



Université Joseph Fourier
U.F.R. Informatiques
Mathématiques Appliquées



I.M.A.G.

DEA D'INFORMATIQUE :

SYSTEMES ET COMMUNICATIONS

Projet présenté par :
Noël De Palma

Reconfiguration Dynamique D'applications Réparties

Effectué au laboratoire : SIRAC

Date : 12 juin 1998
Jury : S. Krakowiak
D. Bert
Y. Rouzeau
L. Bellissard

Table de Matières

<i>Introduction</i>	<i>1</i>
1 Le projet SIRAC	2
1.1 Support Système pour Serveurs d'Information	2
1.2 Gestion de la mobilité dans l'internet	2
1.3 Construction d'applications réparties	2
2 Problématique	3

Chapitre I. Etat de l'Art	7
1 Introduction	8
2 Les différents types de changement	8
2.1 Les changements de structure	8
2.2 Les changements géométriques	9
2.3 Les changements de mise en œuvre	9
2.4 Les changements d'interfaces	10
2.5 Remarques	10
3 Les problèmes généraux	11
3.1 Introduction	11
3.2 Problème de modification des références des services	12
3.3 Problème des messages en transit dans les canaux de communication	13
3.4 Problème de la préservation de l'état du serveur <i>s/l</i>	13
3.5 Problème de la validation des changements	14
4 Les différentes approches	14
4.1 CONIC	15
4.1.1 Présentation	15
4.1.2 Architecture	16
4.1.3 Exemple : Une application annuaire	16
4.1.4 Type de changement supportés	18
4.1.5 Définition des états abstraits	19
4.1.6 Les conditions associées aux primitives de reconfiguration	20
4.1.7 L'algorithme de changement	20
4.1.8 Extension de l'algorithme aux sessions dépendantes	22
4.1.9 Avantages et inconvénients	22
4.2 POLYLITH	22
4.2.1 Présentation	22
4.2.2 Architecture	24
4.2.3 Exemple : l'application Annuaire	24
4.2.4 Type de changements supportés	25
4.2.5 Préservation de l'état	25
4.2.6 Algorithme de reconfiguration	26
4.2.7 Avantages et inconvénients	26
4.3 ARGUS	27
4.3.1 Présentation	27
4.3.2 Exemple : l'application Annuaire	28
4.3.3 Types de changement supportés	28
4.3.4 Algorithme de reconfiguration	28
4.3.5 Avantages et Inconvénients	29
4.4 Les autres pistes possibles	29
4.4.1 L'algorithme du snapshot	29
4.4.2 La migration de processus	30
4.4.3 La liaison dynamique	31
5 Discussions	31

Chapitre II. Un contexte pour la reconfiguration	33
1 Le modèle A3	34
1.1 Le modèle événement-réaction	34
1.2 Présentation du modèle A3	34
1.2.1 Concept de base	34
1.2.2 Propriétés	35
1.3 Mise en œuvre	36
1.3.1 Les agents A3	36
1.3.2 Le bus logiciel A3	36
1.3.3 L'agent Factory	36
1.4 L'application annuaire	37
2 Le modèle OLAN	38
2.1 Introduction	38
2.2 Présentation du modèle OLAN	39
2.3 Le langage OCL	39
2.3.1 Intégration de logiciel	40
2.3.2 Définition de l'Architecture	41
2.3.3 Répartition	44
2.4 Le support de configuration OLAN	45
2.4.1 Présentation	45
2.4.2 Installation d'une application	46
2.4.3 Architecture	47
2.5 Avantages et inconvénients de l'environnement A3-OLAN	47
2.5.1 Avantages	47
2.5.2 Inconvénients	48
3 Conclusion	48

Table des Matières

Chapitre III. Mécanisme de Reconfiguration dans l'Environnement A3	51
1 Introduction	52
2 Primitives de changements proposées	53
2.1 Les primitives de changements de structures	53
2.1.1 Les primitives AddAgent et BindAgent	53
2.1.2 La primitive RebindAgent	54
2.1.3 La primitive CloneAgent	55
2.1.4 La primitive RemoveAgent	56
2.2 Les primitives de changements de géométrie	56
2.2.1 La primitive MoveAgent	56
3 Gestion de la cohérence de l'application	57
3.1 Etat abstrait d'un agent	57
3.2 Transitions	58
3.3 Algorithme <i>Passivate</i>	59
3.4 L'algorithme de reconfiguration	62
3.4.1 Preuve de l'algorithme	63
3.4.2 Propriétés associées à l'algorithme de reconfiguration	64
4 Discussions	64
4.1 Evaluation de l'algorithme	64
4.2 Performance de l'algorithme	66

Table de Matières

Chapitre IV. Support pour la Reconfiguration dans l'Environnement A3	70
1 Introduction	71
2 Architecture générale	72
3 Hiérarchie des classes d'agent utilisées	72
3.1 La classe Agent	73
3.2 La classe Agent Configurable	73
3.3 La classe Agent Interconnectable	74
3.4 La classe Agent Reconfigurable	75
4 L'outil graphique de configuration	75
5 L'Agent configurateur	76
5.1 Analyse des services à fournir	76
5.1.1 Le service de configuration d'une application	77
5.1.2 Le service de reconfiguration	78
5.2 Dialogue Client-Configurateur	79
5.2.1 Accès au configurateur	79
5.2.2 Protocole de communication	79
5.3 Disponibilité des services de configuration	80
5.4 Architecture détaillée du configurateur	82
5.4.1 Structure d'exécution du configurateur	82
5.4.2 Structures de données du configurateur	83
5.5 Les managers	84
6 Conclusion	85

Table des Matières

<i>Chapitre V. Conclusion</i>	87
1 Problématique	88
2 Contexte	88
3 Contribution	89
3.1 Description d'un algorithme de reconfiguration	89
3.2 Mise en œuvre de l'algorithme de reconfiguration	89
3.3 Conclusions générales	90
4 Perspectives	90

Table de Matières

<i>Bibliographie</i>	92
----------------------	-----------

Introduction

Introduction

1 Le projet SIRAC

Ce stage de DEA s'est effectué au sein de l'équipe **SIRAC** (Système Informatiques Répartie pour Application Coopératives) qui est :

- Un projet **INRIA** regroupant l'**UJF**, l'**INPG** et l'Université de Savoie.
- Une unité propre de l'enseignement supérieur associé à l'**UJF**.

Ce projet mène des recherches selon trois axes directeurs :

1.1 Support Système pour Serveurs d'Information

L'objectif de cet axe est de fournir des services génériques et efficaces pour la construction de serveurs d'information extensibles, sur des grappes de machines homogènes. Deux voies sont explorées :

- *Service de gestion mémoire.* L'objectif est de fournir des services qui permettent de partager des données pour la réalisation de serveurs d'information (serveur de fichiers, serveur web, serveur de base de données).
- *Service systèmes pour réseaux de communication rapides.* L'objectif de ce thème est d'exploiter le potentiel des réseaux rapides pour accroître les performances des applications sur des serveurs en grappes. La technologie utilisée est la technologie d'interconnexion SCI.

1.2 Gestion de la mobilité dans l'internet

L'objectif est de fournir des protocoles et des services systèmes pour faciliter l'accès à internet à partir de postes mobiles. Cette activité vise à fournir des mécanismes permettant de supporter simultanément plusieurs interfaces de réseau sur une station mobile dans le but d'utiliser le réseau le plus adapté aux caractéristiques du flot de données de chaque application.

1.3 Construction d'applications réparties

L'objectif est de fournir des outils et services pour le développement et l'exécution d'applications réparties. Deux directions sont actuellement explorées :

- *Intégration et extension d'applications existantes.* Ce thème vise à fournir des outils permettant de construire des applications réparties par assemblage de composants existant et de les déployer sur des plateformes distantes.
- *Application répartie sur internet.* L'objectif de ce thème est de fournir des services systèmes pour permettre d'utiliser efficacement Internet comme environnement d'exécution d'applications réparties. Les services visés en

priorité concernant la gestion d'objets partagés, la gestion de la sécurité et la gestion de l'exécution répartie (migration du code et des données).

Les travaux de DEA se sont effectués au sein du dernier thème, **Application répartie sur internet** dans l'équipe « Intégration et extension d'applications réparties ». Le sujet de stage est la reconfiguration dynamique d'applications réparties dont la problématique est définie ci-dessous.

2 Problématique

Avec le développement des réseaux et la délocalisation de plus en plus importante des entreprises, la mise en œuvre d'applications réparties s'est imposée comme étant une solution permettant la coopération des différents acteurs constituant de l'entreprise. C'est dans le but de réduire les coûts de construction d'applications réparties que des outils de développement de plus en plus perfectionnés ont été conçus.

De nos jours, de plus en plus d'applications réparties s'exécutent vingt quatre heures sur vingt quatre, sept jours sur sept. L'arrêt de ce type d'application dans le but de l'améliorer ou tout simplement d'effectuer des actions de maintenance coûte des sommes d'argent importantes. Ces coûts sont liés :

- A l'arrêt de l'application en cours d'exécution pour effectuer les opérations de maintenance.
- A la réinstallation de la nouvelle version de l'application.
- A la difficulté de faire évoluer ce type d'application.

Une solution à cette problématique est de fournir des mécanismes permettant l'évolution ou la modification d'une application pendant l'exécution *sans l'arrêter*.

La reconfiguration dynamique d'applications réparties[18] peut se définir comme l'ensemble des changements apportés à une application répartie en cours d'exécution. Les changements sont de différentes sortes :

- La modification de l'architecture d'une application (ajout, retrait de modules et de leur liaison).
- La modification de la distribution géographique d'une application.
- Modification de la mise en œuvre d'un composant.
- Modification des interfaces des services.

Nous considérons une classe d'application répartie décrite comme un assemblage d'agents interconnectés par un bus logiciel qui met œuvre un modèle de communication par événement. Cette description est effectuée à l'aide d'un langage de description d'architecture [4][5] permettant de séparer l'assemblage des agents de leur programmation. Un tel langage permet de construire une application avec une vision macroscopique des composants logiciels requis et de leur manière d'interagir et de communiquer dans un environnement réparti. Une

telle approche est souvent appelée « programmation dans le grand » par opposition à la programmation individuelle des composants « programmation dans le petit »[12].

Nous pensons que l'utilisation d'un modèle basé sur des agents homogènes interconnectés et d'un langage de description d'architecture de haut niveau facilite la spécification et la mise en œuvre d'un mécanisme de reconfiguration.

Pour illustrer les avantages de ce modèle pour la reconfiguration, nous décrivons brièvement une application réelle : un logiciel de pare-feu (firewall) NETWALL[25].

Un pare-feu est un programme qui a pour rôle de filtrer des informations échangées entre des réseaux. Chaque paquet à destination du site protégé par le pare-feu est intercepté et analysé. A partir de cette analyse, le paquet est détruit ou retransmis vers le destinataire et cette décision est enregistrée. L'analyse des paquets est conduite par un ensemble de règles défini par l'administrateur du pare-feu. Ces règles peuvent être identiques pour tous les paquets ou spécifiques à une application.

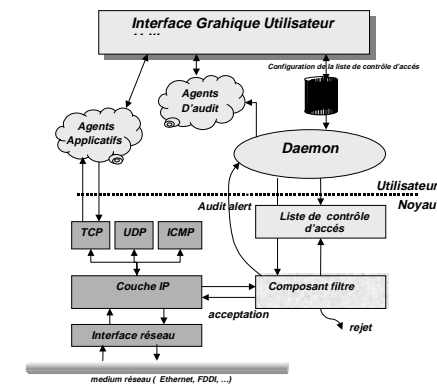


Figure 1. Architecture d'une application de NETWALL

Le mécanisme de reconfiguration dynamique permet d'ajouter ou d'adapter des fonctionnalités existantes en ajoutant/supprimant des agents et des interconnexions.

Des exemples de reconfigurations sont :

- L'adaptation des vérifications à effectuer sur le trafic réseau en ajoutant/supprimant des agents de filtrages.
- La redéfinition des méthodes d'alertes. Par exemple, les alertes peuvent être dirigées vers une console ou une adresse électronique.
- L'ajout de fonctionnalité d'archivage sur les informations stockées par le firewall grâce à l'ajout d'un agent de compression.

L'objectif du stage est de fournir des mécanismes de reconfiguration dynamique dans le cadre d'applications réparties construites à base d'agents communicants grâce à un bus logiciel. Ce travail a pour but de :

- Développer une approche générale permettant de résoudre ces changements.
- Construire des outils systèmes permettant de supporter ces changements.

