaptabilité relatifs au CCM. Nous comptons offrir la possibilité de définir de nouveaux cycles de vie ou d'enrichir ceux déjà spécifiés. Nous allons aussi intégrer les travaux déjà réalisés sur l'interception des appels métiers [VAD 01]. Une optique plus transverse sera l'étude et l'intégration d'un support pour la supervision et l'administration. Celles-ci devront cibler à la fois le conteneur et l'application métier. Finalement, la problématique du tissage de ces points d'adaptabilité sera à résoudre tout au long des expérimentations.

## 5. Bibliographie

- [BRU 00] BRUNETON E., RIVEILL M., « JavaPod : une plate-forme à composants adaptable et extensible », rapport n° 3850, Projet SIRAC, Unité de Recherche INRIA Rhône-Alpes, Montbonnot-St-Martin, Rhône-Alpes, France, Janvier 2000.
- [ECM 00] ECMA, « Common Language Infrastructure (CLI) », ECMA, ECMA/TC39/TG3/2000/2 part 1, 2, 3, 4 and 5, Décembre 2000.
- [ION 01] IONA TECHNOLOGIES I., « ORBacus for C++ and Java », Documentation, IONA Technologies, Inc., Version 4.1.0, 2001.
- [JTC 95] JTC1/SC21/WG7 I., « Reference Model of Open Distributed Processing Part 1, 2, 3 and 4 », ISO/IEC, SC21 N8926rev, Juin 1995.

- [KIC 91] KICZALES G., DES RIVIÈRES J., BOBROW D. G., *The Art of the Metaobject Protocol*, MIT Press, Septembre 1991, ISBN : 0-26-261074-4.
- [KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., VIDEIRA LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », dans 11<sup>th</sup> *European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, Juin 1997.
- [KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. G., « An Overview of AspectJ », dans 15<sup>th</sup> *European Conference on Object-Oriented Programming (ECOOP'01)*, Budapest, Hongrie, Juin 2001.
- [KRA 02] KRAKOWIAK S., « Jonathan Tutorial », Objectweb, « <http://www.objectweb.org/jonathan/doc/tutorial/index.html> », Juin 2002.
- [OMG 00] OBJECT MANAGEMENT GROUP, « Notification Service », OMG Document formal/00-06-20, Juin 2000.
- [OMG 01] OBJECT MANAGEMENT GROUP, « Event Service », OMG Document formal/01-03-01, Mars 2001.
- [OMG 02a] OBJECT MANAGEMENT GROUP, « CORBA Components Specification », OMG Document formal/02-06-65, Juin 2002.
- [OMG 02b] OBJECT MANAGEMENT GROUP, « Common Object Request Broker Architecture », OMG Document formal/02-06-01, Juin 2002.
- [OMG 02c] OMG CCM IMPLEMENTERS GROUP, « CORBA Component Model Tutorial », rapport, Object Management Group, OMG Meeting, Orlando, USA, OMG TC Document ccm/2002-06-01, Juin 2002.
- [PAW 01] PAWLAK R., SEINTURIER L., DUCHIEN L., FLORIN G., « JAC : A Flexible and Efficient Solution for Aspect-Oriented Programming in Java », dans *The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'01)*, Kyoto, Japon, Septembre 2001.
- [RAM 02] RAMMER I., *Advanced .NET Remoting*, The Expert's Voice, Apress, Berkeley CA, 2002, ISBN 1-59059-025-2.
- [SUN 02a] SUN MICROSYSTEMS, « Enterprise JavaBeans (EJB) », SUN Microsystems, Specification Final Release 2.0, 2002.
- [SUN 02b] SUN MICROSYSTEMS, « Java Messaging Service (JMS) », SUN Microsystems, Specification Final Release 1.1, Mars 2002.
- [VAD 01] VADET M., MERLE P., « Les conteneurs ouverts dans les plates-formes à composants », dans *Actes des Journées Composants 2001 : « Flexibilité du système au langage »*, p. 33 - 46, Besançon, France, Octobre 2001.