Le rapport s'organise en quatre chapitres :

- **Le chapitre 1** définit plus précisément la notion de **reconfiguration dynamique** et présente les projets les plus trouvés dans la littérature. A la fin de ce chapitre, le lecteur aura une définition précise de **la reconfiguration dynamique**, une connaissance générale des problèmes intervenants lors de la reconfiguration d'une application et une vue d'ensemble des solutions apportées par différents projets de recherche.
- **Le chapitre 2** présente plus précisément le contexte de travail à partir duquel les mécanismes de reconfiguration dynamique seront développés. Il s'agit de l'environnement d'exécution **AAA** et du modèle de construction d'application **OLAN**.
- **Le chapitre 3** présente notre algorithme de reconfiguration et compare notre approche à celles étudiées dans l'état de l'art du **chapitre 1**.
- **Le chapitre 4** détaille les outils et l'architecture logiciel mis en place pour supporter la reconfiguration dynamique dans l'environnement **AAA**.

Chapitre I.

Etat de l'Art

1 Introduction

Cette section présente différentes approches apportant des solutions au problème de la reconfiguration dynamique d'application répartie. Dans ce cadre, nous définissons une application répartie comme étant un ensemble de composants interconnectés au niveau d'un réseau de communication. Un composant est une entité logicielle qui encapsule des services et qui les rend accessibles au travers d'une interface.

Pour ce type d'application, les mécanismes de reconfiguration dynamique peuvent permettre de modifier une application répartie en terme :

- D'ajouts et de retraits de composants ainsi que de modifications de leurs interconnexions afin d'adapter les fonctionnalités de l'application.
- De modification de la distribution géographique d'une application, pour permettre, par exemple, le déplacement d'une application répartie sur des machines plus performantes.
- De modification de la mise en œuvre d'un composant dans le but de modifier le traitement effectué lors d'un service spécifique par un composant.
- De modification des interfaces des services d'un composant. Ceci permet de faire évoluer l'interface d'accès à un service.

Ensuite, nous présentons une tentative de classement des différents changements[15], les problèmes généraux associés à la problématique et enfin les solutions de *CONIC*[17][18][19], *POLYLITH*[15][16] et *ARGUS*[20] qui représentent les différentes grandes catégories de solutions pour la reconfiguration dynamique d'application répartie.

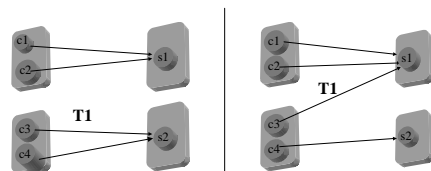
Pour finir, nous prendrons du recul par rapport à ces solutions pour essayer de tirer des conclusions pouvant nous aider lors de la définition de notre algorithme.

2 Les différents types de changement

2.1 Les changements de structure

Nous définissons la configuration structurelle d'une application répartie comme la description de l'ensemble des composants et de leurs interconnexions.

Les changements de structure modifient la configuration structurelle d'une application répartie. Les changements structuraux de base sont : l'ajout, le retrait de composant et la modification des liaisons entre composants. La figure suivante montre une utilisation possible d'un changement de structure dans le but d'effectuer une répartition de charge. Le changement de structure effectué est une modification de la connexion **T1** pour rediriger les requêtes en provenance du composant **c1** vers le composant **s1**.



c1, c2, c3, c4: editeur
s1, s2 : Serveur d'impression

Figure I-1 : Un exemple de changement structure : redirection de la connexion T1 vers le composant s1.

2.2 Les changements géométriques

Les changements géométriques altèrent uniquement le placement des composants en terme de site d'exécution. Intuitivement ce type de changement ressemble au problème de la migration de processus[26][24] si l'on assimile un composant à un processus.

La figure suivante illustre un exemple de changement géométrique. Le composant « quatre » qui s'exécutait sur le **site 2** va continuer à s'exécuter sur le **site 3** sans perte de cohérence pour l'application. Ce type de changement ne modifie pas la *configuration structurelle* de l'application, i.e. il n'y a aucune modification de la structure de l'application et de la topologie des interconnexions.

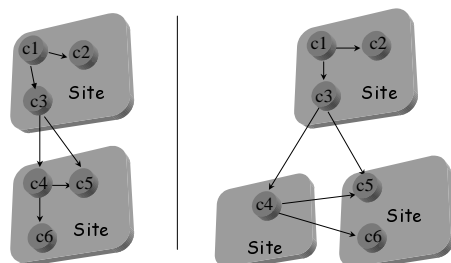


Figure I-2 : Exemple de changement géométrique, le composant c4 et déplacé du site 2 vers le site 3

2.3 Les changements de mise en œuvre

Les changements d'implémentation ne changent ni la structure de l'application ni le placement, ni l'interconnexion des composants mais modifient le code des composants et éventuellement restructurent leurs données sans modifier leur interface.

La restructuration de donnée[15] revient à modifier le type d'une donnée et à effectuer une traduction de la valeur de la donnée de l'ancien type vers le nouveau ; par exemple, une variable de type entier et de valeur 1 pourrait se restructurer en une valeur de type chaîne et de valeur « un ».

Ce type de changement est essentiellement lié au problème de l'évolution des applications car les changements d'implémentation sont utilisés dans un but d'amélioration. De manière générale, les mécanismes d'évolution d'application sont souvent *statiques*, c'est à dire que l'application doit être arrêtée, modifiée puis relancée.

Néanmoins, nous pensons que ce type de changement est lié à la reconfiguration si le changement d'implémentation s'effectue à l'exécution. Il serait très intéressant de proposer des mécanismes d'évolution d'applications en cours d'exécution, ce type de changement se trouvant à la frontière entre l'évolution des applications et la reconfiguration dynamique des applications.

2.4 Les changements d'interfaces

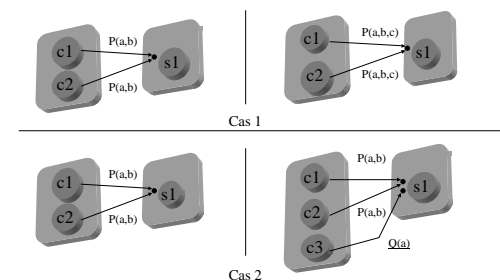


Figure I-3 : Exemple de changements d'interface

Le changement d'interface[15][23] est défini comme la modification de l'ensemble des services fournis par un composant (cas 2) ainsi que la modification de la signature d'un service (cas 1). La signature d'un service correspond à son nom, à son nombre de paramètres et à leur type. La figure suivante illustre ces deux cas.

Dans le premier cas, nous pouvons remarquer que la signature de la méthode « P » a changé, le nombre de paramètres a été modifié. Dans le deuxième cas, un service a été ajouté à l'ensemble des services fournis par le composant. Les changements d'interface posent des problèmes de gestion de version. En effet, les clients d'un composant ne doivent pas être perturbés par le changement d'interface des services du composant. Est-il possible de faire coexister des clients qui accédaient à l'ancienne interface d'un service et des clients qui accèdent à la nouvelle interface du service ?

2.5 Remarques

Le lecteur avisé pourra remarquer que les différents types de changement peuvent s'effectuer à partir des changements de structure. En effet, les changements de géométrie, de mise en œuvre et d'interface peuvent s'effectuer à partir d'un ajout, d'un retrait de composant et d'ajouts, de retrait de connexions. Cela implique qu'un algorithme qui résout les changements de structure peut aussi résoudre les autres types de changement. Néanmoins, le

fait de restreindre les primitives de reconfiguration au changement de structure va complexifier le processus de validation. De plus, il est peut être possible d'optimiser certaines primitives de reconfiguration.

3 Les problèmes généraux

3.1 Introduction

La reconfiguration dynamique soulève de nombreux problèmes inhérents aux classes d'application ou au modèle de communication. Cela provient de la difficulté à généraliser un algorithme de reconfiguration d'un modèle à un autre. Par exemple :

- Les applications temps réels ont des contraintes d'exécutions très fortes, des scénarios de reconfiguration très spécifiques. Elles fonctionnent parfois avec des ressources restreintes. Par exemple, dans le cadre d'une application de vidéo conférence, la contrainte principale des mécanismes de reconfiguration est la vitesse de reconfiguration. Ce type d'application peut tolérer une perte de qualité sur l'image. Par conséquent, l'algorithme de reconfiguration peut éventuellement entraîner la perte de certains messages pour permettre une reconfiguration plus rapide.
- Certains modèles de communication, comme par exemple les modèles à objets répartis partagés[1] induisent des contraintes de cohérences supplémentaires[21].
Ainsi, dans le cadre du changement de la mise en œuvre d'un objet, l'algorithme de reconfiguration doit prendre en compte le fait que l'objet à reconfigurer peut exister en n exemplaires locaux cohérents entre eux.

Néanmoins, il est possible de mettre en évidence un certain nombre de problèmes généraux. Nous illustrerons nos propos à l'aide du scénario suivant :

La figure suivante montre une trace d'exécution possible de l'application répartie.

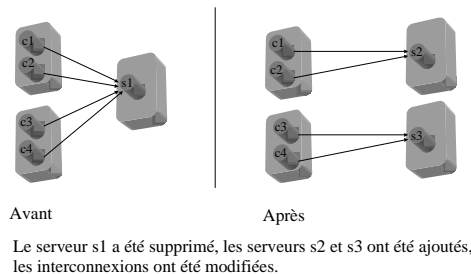


Figure I-4 : scénario illustrant les problèmes liés à la reconfiguration.

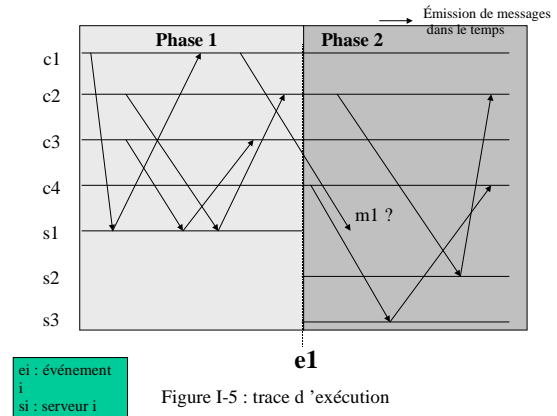


Figure I-5 : trace d'exécution

Ce diagramme temporel peut être divisé en plusieurs phases :

□ Phase 1

La *phase 1* correspond à la trace d'exécution antérieure à l'événement *e1*.

Les clients *c1*, *c2*, *c3* effectuent des requêtes vers le serveur *s1*. A la fin de la phase 1, seul le message *m1* n'est pas encore arrivé à destination, il est encore en transit au début de la *phase 2*.

□ Phase 2

La *phase 2* correspond à la trace d'exécution postérieure à l'événement *e1*. Cet événement marque le début du scénario de reconfiguration suivant :

- Suppression du serveur *s1*.
- Ajout des serveurs *s2* et *s3*.
- Modification des interconnexions entre les clients et les serveurs.

Cette trace d'exécution nous permet de mettre en évidence plusieurs problèmes qui sont explicités dans la suite.

3.2 Problème de modification des références des services

Pour qu'un client *C1* puisse communiquer avec un serveur *S1*, il faut que le client possède une référence vers le serveur *S1*.

Lors d'une reconfiguration, cette référence peut être amenée à changer pour permettre, par exemple, au client de s'adresser à un autre serveur. Dans le cadre de la reconfiguration, il faut avoir un mécanisme permettant de modifier ces références en cours d'exécution.

- Dans le cas de la figure précédente:

Une fois passé l'événement $e1$:

- Les interconnexions entre $c1$, $c2$, $c3$, $c4$ et $s1$ ont été *altérées*.

Les clients $c1$ et $c2$ doivent adresser leurs requêtes au serveur $s2$. Il en est de même pour $c3$ et $c4$ qui s'adressent à $s3$. Les anciennes références vers $s1$ contenues dans $c1$, $c2$, $c3$, $c4$ doivent être mises à jour pour référencer le serveur $s2$ pour $c1, c2$ ou le serveur $s3$ pour $c3$ et $c4$.

L'algorithme de reconfiguration doit pouvoir modifier des références à distance.

3.3 Problème des messages en transit dans les canaux de communication

- Lors d'une reconfiguration, des messages peuvent être en transit dans les canaux de communication. Les messages en transit ne doivent pas être perdus.
- Ainsi, dans l'exemple précédent, le message $m1$ est envoyé lors de la *phase 1*. Ce message ne doit pas être perdu malgré le fait qu'il arrive à destination alors que le serveur $s1$ a déjà été supprimé.

L'algorithme de reconfiguration doit assurer qu'aucun message n'est perdu lors d'une reconfiguration et que l'ordre d'arrivée des messages n'est pas altéré.

3.4 Problème de la préservation de l'état du serveur $s1$

Nous appelons *état du serveur $s1$* l'ensemble des informations internes à $s1$ nécessaires à son bon fonctionnement. Dans le cadre de la reconfiguration, l'état d'un serveur est constitué de l'ensemble de ses données.

- Le serveur $s1$ peut être en cours d'exécution lorsque survient une reconfiguration.

Il n'est pas tolérable que l'état de $s1$ disparaisse avec la suppression de $s1$ (Par exemple les données contenues dans l'état de $s1$ sont nécessaires à la bonne continuation du service).

L'état du serveur $s1$ doit être transféré aux serveurs $s2$ et $s3$ pour que ces derniers puissent commencer leur exécution dans un état cohérent par rapport au reste de l'application.

L'algorithme de reconfiguration doit pouvoir sauver et restaurer l'état d'un composant à distance.

3.5 Problème de la validation des changements

Lors d'une reconfiguration, il faut s'assurer que l'exécution de l'application reste cohérente. L'exécution d'une application est cohérente si la continuation de l'application après une reconfiguration est correcte par rapport à l'exécution de l'application avant la reconfiguration et aux modifications dues à la reconfiguration.

Par exemple, lors d'un changement géométrique, il faut s'assurer que les nouveaux sites d'hébergement des serveurs disposent des ressources nécessaires à leur exécution. Il faut également, lors d'un changement de structure cette fois, vérifier que l'on interconnecte bien des entités compatibles. Par exemple : le service proposé correspond - t - il au service requis ?

4 Les différentes approches

L'étude bibliographique a permis de mettre en évidence certaines approches intéressantes des recherches sur la reconfiguration dynamique d'application répartie.

- La première approche se base sur une abstraction de l'état réel des composants participants à une application. Elle est développée par le projet *CONIC*[17][18] qui fournit un environnement de programmation d'applications réparties construites à partir de composants interconnectés.
- La deuxième met en œuvre des mécanismes de reconfiguration sur un bus logiciel, i.e. au niveau du système de communication entre composants. Cette approche est développée dans le projet *POLYLITH*.
Elle diffère de la première approche dans le sens où le programmeur inclut dans la mise en œuvre d'un composant un certain nombre d'informations permettant de le reconfigurer.
- La dernière approche se base sur les mécanismes de tolérances aux fautes. Elle fut utilisée initialement par le projet *ARGUS*[20] qui propose un système d'exploitation permettant de reconfigurer une application répartie.

Cette dernière approche diffère des deux précédentes par le fait que les mécanismes de reconfiguration font partie d'un système d'exploitation assurant des propriétés transactionnelles.

De nombreux travaux ont été effectués sur la reconfiguration dynamique. Nous pensons que ces trois approches constituent les trois courants de base des recherches dans la reconfiguration dynamique. Nos recherches ont été fortement influencées par les projets *CONIC* et *POLYLITH*. Par la suite, nous détaillerons plus précisément ces deux projets. Le projet *ARGUS* ainsi que des mécanismes (comme par exemple les techniques de *chargement*

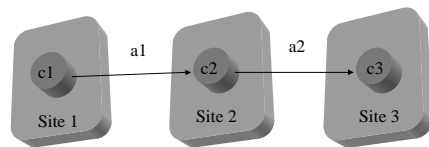
dynamique de code) qui pourraient avoir un intérêt pour la reconfiguration seront abordés en dernier.

4.1 CONIC

4.1.1 Présentation

CONIC a été développé au Distributed Software Engineering Group de l'Impérial College. CONIC permet la programmation constructive d'applications réparties utilisant un *modèle composant* et un *langage de configuration*. Une application est un ensemble de composants coopérant grâce à un modèle de communication de type *RPC*[7] (Remote Procedure Call). L'environnement *CONIC* permet de séparer la programmation des composants de leur interconnexion dans le but de faciliter leur réutilisation. *CONIC* fournit un langage de configuration permettant de décrire une application sous la forme d'un assemblage de composant. De plus *CONIC* permet de déployer automatiquement une application répartie. Cette étape est gérée par un composant spécifique : le configurateur. Le configurateur a pour charge d'installer les composants sur différents sites et de les interconnecter.

Dans la suite, nous supposons que les requêtes sont indépendantes, c'est à dire que la terminaison d'une requête ne dépend pas d'une autre requête. Cette limitation a été introduite dans un premier temps par les concepteurs du système *CONIC* pour faciliter la compréhension de l'algorithme de reconfiguration.



Requêtes indépendantes :
l'achèvement de la requête a1 ne dépend pas de l'achèvement de la requête a2.

Figure I-6 : Illustration d'une requête dépendante

Les concepteurs de *CONIC* partent de l'hypothèse que *le configurateur* connaît les composants participants à une application et qu'il est capable de modifier ces interconnexions et de sauvegarder/restaurer l'état d'un composant.

Il reste à résoudre le problème des messages en transit et celui de la validation des changements.

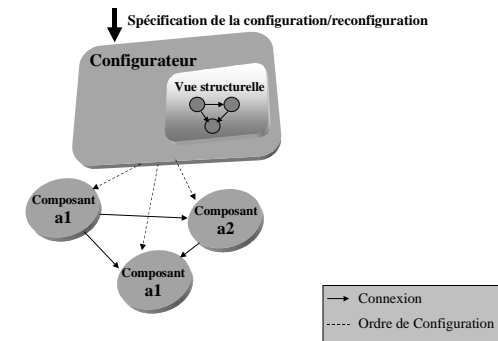


Figure I-6 : Architecture de CONIC

4.1.2 Architecture

L'architecture de CONIC s'articule autour d'un *configurateur*. Le configurateur a pour charge d'installer les composants sur différents sites, de les créer et de les interconnecter suivant la description de la configuration. Il possède une vue structurelle de l'application qui est construite à partir de la configuration de l'application. Le configurateur a également pour rôle de reconfigurer une application. La figure suivante illustre le processus de reconfiguration mené par le configurateur. Le configurateur reconfigure une application à partir des ordres de configuration, de la description structurelle de la configuration courante et des structures d'exécution de l'application (composants et interconnexion).

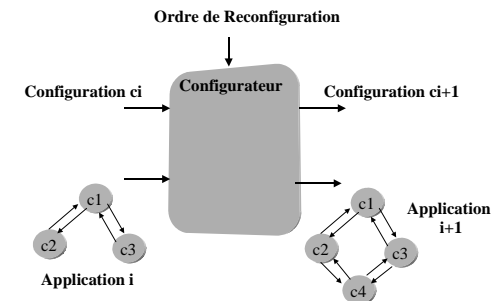


Figure I-7 : Illustration du support de CONIC pour la reconfiguration

4.1.3 Exemple : Une application annuaire

Cette petite application doit permettre à une personne de rechercher des individus sur des annuaires distants. Elle est construite à partir de deux composants :

- Un composant *client* : ce composant est la partie cliente de l'application, il contient l'interface graphique et la logique d'accès à un annuaire.
- Un composant *annuaire* : ce composant gère les annuaires et répond aux requêtes des clients.

Description visuelle de l'application

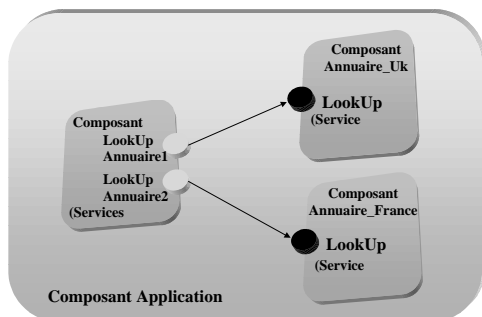


Figure i-8: description de l'application

Description des composants

- *Le composant client*

```

Group module client (maxannuaire = 3);
Use myTypes : annuaireEntry ;
Export Lookup_Annuaire [1..maxannuaire] reply annuaireEntry ;
End

```

- *Le composant annuaire*

```

Group module annuaire ;
Use myTypes : annuaireEntry ;
EntryPort lookup : lookupType reply annuaireEntry
End

```

Description de l'application

```

System Application ;
Create
  client1 : client          at toua.inrialpes.fr
  Annuaire_France : annuaire at dyade.inrialpes.fr
  Annuaire_UK : annuaire    at test.des.doc.ic.ac.uk ;
Link
  client1.Lookup_annuaire[1] to Annuaire_France.lookup ;
  client1.Lookup_Annuaire[2] to Annuaire_UK.lookup ;

```

4.1.4 Type de changement supportés

CONIC permet d'effectuer des **changements de structure** et des **changements de géométrie**. L'une des particularités de CONIC est que tous les types de changements utilisent une même mise en œuvre effectuée avec des primitives de base.

Les primitives de reconfiguration de CONIC sont :

- **Link** (ajout d'une connexion)

Cette primitive permet de connecter deux composants entre eux. Une fois deux composants connectés, ils peuvent communiquer librement.

- **Unlink** (retrait d'une connexion)

Cette primitive permet de déconnecter deux composants. Une fois qu'un composant est complètement déconnecté, il est dit *isolé*.

Pour que le configurateur puisse effectuer les ordres *link* et *unlink*, il doit pouvoir modifier dynamiquement les liaisons entre deux agents. De plus il est nécessaire que le canal de communication représentant la connexion à modifier soit vide sans quoi des messages en transit peuvent se perdre.

- **Add** (ajout d'un composant)

Cette primitive permet d'ajouter un nouveau composant dans la configuration d'une application.

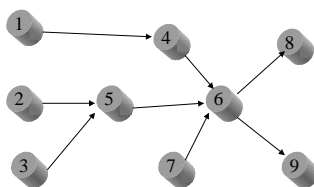
Cette primitive est utilisée lors de la *configuration* d'une application. Néanmoins, elle doit aussi pouvoir être utilisée lors de la reconfiguration. En effet, il doit être possible d'*ajouter dynamiquement* des composants à une application en cours.

L'ajout d'un composant à une application existante ne pose pas de problème car cette primitive ne modifie pas les composants et les interconnexions existantes. Lorsque des composants sont ajoutés à la configuration, ils sont isolés et ils ne perturbent pas l'application.

- **Remove** (retrait d'un composant)

Cette primitive permet de supprimer un composant de la configuration.

Pour pouvoir supprimer un agent, le *configurateur* doit être sûr qu'il n'y a plus de notification en transit *mettant en jeu* l'agent à supprimer. Le configurateur doit donc avoir la possibilité de vider ces canaux.



Pour que le composant 6 soit gelé, il faut que les composants 4,5,6,7 soient passifs

Figure I-9 : Exemple de gel d'un composant

4.1.5 Définition des états abstraits

L'état *abstrait* d'un composant est un état décrivant le composant par rapport aux requêtes dans lesquelles il intervient (par exemple un composant n'envoie plus de requête ...).

Pour préserver la cohérence de l'application, CONIC observe l'état *abstrait* des composants. L'état *abstrait* d'un composant est un état qui est déduit de l'état réel du composant. Les états abstraits permettent de déduire certaines propriétés de l'état concret d'un composant (comme par exemple le fait qu'un composant n'a pas de messages en transit ...). Le configurateur utilise ces propriétés pour reconfigurer les applications de manière cohérente.

Etat actif : Représente un composant en train de s'exécuter normalement. Le composant peut démarrer, accepter et exécuter des requêtes.

Etat passif : Le composant n'initialisera plus de requêtes et il n'y a pas de requête en cours dont il soit l'initiateur. Néanmoins, le composant peut continuer à accepter et à servir des requêtes (rappel : les requêtes sont indépendantes, le composant initiateur d'une requête à destination d'un composant passif ne risque pas d'être bloqué). Dans cet état, les canaux sortants sont vides et le resteront.

Etat gelé : Le composant ne recevra plus de requêtes, n'en n'initialisera plus et ne sera pas en train d'en servir une. Dans cet état, les canaux entrant/sortant sont vides et le resteront. Le composant est dans un état stable et cohérent. Un composant est dans l'état gelé s'il réunit les deux conditions suivantes : il est passif, et les composants à l'origine d'une connexion dont le destinataire est le composant gelé, sont passifs.

Le configurateur peut forcer un composant à passer dans l'état actif ou dans l'état passif grâce aux ordres « *activate* » et « *passivate* » auxquels chaque composant doit réagir. La figure suivante illustre les transitions d'états possible pour un composant. L'état gelé n'apparaît pas sur cette figure car cet état dépend de l'observation globale des états abstraits des composants de l'application.

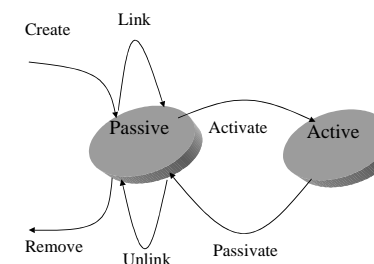


Figure I-10 : Transition d'état d'un composant

4.1.6 Les conditions associées aux primitives de reconfiguration

Cette section tente de formaliser les conditions associées aux primitives de reconfiguration par rapport aux états abstraits et aux remarques de la section précédentes.

□ Modification des interconnexions (Link/Unlink)

Pour que le configurateur puisse modifier des interconnexions, il faut que le composant à l'origine de la connexion soit gelé. De la sorte, aucune requête utilisant une connexion en cours de modification ne peut être initiée, les canaux relatifs à ces connexions sont vides et il n'y a aucune requête en cours utilisant ces connexions.

□ Suppression un composant (Remove)

Pour que le configurateur puisse supprimer un composant de l'application, il faut que le composant soit gelé et isolé. Un composant est isolé s'il est entièrement déconnecté. Les canaux relatifs à ces connexions sont vides et il n'y a aucune requête en cours utilisant ces connexions.

□ Ajout un composant (Add)

Il ne faut aucune condition particulière car un composant ajouter est toujours isolé.

Remarque : Les conditions associées aux états abstraits suivant la primitive de reconfiguration résolvent les problèmes des messages en transit dans les canaux. Toute la complexité de l'algorithme de CONIC réside dans le fait de pouvoir faire passer certains composants dans l'état *passif* ou *gelés* afin de vider les canaux de communication nécessaires pour effectuer une reconfiguration sans perdre les messages en transit.

4.1.7 L'algorithme de changement

Les conditions associées aux primitives de configuration précisées, nous pouvons décrire l'algorithme de changement.

Intuitivement, l'algorithme de reconfiguration doit vider certains canaux de communication pour ne pas perdre de message en transit lors d'une reconfiguration. Pour cela, l'algorithme de *CONIC* s'appuie sur la définition des états abstraits et notamment l'état *gelé* qui assure, entre autres, qu'aucun message n'est en transit sur les canaux de communications entrants et sortants d'un composant. Pour effectuer un ordre de reconfiguration, le configurateur doit donc faire passer des agents dans l'état *gelé* et pour cela, il doit aussi faire passer des composants dans l'état *passif*.

On appelle ensemble *QS* (Quiescent Set) l'ensemble des composants à faire passer dans l'état *gelé* pour effectuer un ordre de reconfiguration.

On appelle ensemble *CPS* (Change Passive Set) l'ensemble des composants à faire passer dans l'état *passif* pour geler les agents nécessaires à la reconfiguration.

Ces deux ensembles peuvent être déterminés en fonction de l'ordre de configuration et de la connaissance de la *configuration* de l'application par le configurateur.

Le configurateur détermine l'ensemble *CPS* suivant les composants à geler. Un composant appartient à *CPS* s'il a une connexion vers un composant à geler, cette information est connue du configurateur grâce à sa connaissance de la configuration.

L'algorithme de changement de *CONIC* se décompose en quatre étapes :

- Déterminer l'ensemble *QS* (Quiescent Set) des composants à mettre dans l'état Gelé. Cet ensemble dépend des primitives de changements utilisées et des conditions associées à ces primitives.
- A partir de l'ensemble *QS*, il faut déterminer l'ensemble *CPS* (Change Passive Set) des composants à mettre dans l'état passif pour que les composants appartenant à *QS* soient gelés.
- Procéder aux changements dans l'ordre suivant :
 - Envoyer l'ordre « passivate » à tous les composants appartenant à l'ensemble *CPS*.
 - Déconnexion des composants.
 - Destruction des composants.
 - Création des composants.
 - Connexion des composants.
 - Activation des composants.

Dans *CONIC*, les ordres de reconfiguration sont déclaratifs, c'est à dire que les ordres de reconfiguration ne sont pas forcément exécutés dans l'ordre effectué par l'utilisateur. Ce postulat permet au configurateur de paralléliser certains ordres de reconfiguration.

4.1.8 Extension de l'algorithme aux sessions dépendantes

L'algorithme présenté plus haut fonctionne sous la contrainte que les requêtes soient indépendantes. Les auteurs de *CONIC* ont étendu leur algorithme pour prendre en compte les sessions dépendantes. Pour cela, les auteurs partent du postulat qu'il peut exister une « chaîne de dépendance » entre les requêtes. Cette chaîne de dépendance peut comporter des cycles mais une requête est toujours bornée. Les requêtes dépendantes peuvent poser problème pour geler certains composants de l'application. L'exemple suivant illustre les problèmes rencontrés :

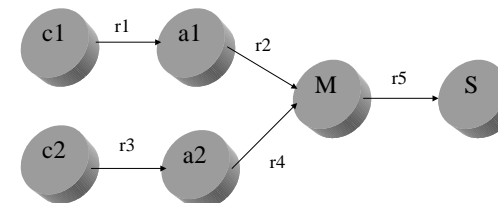


Figure I_11 : Application avec sessions dépendantes

Pour supprimer *S*, *M* et *S* doivent être gelés. De plus, les composants (*ai*) doivent être passifs. Le problème vient du fait que *c1* et *c2* peuvent envoyer des requêtes dépendantes vers *a1/a2* alors que *a1* et *a2* sont passifs. De même, si *M* est passif avant les composants (*ai*), l'algorithme initial peut se bloquer. Pour y remédier, les auteurs de *CONIC* proposent d'étendre le *CPS* (Change Passive Set) avec l'ensemble des composants qui peuvent initier une requête vers un composant passif ou gelé.

4.1.9 Avantages et inconvénients

Cet algorithme souffre d'un problème de performance dans le cas des transactions dépendantes. En effet, dans ce cas, le nombre de composants de l'application à geler peut être très important. De plus, le modèle de communication est forcé de type Client/Serveur. Néanmoins, *CONIC* a résolu les problèmes systèmes énoncés plus haut et évite toutes les incohérences lors d'une reconfiguration.

4.2 POLYLITH

4.2.1 Présentation

POLYLITH[22] est un projet de recherche constitué d'un environnement de développement dont une des composantes est un MIL (Langage d'interconnexion de module) permettant de générer des applications réparties.

L'approche de *POLYLITH* diffère celle de *CONIC* dans le sens où le programmeur inclus dans la mise en œuvre d'un composant un certain nombre d'informations permettant de le reconfigurer.

Le but d'un MIL est de séparer l'interconnexion des modules de leurs programmations. La principale différence entre les langages de configuration et les langages d'interconnexions de modules est que le composant est considéré comme une entité instanciable. La description d'un composant au niveau langage permet de créer de multiples instances d'un composant lors de l'exécution alors que l'entité module des MIL a le même statut que le module des langages de programmation classique : il est utilisable par plusieurs applications mais un module est unique dans un même espace d'exécution. De plus, ces langages introduisent une certaine structuration hiérarchique des modules logiciels de l'application accentuant ainsi leur intérêt pour la définition d'architecture réutilisable.

Ces applications se basent sur un *bus logiciel* pour la communication entre modules résidant sur des sites différents. Le bus logiciel doit permettre d'abstraire le modèle de communication, c'est à dire que l'on peut remplacer le modèle de communication (par exemple client/serveur) d'une application par un autre modèle de communication (par exemple mémoire partagée) sans modifier l'application. De plus, le bus logiciel est en charge de la localisation des services, de l'acheminement des requêtes et du contrôle des communications. Une des propriétés intéressantes de *POLYLITH* est que les applications réparties peuvent être conçues pour des systèmes hétérogènes.

4.2.2 Architecture

Dans l'architecture de *POLYLITH*, une application répartie est décrite comme un ensemble de modules interconnectés à l'aide d'un bus logiciel.

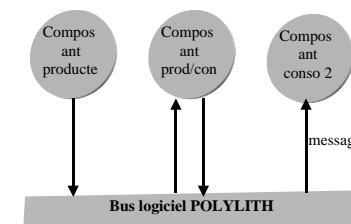


Figure I-12 : Architecture de *POLYLITH*

4.2.3 Exemple : l'application Annuaire

Pour fixer les idées, nous présentons l'application annuaire construite avec *POLYLITH*. Le programme MIL est composé en premier lieu de la définition des interfaces des deux modules, en terme de ressources qui sont fournis par le module (mot clé **define**) et de ressources dont le module requiert l'utilisation (mot clé **use**). Les ressources sont des interfaces de fonctions fournies ou requises pour le bon fonctionnement des modules. Chacun de ces modules contient aussi l'information sur la localisation des fichiers contenant la mise en œuvre des modules (attribut **SOURCE** de la description). L'application finale est constituée de la liste des composants nécessaires au fonctionnement (composant client et composant annuaire) pour lesquels il est possible de spécifier le site d'exécution (attribut **LOCATION**), ainsi que les liaisons entre les services requis et les services fournis par ces différents composants.

□ Description des composants

- Le composant client

```
Module client {
  Use interface Lookup : PATTERN = string
} : SOURCE=client.c
```

- Le composant annuaire

```
Module annuaire {
  Define interface Lookup : PATTERN=string
  RETURNS = ↑{string}
} : SOURCE=annuaire.c
```

□ Description de l'application

```
Module application {
  tool client : LOCATION=toua.inrialpes.fr
```

```

tool annuaire : LOCATION=dyade.inrialpes.fr
bind client.Lookup annuaire.Lookup
}

```

4.2.4 Type de changements supportés

POLYLITH supporte les changements suivants :

- ❑ Les changements de structure.
- ❑ les changements de géométrie.
- ❑ les changements d'implémentation.

Comme nous l'avons dit, l'architecture de *POLYLITH* s'articule autour du *bus logiciel*. Le bus logiciel a pour rôle d'acheminer les requêtes et de contrôler les communications. Contrairement à *CONIC*, les mécanismes de reconfiguration sont implantés dans le mécanisme de communication (i.e. dans le bus logiciel). Cela permet de faciliter le contrôle des messages en transit et la modification des références des services.

La possibilité pour le programmeur de définir des *points de reconfiguration* est l'une des particularités de *POLYLITH*. **Entre deux points de reconfiguration, aucun changement n'est possible.** La définition des points de reconfiguration dans le code du module est à la charge du programmeur. Le point de reconfiguration par défaut est le début du programme. Les points de reconfiguration permettent au programmeur d'adapter les possibilités de reconfiguration d'un module lors de sa mise en œuvre.

4.2.5 Préservation de l'état

La préservation de l'état est à la charge du programmeur. En effet, tout composant *POLYLITH* reconfigurable doit mettre en œuvre une fonction d'encodage de l'état du composant et une fonction de décodage de l'état du composant. La fonction d'encodage sert à sauvegarder l'état du composant dans un format spécifique et la fonction de décodage sert à décoder l'état du composant à partir d'un état encodé. Ces deux fonctions servent à préserver l'état d'un composant lors d'une reconfiguration. L'exemple suivant illustre l'utilisation de ces deux fonctions pour la reconfiguration.

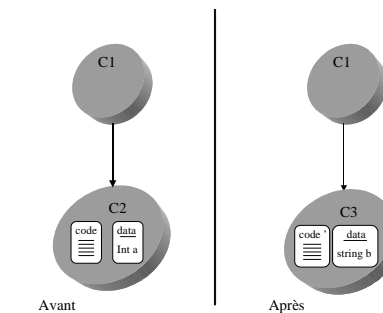


Figure II-13 : figure illustrant la restructuration de donnée

Dans ce scénario, le composant C2 est remplacé par le composant C3. Les structures de données et l'implémentation de C2 et de C3 sont différentes. La fonction d'encodage de C2 est utilisée pour sauvegarder l'état de C2 dans un format compréhensible par la fonction de décodage de C3. La fonction de décodage de C3 doit initialiser l'état de C3 **en fonction** de l'état de C2. Ces deux fonctions permettent de restructurer les données (Int->String, dans l'exemple). Néanmoins, cette restructuration nécessite l'implémentation spécifique des fonctions d'encodage et de décodage par le programmeur.

4.2.6 Algorithme de reconfiguration

De manière similaire à *CONIC*, *POLYLITH* utilise la même mise en œuvre pour tous les types de changements. L'algorithme de reconfiguration se compose de trois étapes :

- ❑ Blocage des canaux de communication nécessaires à l'isolement d'un composant.
- ❑ Envoi d'un message d'encodage de l'état qui sera pris en compte à un point de reconfiguration.
- ❑ Création du nouveau composant et initialisation avec l'état encodé de l'ancien composant. Les références sont gérées par le bus logiciel (Avec *CONIC*, c'est le configurateur qui gère ces références).

On constate que *POLYLITH* ne bloque que les canaux nécessaires pour la reconfiguration. Il n'y a pas de notion d'état abstrait, les composants réagissent de manière normale, ils peuvent recevoir et envoyer des messages. Néanmoins l'acheminement de ces messages est géré par le bus logiciel qui peut en bloquer certains pour isoler le composant reconfiguré.

4.2.7 Avantages et inconvénients

POLYLITH permet de reconfigurer des applications basées sur un bus logiciel. Le modèle de communication dépend du bus logiciel. *POLYLITH* permet la restructuration de données. Le bus logiciel facilite de nombreux problèmes liés à la reconfiguration (comme par exemple la

gestion des références qui peut être gérée par le bus logiciel). Le programmeur peut spécifier des points de reconfiguration. Globalement *POLYLITH* a de meilleures performances que *CONIC*.

Néanmoins, les mécanismes de reconfiguration de *POLYLITH* demandent du travail au programmeur dans la définition des points de reconfiguration et dans la définition des fonctions d'encodage et de décodage de l'état des composants. Les primitives de reconfiguration de *POLYLITH* sont de très bas niveau et sont directement adressées au bus logiciel. *POLYLITH* ne tient pas compte de la validation des changements. De plus, le bus est un goulot d'étranglement potentiel au niveau de l'acheminement des messages.

4.3 ARGUS

4.3.1 Présentation

ARGUS[20] a été développé au Massachusetts Institute of Technology (MIT). Il s'agit d'un langage et d'un système de programmation développé pour construire des applications réparties gérant des données persistantes et permettant un accès concurrent à ces données. Le but d'*ARGUS* est de fournir des mécanismes de tolérance aux fautes. Les mécanismes de tolérance aux fautes permettent à une application de continuer ou de reprendre son exécution dans un état cohérent.

Par rapport aux autres approches, *ARGUS* impose un système de programmation à base de « guardian » et « d'action ». Cette dernière approche diffère des deux précédentes par le fait que les mécanismes de reconfiguration fassent partie d'un système d'exploitation assurant des propriétés transactionnelles.

Comme nous le verrons dans la suite, l'idée d'*ARGUS* est d'utiliser des mécanismes transactionnels pour reconfigurer une application. Cette approche contraste avec *CONIC* et *POLYLITH* par le fait que *ARGUS* ne cherche pas à isoler un composant pour reconfigurer l'application dans un état stable. L'idée contenue dans *ARGUS* est que le mécanisme de retour arrière est utilisée pour reconfigurer une application dans un état stable et cohérent, le début d'une transaction étant forcément stable et cohérent.

Les deux principales abstractions définies par *ARGUS* sont le *Guardian* et l'*action* :

- Un Guardian est un objet actif désigné de façon unique dans le système. Un Guardian encapsule un ensemble d'objets de plus petite taille et un ensemble de flots d'exécution (threads) pouvant opérer sur ces objets. Les threads dans un *guardian* sont créés à la demande. L'interface d'accès à un *guardian* est composé d'un ensemble de méthodes (*handler*), permettant les appels entre gardians. Les objets locaux ne peuvent pas être partagés entre les gardians, ils n'appartiennent qu'à un seul Guardian. Les Gardians peuvent être répartis sur les machines du réseau.
- Une action est une transaction, les transactions sont des parties de code qui s'exécutent de manière atomique (validée ou annulée). De plus, les actions sont sérialisées, c'est à dire que le fait d'exécuter un groupe d'actions de manière concurrente est équivalent à une exécution séquentielle.

4.3.2 Exemple : l'application Annuaire

La figure suivante présente un *guardian* qui manipule un annuaire. Le *guardian* fournit le handler **Lookup** permettant de rechercher une personne dans l'annuaire.

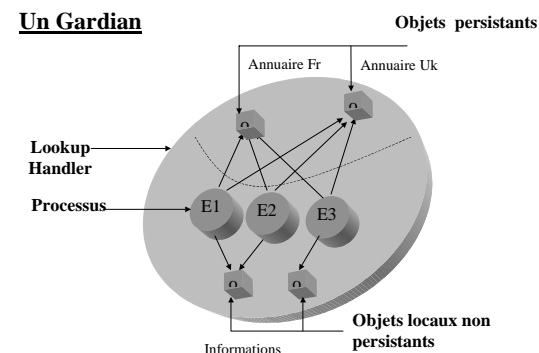


Figure I-14 : un exemple de guardian

Le guardian manipule :

- Des objets persistants pour gérer les annuaires. Les annuaires peuvent donc survivre à un crash de l'application.
- Des objets non persistants représentant des données non fondamentales ou pouvant être recalculées. Dans le cadre de l'application **annuaire**, ces objets pourraient gérer des informations additionnelles secondaires sur les personnes présentes dans l'annuaire.

4.3.3 Types de changement supportés

ARGUS permet d'effectuer les changements suivants :

- Ajout de composants.
- Retrait de composants.
- Déplacement de composants.

4.3.4 Algorithme de reconfiguration

Pour reconfigurer une application, *ARGUS* utilise le mécanisme de réparation d'erreur fourni par le système transactionnel intégré.

- Le système provoque un retour arrière, les actions en cours d'exécution qui ne sont pas encore validées sont annulées. Tant qu'une transaction n'est pas validée, le système maintient une version antérieure et correcte des gardians.

- Les versions antérieures des gardians de l'application correspondant à la dernière transaction validée, sont modifiées suivant les primitives de reconfiguration à exécuter.
- Les gardians modifiés sont redémarrés.

Il est important de souligner, à ce stade, la différence entre la reconfiguration dynamique et la tolérance aux fautes. L'objectif de la tolérance aux fautes est de fiabiliser une application en lui permettant de poursuivre ou de reprendre son exécution dans un état cohérent malgré l'occurrence de fautes. Le but de la reconfiguration dynamique est de fournir des mécanismes qui permettent de faire évoluer une application en cours d'exécution et de fournir un support pour effectuer de la répartition de charge. Certaines mises en œuvre de la tolérance aux fautes peuvent être utilisées pour effectuer la reconfiguration dynamique. C'est le cas des mécanismes de tolérance aux fautes qui se basent sur la capture de l'état global de l'application pour reprendre l'exécution de l'application à partir d'un état stable et cohérent.

4.3.5 Avantages et Inconvénients

Le système *ARGUS* permet de reconfigurer une application en utilisant des mécanismes transactionnels. La reconfiguration dans *ARGUS* ne nécessite pas de mécanismes supplémentaires pour effectuer la reconfiguration. *ARGUS* permet de disposer à la fois d'un mécanisme de reconfiguration et d'un mécanisme de reconfiguration. Un des intérêts d'*ARGUS* est que le système n'a pas besoin d'attendre qu'un gardian soit dans un état particulier pour le reconfigurer. Dès qu'une reconfiguration doit être effectuée, le système modifie les dernières versions du gardian. Ces versions ont été validées et sont forcément stables.

Néanmoins, *ARGUS* impose un modèle de programmation spécifique qui n'est pas forcément utilisable par toutes les applications. De plus, les applications doivent supporter le surcoût dû à l'aspect transactionnel d'une action. Les mécanismes de reconfiguration d'*ARGUS* sont difficiles à maintenir et à faire évoluer car ils sont mis en œuvre comme des mécanismes de bas niveau à l'intérieur du système *ARGUS*.

4.4 Les autres pistes possibles

4.4.1 L'algorithme du snapshot

□ Présentation

L'algorithme du Snapshot[10] permet de capturer un état global cohérent d'une application répartie. Dans la suite, une application répartie est constituée d'un ensemble de processus réparti sur un ensemble de sites et coopérant par échange de messages. L'état global capturé par le snapshot peut être utilisé pour redémarrer une application distribuée dans un état cohérent après un crash ainsi que pour détecter des prédicats stables ou instables [Babaoglu]. Un prédicat stable est une propriété qui, une fois qu'elle est vérifiée, le restera toujours. Par exemple, un deadlock est une propriété stable. Un état global cohérent d'une application

distribuée est constitué de l'état de tous les processus participants à l'application et des messages en transit sur tous les canaux de communication.

□ Intérêt du snapshot pour la reconfiguration

L'algorithme du snapshot peut être utilisé pour reconfigurer une application. En effet, certains problèmes de la reconfiguration sont liés à l'état des processus et à l'état des canaux. La principale difficulté de la reconfiguration réside dans le fait de préserver la cohérence de l'application. L'idée est d'effectuer la reconfiguration à partir de l'état stable et cohérent capturé par le snapshot. Néanmoins, il est nécessaire d'avoir des mécanismes pour modifier la configuration de l'application à partir de l'état cohérent capturé.

Ce type de mécanismes est très proche de *ARGUS*, le mécanisme de tolérance aux fautes utilisé (l'algorithme du snapshot) permet d'effectuer un retour arrière. Par rapport au système *ARGUS*, seul le mécanisme de tolérance aux fautes a été remplacé.

□ Avantages et inconvénients de l'algorithme du snapshot

Le principal inconvénient de l'algorithme du snapshot réside dans le fait que c'est un mécanisme coûteux. L'avantage d'utiliser un snapshot pour reconfigurer est que l'on dispose d'un état stable cohérent qui peut aussi être utilisé pour redémarrer l'application dans l'éventualité où une erreur fatale est survenue.

4.4.2 La migration de processus

□ Présentation

Les mécanismes de *migration de processus* [26][24] permettent de déplacer un processus en cours d'exécution d'une machine source vers une machine destination, ces deux machines étant reliées par un réseau de communication et n'utilisant pas de mémoire partagée. Un mécanisme de migration de processus consiste principalement à :

- Extraire le contexte d'exécution du processus sur la machine source vers la machine de destination.
- Créer, sur cette dernière, un processus auquel on affectera le contexte d'exécution transféré auparavant.
- Mettre à jour les liens de communication avec les autres processus.
- Détruire le processus résidant sur la machine source.

Il existe différentes stratégies de migration de processus. Cette différence est due principalement au contexte d'exécution transféré différent d'un mécanisme à l'autre. Si l'on considère que la structure d'exécution d'un composant est un processus, les différentes techniques de migration de processus sont alors équivalentes aux *changements de géométries*. De plus les techniques de migrations de processus se heurtent aussi aux problèmes liés à la

gestion des ressources. Les travaux effectués pour la gestion des ressources dans le cadre de la migration de processus peuvent s'avérer extrêmement intéressants dans le cadre de la reconfiguration dynamique.

4.4.3 La liaison dynamique

□ Présentation

Les mécanismes de liaison dynamique permettent de lier les références à des procédures externes pendant que l'application est en cours d'exécution. Ce type de mécanisme semble adapté aux changements d'implémentation étant donné qu'il permet de charger dynamiquement du code à l'exécution. Néanmoins, dans la plupart des implémentations des mécanismes de liaisons dynamiques, les références aux procédures externes ne peuvent pas être modifiées en cours d'exécution une fois qu'elles ont été établies.

5 Discussions

L'étude bibliographique réalisée m'a permis de définir le rôle de la reconfiguration, les questions qu'elle soulève et les réponses apportées par les autres projets. Elle m'a permis aussi d'examiner les principaux courants de recherche dans le domaine de la reconfiguration d'applications réparties à savoir les approches de *CONIC*, *ARGUS*, *POLYLITH*.

Ces trois approches se différencient par les points suivants :

- L'approche *CONIC* se base sur une abstraction de l'application et de l'état de ses composants. *CONIC* propose une vision « algorithmique » de la reconfiguration.
- L'approche de *POLYLITH* met en œuvre des mécanismes de reconfiguration sur un bus logiciel, i.e. au niveau du système de communication entre composants.
Cette approche se base sur le fait que le programmeur inclus dans la mise en œuvre d'un module, un certain nombre d'informations permettant de le reconfigurer : **ce sont les points de reconfigurations**.
- L'approche d'*ARGUS* diffère des deux précédentes par le fait que les mécanismes de reconfiguration font partie d'un système d'exploitation assurant des propriétés transactionnelles.

Ces approches proposent des solutions à partir desquels nous pouvons dégager certaines conclusions intéressantes :

- Les mécanismes de reconfiguration doivent être indépendants d'un système d'exploitation particulier pour permettre une maintenance et une portabilité plus aisées. Des systèmes comme *ARGUS* qui ont été développés sur Unix

sont très peu portables et les mécanismes de reconfiguration sont très liés à ce système.

- Il est possible de corriger les incohérences en utilisant des techniques de points de reprise et de recouvrement (cf *ARGUS*). Mais ces solutions sont coûteuses.
- L'occurrence d'une panne lors de la reconfiguration d'une application ne doit pas laisser l'application dans un état incohérent. Cela implique que notre mécanisme de reconfiguration doit être tolérant aux pannes.
- L'occurrence d'une reconfiguration est un événement de courte durée dans la vie d'une application. Il est donc acceptable que le mécanisme de reconfiguration ait un certain coût à l'exécution à partir du moment où la reconfiguration laisse l'application dans un état cohérent.
- Il n'est pas déraisonnable de demander aux programmeurs un minimum de coopération. Cette coopération peut se matérialiser comme un ensemble de conventions de programmation.
- La reconfiguration est grandement facilitée par la séparation de la programmation des composants et de leurs assemblages (cf *CONIC*). En effet, la modification des interconnexions des composants d'une application est grandement facilitée si la définition de ces interconnexions n'est pas exprimée dans la mise en œuvre des composants.
- Un composant reconfiguré doit pouvoir continuer son exécution à l'endroit où il en était avant sa reconfiguration. L'endroit où un composant peut être reconfiguré est important car il doit permettre facilement au composant de reprendre son exécution.
- Le modèle de communication synchrone facilite le vidage des canaux de communication car lorsqu'un composant reçoit la réponse à une requête, il peut être sûr que le message qu'il a envoyé est arrivé à bon port. Cependant le modèle de communication asynchrone pose des problèmes lors des requêtes dépendantes (cf *CONIC*).
A l'inverse, le modèle de communication asynchrone permet de faciliter la reconfiguration dans le cas des requêtes dépendantes car un composant n'est pas bloqué en attendant le résultat d'une requête.

La suite du travail consiste à définir les différents types de changements que l'on veut rendre possibles et de s'inspirer de l'état de l'art pour mettre en œuvre des supports pour la reconfiguration dynamique dans un contexte d'exécution particulier, le modèle de composant OLAN et le bus de message AAA qui sont présentés dans la suite.

Chapitre II.

Un contexte pour la reconfiguration

1 Le modèle A3

1.1 Le modèle événement-réaction

Durant ces dernières années, le domaine des applications réparties a connu, avec le succès des réseaux, un engouement sans précédent de la part des industriels. Pour répondre à ces besoins, la plupart des développeurs ont utilisés pratiquement systématiquement des modèles de communications synchrones. Cependant, pour certains types d'applications [byte], les modèles de communications asynchrones sont plus adaptés. Le modèle **événement-réaction** est un modèle de communication asynchrone dans lequel un message représente un événement auquel un certain nombre d'agents doivent réagir.

Agent : Les Agents sont des objets réactifs qui communiquent grâce à des événements. Lorsqu'un agent reçoit un événement, il exécute la réaction associée.

Événement : Un événement est représenté par une transition d'état significative à laquelle un ou plusieurs agents vont réagir.

1.2 Présentation du modèle A3

Le modèle **A3** (Agent Anytime Anywhere) a été développé par l'action **DYADE** dans le cadre d'une coopération **BULL-INRIA**. L'environnement A3 fournit un modèle de base pour la construction d'applications réparties à base d'agents. Les motivations des architectes du modèle A3 sont de fournir un environnement de développement d'applications réparties conformes au modèle événement-réaction. Cet environnement permet la conception d'applications tolérantes aux pannes en s'appuyant sur certaines propriétés concernant l'acheminement des messages.

1.2.1 Concept de base

□ Les agents A3

Les agents A3 sont des objets **passifs** et **persistants**.

- **Objet Persistant :** Un objet est persistant si son état (code + donnée) ne disparaît pas avec l'arrêt de la machine sur lequel il réside.
- **Objet passif :** Un objet est passif lorsqu'il ne contrôle pas son flot d'exécution. Cela veut dire que l'exécution d'un objet passif est cadencé par une entité logicielle externe à l'objet. Dans le cadre du modèle A3, cette entité externe est **un moteur d'exécution**.

□ Les notifications

Dans ce modèle, un événement est représenté par *une notification* ; une notification est un objet passif qui est signalé à l'agent destinataire afin qu'il exécute la réaction appropriée. Les notifications transitent via un bus logiciel.

- **Le bus logiciel** est l'entité logicielle **répartie** qui a pour rôle d'acheminer des notifications à un ensemble de destinataires en suivant certaines propriétés sur la transmission des notifications.

1.2.2 Propriétés

Cette section précise les propriétés associées au modèle A3. Ces propriétés sont au nombre de deux : la première concerne l'exécution des réactions et la deuxième est relative à l'acheminement des notifications.

- **Exécution des réactions de manière transactionnelle**

Cette propriété permet d'assurer que l'exécution des réactions est transactionnelle (ou *atomique*), c'est à dire qu'une réaction est annulée si elle ne peut être menée à son terme. Cette propriété évite que l'application soit dans un état incohérent à cause d'une panne survenue au milieu de la réaction d'un agent (Les notifications envoyées par un agent lors d'une réaction, ne sont effectivement envoyées que lorsque la réaction est validée).

C'est *le moteur d'exécution* qui assure que les réactions s'effectuent de manière *transactionnelles*.

- **Diffusion fiable et causale des notifications[9]**

Cette propriété assure que l'envoi d'une notification s'effectue :

- **De manière fiable**, c'est à dire que toutes les notifications seront délivrées une fois et une seule à chaque destinataire.
- **Suivant l'ordre causale d'émission des notifications**.

Un agent A émet successivement les notifications n1 et n2, alors aucun agent ne recevra n2 avant n1.

Un agent A1 émet une notification n2, et si un agent après avoir reçu n1 émet une notification n2, alors aucun agent ne recevra n2 avant n1.

L'intérêt de cette propriété est d'assurer la réception de notifications suivant leur « ordre de dépendance ». Ceci permet d'éviter qu'un agent reçoive des notifications réagissant à une notification dont il n'a pas encore connaissance.

1.3 Mise en œuvre

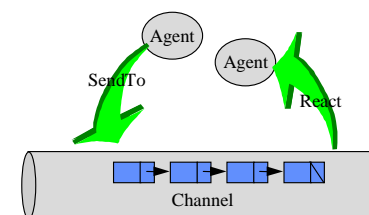


figure II-1 : Architecture générale du modèle a3

1.3.1 Les agents A3

Les agents A3 sont des objets définis à partir d'une classe de base : la classe **Agent**. Cette classe sert de canevas à tous les agents, les autres classes d'agents héritent de cette classe. La méthode *react* d'un agent définit les réactions associées aux notifications auxquelles l'agent doit réagir.

Le moteur d'exécution soumet cette notification à l'agent destinataire en appelant la méthode *react* de cet agent. C'est *le moteur d'exécution* qui assure que l'exécution de la méthode *react* est atomique (soit elle est validée, soit elle est annulée).

1.3.2 Le bus logiciel A3

Le bus logiciel A3 est répartie sur un exemple de machines. Son rôle est d'acheminer les notifications vers les agents destinataires. Un agent envoie une notification en utilisant la primitive **Sendto** fournie par le bus. Cette primitive envoie les notifications vers *le moteur d'exécution* associé aux agents destinataires. *Le moteur d'exécution* appellera alors les réactions correspondantes des agents destinataires.

1.3.3 L'agent Factory

Les agents A3 peuvent être créés à distance, l'intérêt de cette propriété est de permettre un certain dynamisme dans le processus de création des agents.

L'agent Factory est un agent qui a pour rôle de *créer* les agents. Si la classe représentant l'agent n'est pas présente localement, l'agent factory charge la classe à partir d'un site distant. Cela facilite, entre autres, le déploiement des agents participants à une application.

La création d'un agent s'effectue en deux phases :

- *Phase 1 : création et initialisation de l'objet représentant le futur agent*

Un objet représentant le futur agent est créé localement, il sert à faciliter l'initialisation de l'agent.

- *Phase 2 : déploiement de l'agent sur son site d'exécution*

Le déploiement est effectué en envoyant une notification de création d'agent vers l'agent factory du site où doit résider l'agent. Cette notification de création contient l'objet représentant le futur agent. Cette notification de création est soumise aux mêmes propriétés transactionnelle et de causalité que toutes les autres notifications. De cette manière, dès que la notification de création est envoyée, on est assuré que l'agent sera créé.

1.4 L'application annuaire

□ L'agent client

```
public class Client_annuaire extends Agent
{
  AgentId IDANNUAIRE; // référence initialisée par le configurateur

  private void GetOrder() {
    DataInputStream in = new DataInputStream(System.in);
    String s=in.readLine();
    sendTo(IDANNUAIRE, new LookUpNot(s));
  }
  public void react(AgentId from, Notification not)
  {
    if (not instanceof LookUpAckNot)
    {
      // affichage des résultat de la recherche
      System.out.println(((LookUpAckNot)not).result) ;
    }
    this.GetOrder();
  }
}
```

La méthode GetOrder permet d'envoyer une notification d'interrogation (Lookup) vers l'agent annuaire identifié par la variable IDANNUAIRE qui est initialisée par le configurateur.

□ L'agent annuaire

```
public class annuaire extends Agent
{
  private void LookUp() {
    ...
  }
  public void react(AgentId from, Notification not)
  {
    if (not instanceof LookUpNot)
    {
      // affichage des résultat de la recherche
      string result=LookUp(((LookUpNot)not).name) ;
      sendto(from, new LookUpAckNot(result)) ;
    }
  }
}
```

Lorsque l'agent annuaire reçoit une notification *Lookup*, il effectue la réaction correspondante pour rechercher une personne dans l'annuaire.

2 Le modèle OLAN

2.1 Introduction

L'évolution conjuguée de la technologie des télécommunications et de la structure des entreprises et des organisations conduit vers l'émergence d'applications réparties de grande complexité. La conception, le développement et la maintenance de ces applications posent des problèmes difficiles pour lesquels les solutions actuellement disponibles sont jugées insuffisantes par les utilisateurs. Parmi les contraintes principales auxquelles sont confrontés les concepteurs d'applications, on peut citer :

- la **réutilisation** et l'**intégration** du code existant,
- la **configuration d'une application**. Cette opération recouvre deux étapes du cycle de vie d'une application. Dans un premier temps, il faut décrire les entités (composants logiciels) qui constituent le corps de l'application, puis les schémas de communication et de synchronisation entre ces entités. Dans un deuxième temps, il faut **déployer** les éléments d'une instance de l'application dans un environnement donné.
- la **capacité d'administrer** les divers éléments de l'application pour adapter la structure opérationnelle de l'application aux ressources disponibles sur le système hôte. Cette capacité d'adaptation doit également permettre de répondre à l'évolution de l'application et de son environnement d'exécution dans le temps.

L'objectif de l'action OLAN est de fournir des outils répondant à deux besoins :

- Construire des applications réparties en combinant des techniques de programmation à base d'objets et des techniques d'intégration de composants hétérogènes en minimisant les coûts de développement inhérents au facteur de répartition.
- Faciliter l'administration, la configuration et l'évolution de ces applications. Notamment fournir des outils permettant de déployer une application automatiquement, d'exprimer des contraintes sur le placement des composants.

Le principe directeur est que les applications distribuées doivent être conçues et réalisées comme des ensembles ouverts et évolutifs de sous-systèmes applicatifs (appelés composants) coopérants, dont certains sont constitués d'applications préexistantes. Une application est **donc un ensemble de composants interconnectés** et cet ensemble est lui même un composant.

Un des intérêts de ce type d'approche est de séparer la programmation des composants de leur mise en œuvre. Ceci permet de faciliter la construction d'application car la conception logique d'une application est séparée en deux niveaux distincts :

- **La description de l'assemblage des composants utilisés.** Cette étape offre une vue structurelle de l'application qui peut servir de canevas à la conception de l'application.
- **La mise œuvre des agents.** Cette étape peut être effectuée de manière indépendante de l'application auquel le composant va participer. Ceci permet de faciliter la *réutilisation* du composant dans une autre application.

2.2 Présentation du modèle OLAN

Le modèle OLAN [2],[3] fournit un ensemble d'outils composés d'un langage de configuration et d'un support d'exécution. Il facilite la conception, la configuration et l'évolution des applications distribuées faites de composants logiciels hétérogènes.

La configuration d'une application décrit son organisation et son déploiement ainsi que son comportement à l'exécution. La configuration s'effectue à deux niveaux :

- Pour le constructeur d'application, l'identification des composants logiciels et la description de leurs interconnexions et communications,
- Pour les administrateurs, l'utilisation précise des ressources systèmes fournies par l'environnement cible comme le placement des composants.

Le modèle OLAN est divisé en deux parties qui sont :

1/ Tout d'abord le niveau application, qui permet de décrire l'application en terme de composants , connecteurs et interconnexions entre eux à l'aide d'un langage de configuration d'application ;

2/ Puis le support d'exécution, il a pour rôle de supporter la programmation par composants et connecteurs. Lorsqu'une application est déployée sur le support OLAN, le compilateur doit se charger de générer le code exécutable par la machine virtuelle OLAN.

2.3 Le langage OCL

Le langage OCL est un langage de définition d'architecture conçu pour des applications réparties. Les concepts de base de ce langage sont les composants, connecteurs et les configurations, tels que nous les avons définis dans les sections précédentes. L'utilisation de ces concepts au sein du langage OCL apporte des solutions aux préoccupations suivantes :

- L'intégration de composants logiciels hétérogènes au sein d'un modèle uniforme
- La description explicite de l'architecture de l'application, i.e. des structures d'exécution requises et de leur manière de communiquer entre elles.
- La définition du placement de l'application sans programmation au sein même des composants logiciels

2.3.1 Intégration de logiciel

Le concept de composant dans le langage OCL se traduit par deux types d'entités : les composants primitifs et les composites. Les composants primitifs sont les unités d'encapsulation de code. Ils possèdent une interface qui explicite les services disponibles aux autres composants de l'application. La particularité de l'interface des composants est de rendre visible en son sein les services requis par le composant pour fonctionner correctement et pour pouvoir jouer son rôle dans une architecture. Cela correspond donc à exhiber les dépendances fonctionnelles du composant avec le monde extérieur. Le corps même d'un primitif contient un ensemble d'informations qui décrivent de la manière la plus complète possible la nature du logiciel encapsulé, sa localisation physique et la manière d'y accéder. Les composants primitifs sont ainsi les briques de base car ce sont eux qui contiennent le code opérationnel de l'application.

Voici l'exemple OCL de l'interface du composant Annuaire. L'annuaire possède un attribut Pays qui indique le lieu géographique des personnes contenues dans l'annuaire. Il offre un service de récupération de l'adresse d'une personne, Lookup. Pour illustrer les différents types de services, nous introduisons un nouveau service par rapport à l'exemple décrit dans le chapitre précédent. Ce service envoie, de manière asynchrone, un rapport des statistiques d'utilisation de l'annuaire, SendStats.

```

From oil::types import * // Inclusion de fichiers de définition d'interfaces.

typedef sequence<char> statStruct; // déf d'un type séquence illimitée de caractères

// Définition de l'interface de l'annuaire
interface AnnuaireItf {
readonly attribute string Pays;
provide Init();
provide Lookup( in string cle, out string email);
notify SendStats(in statStruct statReport);
}

```

Figure II-2. Interface OCL de l'Annuaire

La syntaxe d'une implémentation est illustrée avec l'exemple suivant de l'annuaire. Une implémentation est identifiée par un nom et elle "*implémente*" une interface. L'opérateur syntaxique ":" est ici pour indiquer la relation entre l'identifiant de l'implémentation et une interface. Le reste de l'implémentation contient les liaisons entre les services et attributs de l'interface et ceux présents dans les modules. L'absence de liaison entre une déclaration de l'interface et un module déclenche des avertissements au niveau du compilateur du langage, car cela peut induire un possible dysfonctionnement du composant.

```

from oil::annuaire import AnnuaireItf // inclusion de la déf de l'interface

primitive implementation AnnuaireImpl : AnnuaireItf
use AnnuaireMod // liste des modules utilisés
{
// liste de la projection de l'interface vers les modules
Pays -> AnnuaireMod.Pays; // L'attribut Pays existe dans le module
Init -> AnnuaireMod.Init();
Lookup() -> AnnuaireMod.Lookup();
AnnuaireMod.Stat() -> SendStats();
}

```

Figure II-3. Implémentation Primitive de l'Annuaire en OCL

Le composant dans le modèle OCL peut être qualifié de glu entre une interface et une implémentation. Cette glu est similaire au concept de type de composant de Darwin ou UniCon dont les architectures sont composées d'instances de ces types. La syntaxe OCL d'un composant est la suivante :

```

component Annuaire {
interface AnnuaireItf;
implementation AnnuaireImpl;
}

```

Figure II-4. Composant OCL

Le module contenu dans le composant Annuaire indique l'emplacement des fichiers sources contenant le code de l'annuaire. Ce code est écrit en Python et la syntaxe du module est la suivante :

```

from oil::annuaire import AnnuaireItf

module "Python" AnnuaireMod : AnnuaireItf { // on réutilise l'interface du
// primitif
path: "${OLAN_SRC}"/examples/annuaire";
sourceFile: "AnnuaireMod.py";
}

```

Figure II-5. Module Annuaire en OCL

2.3.2 Définition de l'Architecture

Une architecture d'application (appelée parfois une configuration) contient toutes les informations relatives à l'instantiation des composants logiciels et des interconnexions entre ces composants, c.à.d. la résolution des dépendances entre composants et chaque protocole et mécanisme qui seront utilisés lors des communications entre composants. La définition de cet assemblage de composants est faite au sein des composites, qui sont des composants sans effet pour l'exécution mais qui aident à structurer une application en une hiérarchie de composants interconnectés.

Structure d'une application

La structure de l'application est donc contenue dans des composites. Ce sont des composants, avec une interface ayant les mêmes caractéristiques que celle des composants primitifs. Seule la mise en œuvre d'un composite est radicalement différente de celle d'un primitif car elle ne

contient aucune référence directe au logiciel. Elle contient la définition de l'instantiation des composants nécessaires à remplir les services fournis au niveau de l'interface du composite. Elle contient aussi la définition de toutes les communications pouvant se produire entre ces composants nécessaires. Un composite est donc une entité de structuration, qui offre la possibilité à l'architecte de grouper des entités logicielles (i.e. des composants primitifs) selon son bon vouloir : critères fonctionnels, de placement, d'utilisation de ressources communes, ...

La figure suivante illustre la mise en œuvre du composant *application annuaire*. Nous pouvons remarquer que ce composant comporte des composants primitifs (annuaire, client et admin). Cet exemple montre la projection des services de l'interface vers les services des sous composants

```

Implementation AppliImpl : AppliItf
use AnnuaireItf, ClientItf, AdminItf // Inclusion des interfaces des sous
composants nécessaires
{
// Définition des sous composants
client = instance ClientItf;
annuaireFr = instance AnnuaireItf("FRANCE");
annuaireUk = instance AnnuaireItf("UK");
admin = instance AdminItf;

// Projection des services de l'interface vers les services des sous composants
LanceClient() => client.Init();
LanceAdmin() => admin.Init();
...
}

```

Figure II-6. Implémentation d'un composite en OCL

Interconnexions

Elles permettent de spécifier la communication entre 1 ou plusieurs composants. Ceci comprend la définition explicite des dépendances fonctionnelles entre composants (le composant A a besoin d'un service R qui sera fourni par le composant B), l'adaptation de la communication selon la nature des composants (je dois faire communiquer un objet Java avec un objet CORBA), l'adaptation du flot de données (le paramètre A qui est un entier doit être transformé en une chaîne de caractères selon sa valeur), et finalement la nature du mécanisme de communication qui sera utilisé à l'exécution (un appel synchrone entre deux sites reposera sur des sockets, un RPC ou un appel via un courtier d'objet CORBA). Un objet particulier permet de spécifier ces propriétés : les connecteurs dont la table suivante montre quelques uns de ceux disponibles.

Nom	Appelant(s)	Appelés(s)	Propriétés	Mécanisme	Paramètres
<i>SyncCall</i>	1 require	1 provide	Communication Synchrones Signatures compatibles ou clause do	Local: appel de fonction ou méthode IPC: utilisation de ILU Distant: utilisation d'ILU	Aucun
<i>AsyncCall</i>	1 notify	1 react	Communication asynchrone Signatures compatibles ou clause do	Local: appel de fonction dans un thread séparé IPC : utilisation d'ILU en asynchrone Distant : utilisation d'ILU en asynchrone	Aucun
<i>RandSyncCall</i>	1 require	1 à n provide	Idem syncCall, mais choix aléatoire d'un seul destinataire	idem syncCall	Aucun
<i>CreateIn-Collection</i>	1 require	1 provide au sein d'une collection	Création d'une instance dans une collection, puis appel synchrone du service appelé. Signatures compatibles ou clause do	idem syncCall en fonction du site de création du composant de la collection	Valeurs des attributs du composant créé.
<i>DeleteIn-Collection</i>	1 require	1 provide au sein d'une collection	Appel du service appelé synchrone puis suppression du composant de la collection	Idem appel synchrone.	Aucun
<i>Aaa</i>	1 notify d'un composant primitif contenant des modules "AAA"	1 react d'un composant primitif contenant des modules "AAA"	Appel asynchrone Signatures compatibles. Pas de clause do	Bus logiciel AAA prenant en charge la répartition des composants.	Aucun

Figure II-7 :Connecteurs OCL

Les connecteurs sont désignés par un nom unique, chacun d'entre eux correspondant à un type de communication particulière et à leur mise en œuvre. On peut voir sur le tableau précédent qu'il existe un connecteur pour la communication synchrone et asynchrone. Ils utilisent tous les deux ILU, un courtier d'objet dès lors que les composants interconnectés sont sur des sites différents. La création d'un nouveau connecteur fournissant la même nature de communication mais reposant sur un mécanisme différent (socket TCP et UDP par exemple) est tout à fait possible. Du point de vue de l'architecte d'application, cela n'a aucune importance du moment que la mise en œuvre est conforme à un appel synchrone ou asynchrone dans ce cas précis.

Reprenons l'exemple de l'annuaire. L'application est composée de plusieurs instances de composants : le client, l'administrateur et les annuaires. Son implémentation contient la définition des interconnexions de ces composants.

```

....
client.Lookup_Annuaire (cle, adresse) -> annuaire.Lookup(cle, adresse)
    using syncCall();
annuaire.SendStats( statReport) -> admin.GetReport( report)
    using asyncCall()
do { // clause de transformation de paramètres
    report = annuaire.Pays + " : "+ statReport;
    // concaténation du nom du pays en début de rapport.
    // Le rapport est une séquence illimitée de caractères,
    // d'où la possibilité d'utiliser les opérateurs de
    // concaténation de chaîne +
}
...

```

Figure II-8. Interconnexions dans l'application Annuaire

Le client est connecté à un annuaire par le biais d'un connecteur effectuant un appel synchrone, syncCall. Les signatures sont identiques et les services sont un require et un provide. L'interconnexion est donc valide comme le montre le Figure qui dresse la liste des connecteurs existants dans OCL ainsi que leurs caractéristiques. La seconde interconnexion utilise un connecteur asynchrone, car les services sont de type notify et react. On peut toutefois remarquer la présence d'une clause de transformation qui ajoute en entête du rapport transmis d'un annuaire vers l'administrateur, le nom du pays de l'annuaire. Dans cette clause, l'opérateur de concaténation entre chaînes est utilisé ainsi que la valeur d'un attribut de l'annuaire.

2.3.3 Répartition

Une fois que l'architecture est définie, il est possible de spécifier le placement des composants de manière orthogonale, car chaque composant est susceptible de s'exécuter indépendamment des autres composants et les dépendances entre composants passent par le médiateur qu'est le connecteur. OCL associe par défaut à chaque composant un ensemble d'attributs d'administration, dont les valeurs positionnées par l'architecte permettent de choisir le site de placement au moment du déploiement de l'application.

Les attributs d'administration *Node* et *User* permettent de spécifier les contraintes imposées pour le choix du site d'exécution et de l'utilisateur pour qui l'exécution du composant aura lieu. Le choix d'un site et d'un utilisateur permet de choisir un espace d'exécution du composant appelé *Contexte*. Tous les composants dont le site et l'utilisateur choisis sont identiques, sont placés dans le même contexte. Un contexte est généralement constitué d'un processus. Néanmoins, en fonction du type de logiciel intégré par les composants, il est possible que le Contexte gère plusieurs processus.

L'architecte est en fait capable d'exprimer le positionnement de ces valeurs par le biais des politiques d'administrations dont voici un exemple :

```

management AnnuaireMgmt : AnnuaireImpl {
Node.name == "db?.inrialpes.fr"; // un site dont le nom commence par
// db, puis un caractère dans le doamine
//inrialpes.fr
Node.CPULoad <= 10; // La charge moyenne constatée lors de
//l'installation doit être inférieure à 10
//echelle de 0 à l'infini, la charge 100 étant
// une utilisation standard.
Node.UserLoad <= 10; // Nombre d'utilisateurs moyen inférieur à 10
User.name == "admin"; // appartient à l'utilisateur privilégié
}

// Composant de surveillance des exécutions
management AdminMgmt : AdminImpl {
User.name == "admin"; // appartient à l'utilisateur privilégié
}

management AppliMgmt : AppliImpl{
client.Node != annuaires.Node ; // Le client est sur un site différent
// de la collection d'annuaires.
// la collection des annuaires peut être
// localisée sur plusieurs sites
// en fonction des créations d'annuaires.
}

component Appli {
interface AppliItf;
implementation AppliImpl {
ClientItf is Client;
AnnuaireItf is Annuaire;
};
management AppliMgmt;
}

```

Figure II-9. Répartition de l'application Annuaire en OCL

Il est aussi possible d'utiliser des attributs du composant pour la définition des critères de placement. Par exemple, imaginons que le placement des annuaires se fasse sur un site appelé `db_<pays>.inrialpes.fr` avec `pays` la valeur de l'attribut de l'annuaire lors de sa création. La section `management` suivante permet de le faire :

```

management AnnuaireMgmt : AnnuaireImpl {
Node.name == "db_"+pays+".inrialpes.fr";
...
}

```

Ceci permet de paramétrer le placement du composant par rapport à des attributs qui peuvent varier d'une application à une autre.

2.4 Le support de configuration OLAN

2.4.1 Présentation

Le support de configuration OLAN a pour charge de permettre la configuration et l'exécution d'applications décrites avec le langage OCL. Ses différentes fonctions sont :

- **Le déploiement** des composants logiciels en fonction des contraintes de placement contenues dans la description OCL.

- **L'installation** des composants logiciels à l'endroit choisi et le positionnement des attributs.
- La mise en place des **interconnexions** entre les composants.
- **La gestion des exécutions.**

2.4.2 Installation d'une application

□ Le déploiement

Le déploiement est l'action de choisir les sites et les processus qui hébergeront les différents composants de l'application. Le support de configuration interprète le script de configuration fourni par le compilateur OCL. Ce script contient en premier lieu toutes les indications de création des composants auxquelles sont associés les critères de placement des composants. Le support est alors en charge de trouver un site qui soit capable de créer le composant, i.e. qui ait accès au logiciel intégré dans le composant et qui réponde aux critères de placement.

□ L'installation

L'installation du composant consiste à créer l'objet composant que le compilateur produit, et à charger le logiciel intégré ainsi que les talons et les empaqueteurs associés. De plus, le support positionne les valeurs des attributs des composants.

□ Les interconnexions

La dernière phase de mise en place de l'application consiste à créer les interconnexions de composants. Il s'agit ici de mettre en place les objets connecteurs et de connecter les services d'entrée et de sortie des composants aux connecteurs adéquats. La phase de mise en place d'un connecteur contient le choix du mécanisme de communication en fonction du placement des composants, la création des objets permettant d'utiliser le mécanisme de communication choisi et l'insertion de l'éventuel code de transformation des paramètres et de désignation associative.

□ L'exécution

Le support a pour ultime tâche la gestion du lancement de l'application, de la connexion et de la déconnexion des utilisateurs habilités à joindre une application en cours d'exécution.

2.4.3 Architecture

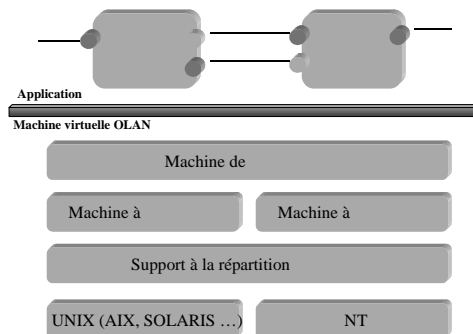


Figure II-10 : L'architecture OLAN

Le niveau « application » est fait d'un langage de configuration d'application. Il a pour rôle de décrire l'application en définissant les composants, les connecteurs nécessaires à l'application et ensuite l'interconnexion entre les composants.

La machine virtuelle OLAN comprend trois machines :

- **La machine à composants** qui gère l'encapsulation d'entité logicielles,
- **La machine à connecteurs** qui gère les interactions entre les composants,
- **La machine à configuration** qui effectue les opérations de structuration des composants en une hiérarchie ainsi que les opérations de configuration de l'application.

Une des propriétés de la machine virtuelle est qu'elle doit, d'une part, pouvoir être portée d'un environnement d'exécution à un autre, et que, d'autre part être hétérogène, c'est à dire elle doit fonctionner de la même manière avec les mêmes codages et types d'information d'un système à un autre. Pour assurer ces deux propriétés, le développement de cette machine s'appuie sur le langage et le noyau d'exécution python disponible sur de nombreuses plateformes et gérant l'hétérogénéité des environnements.

2.5 Avantages et inconvénients de l'environnement A3-OLAN

2.5.1 Avantages

L'environnement A3 possède des propriétés intéressantes pour la reconfiguration :

- La propriété de causalité entre les notifications permet un contrôle facile des canaux de communication.

- L'environnement A3 est écrit en java, ce qui permet la portabilité de l'environnement sur d'autres plates-formes et cela permet d'utiliser des mécanismes «objets» (comme l'héritage par exemple) pour créer notre support pour la reconfiguration.
- L'utilisation de l'environnement A3 nous permet de concevoir facilement un mécanisme de reconfiguration transactionnel.
- Le modèle OLAN possède aussi des idées intéressantes pour la reconfiguration :
 - la notion d'ADL permet de décrire la configuration d'une application. Si une application est décrite grâce à l'ADL, elle est construite correctement. Cela nous permet de reposer le processus de validation des changements sur l'ADL.
 - La séparation de la programmation des composants de leur interconnexions réduit la spécialisation d'un agent et augmente sa reconfigurabilité.

Pour ces raisons, nous voulons transférer une partie des caractéristiques du modèle OLAN sur A3. Nous pensons que certaines idées contenues dans OLAN seraient très intéressantes dans le cadre de A3 et de la reconfiguration.

2.5.2 Inconvénients

- L'environnement A3 ne permet de créer que des agents à un seul flot d'exécution.
- Le modèle de communication utilisé par A3 est asynchrone, ce qui pose des problèmes supplémentaires par rapport aux modèles synchrones. En effet, avec le modèle synchrone de type «RPC»[18] les requêtes sont effectuées une à une et de manière synchrone, cela facilite le vidage des canaux de communication car un agent sait s'il a une requête en cours de traitement ou pas.
- L'environnement A3 n'offre pas de possibilité pour intégrer du logiciel.
- Les utilisateurs sont contraints à développer sur la plateforme A3.
- L'environnement OLAN quand à lui s'appuie sur un composant configurateur pour déployer une configuration. Un des problèmes de ce modèle est que la configuration est relativement statique, c'est à dire qu'un composant ne doit pas créer de nouveaux composants ou transmettre une référence sur un composant.
- Le configurateur est un point faible de l'architecture. Si le configurateur tombe en panne, aucune application ne peut plus être déployée ou reconfigurée.

3 Conclusion

Nous allons essayer de tirer partie des avantages des deux modèles en essayant de minimiser leurs inconvénients.

- Un des inconvénients des modèles OLAN et AAA est que la description d'une application est statique, c'est à dire que des agents ne peuvent pas être créés et interconnectés dynamiquement. Cet inconvénient est résolu par la reconfiguration dynamique d'application répartie.

- Le modèle OLAN permet d'intégrer des logiciels existants. Cette propriété apporte des solutions d'ouverture vers l'extérieure au modèle AAA.
- Le modèle AAA fournit des propriétés de fiabilisation des mécanismes de déploiement, ces propriétés peuvent être utilisées dans le cadre de la fiabilisation du configurateur.

Chapitre III.

Mécanisme de Reconfiguration dans

l'Environnement A3

1 Introduction

Rappel

Nous définissons notre algorithme dans le contexte de l'environnement A3-OLAN (Agent Anytime Anywhere)

Dans cet environnement, les applications sont décrites comme un **assemblage d'agents interconnectés** au travers d'un réseau de communication. La description d'une application (que l'on nomme **configuration** d'une application) s'effectue grâce à un ADL (Architecture Description Language), le langage **OCL**. A partir de cette description, l'application peut être déployée automatiquement grâce à des services systèmes. La programmation des agents peut être séparée de celle des communications, ceci permet d'améliorer la **réutilisation** d'un agent et de séparer deux niveaux de conceptions:

- Le niveau de structuration de l'application en terme d'interconnexion d'agents.
- Le niveau de mise en œuvre des agents.

L'agent configurateur est la clé de voûte des services systèmes de déploiement. C'est un agent particulier qui connaît la configuration d'une application, il a pour rôle d'installer et de créer les agents sur les sites distants selon les spécifications de placements contenues dans la configuration. De plus, il doit mettre en place les interconnexions entre agents.

Ce chapitre présente les algorithmes nécessaires pour la reconfiguration dynamique d'applications réparties et compare notre approche à celles étudiées dans l'état de l'art.

La conception d'une solution pour la reconfiguration dynamique nécessite d'examiner les points suivants :

- **Le processus de validation** : Le processus de validation doit permettre la vérification de la description d'une configuration/reconfiguration. Il existe deux types de vérification ;
 - *La vérification syntaxique* qui vérifie des règles de construction syntaxique comme, par exemple, vérifier pour une interconnexion qu'un service fourni correspond bien à un service requis (cf terminologie OLAN).
 - *La vérification sémantique* qui vérifie que les changements décrits ont effectivement un sens en fonction de la sémantique des composants et de l'application.
- **La gestion des ressources** : La gestion des ressources est l'ensemble des règles définissant le comportement à adopter lors d'une reconfiguration comme le comportement à adopter envers les fichiers ouverts, les sockets ... En effet, supposons qu'un agent manipule un fichier lorsque survient une reconfiguration,

que l'agent possède à cet instant un descripteur de fichier qui a un certain état, que faire du fichier si cet agent est déplacé ? Le configurateur doit-il également déplacer le fichier ? Imaginons que le fichier soit partagé, les agents doivent-ils fermer les fichiers à la fin d'une réaction et les ouvrir au début d'une réaction ? ... Le configurateur doit pouvoir répondre à ces questions.

- **L'algorithme de reconfiguration** : Il définit les mécanismes de bas niveau nécessaires pour la reconfiguration.

Dans cette section, nous allons présenter les algorithmes utilisés pour la reconfiguration. Les autres points (comme le processus de validation) ne seront pas abordés dans le DEA pour des raisons de temps.

2 Primitives de changements proposées

L'étude de la littérature[18][15][20] a permis de dégager quatre primitives de base pour les changements. Ces primitives sont :

- L'ajout et le retrait de composant.
- L'ajout et le retrait de connexion.

Ces primitives de changements sont très fines et un certain nombre de primitives plus complexes, comme par exemple le déplacement d'un composant, peut en être dérivé. Néanmoins, nous avons décidé de proposer un certain nombre de primitives de plus haut niveau (comme par exemple la migration d'agent) car nous pensons que certaines primitives de reconfiguration pourraient avoir une mise en œuvre différente pour des raisons d'efficacité. De plus, le fait d'avoir des primitives plus spécialisées facilite le processus de validation.

2.1 Les primitives de changements de structures

Les primitives de reconfiguration que nous proposons sont :

- La primitive **AddAgent**.
- La primitive **BindAgent**.
- La primitive **RebindAgent**.
- La primitive **MoveAgent**.
- La primitive **CloneAgent**.

2.1.1 Les primitives AddAgent et BindAgent

La primitive **AddAgent** permet d'ajouter un agent dans une configuration et la primitive **BindAgent** permet de connecter deux agents entre eux.

Ces primitives sont utilisées lors de la **configuration** d'une application. Néanmoins, elles doivent aussi pouvoir être utilisées lors de la reconfiguration. En effet, il doit être possible **d'ajouter dynamiquement** des agents à une application en cours d'exécution et **lier dynamiquement** des agents entre eux.

L'ajout d'un agent et la connexion de cet agent à une application existante ne pose pas de problème car ces primitives ne modifient pas les agents et les interconnexions existantes. Les agents sont simplement ajoutés à la configuration, ils ne perturbent pas l'application.

La figure suivante illustre l'ajout d'un agent « producteur » dans une application en cours d'exécution.

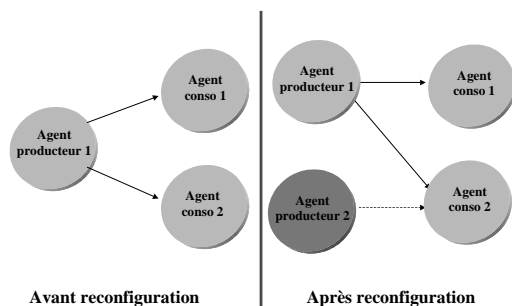


Figure III-1 : Exemple d'ajout d'un agent et d'une connexion

2.1.2 La primitive RebindAgent

La primitive de reconfiguration **RebindAgent** permet de modifier, à l'exécution, une interconnexion entre deux agents dans le but de changer le destinataire d'une connexion existante.

Nous différencions la primitive **BindAgent** de la primitive **RebindAgent** car la primitive **RebindAgent** est plus complexe puisqu'elle modifie une connexion existante qui peut éventuellement être en cours d'utilisation.

Pour que le **configurateur** puisse effectuer cet ordre, il doit pouvoir modifier **dynamiquement** les liaisons entre deux agents. De plus il est nécessaire que le canal de communication représentant la connexion à modifier soit vide sans quoi la propriété de causalité ne peut pas être préservée.

L'exemple suivant montre un exemple d'utilisation de la primitive Rebind pour adapter l'utilisation d'agents filtres par des agents producteurs.

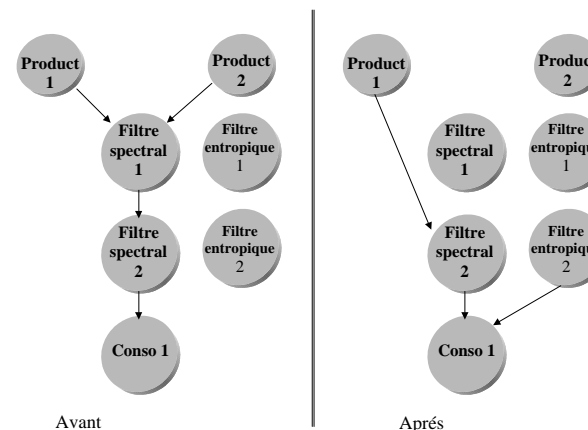


Figure III-2 : exemple d'utilisation de la primitive rebind agent pour gérer des filtres

2.1.3 La primitive CloneAgent

La primitive **CloneAgent** permet de dupliquer un agent. Le nouvel agent créé est la réplique **exacte** de l'agent dupliqué. Cette primitive est fournie pour faciliter principalement des opérations de répartition de charge.

Pour cloner un agent, son état doit être copié. Pendant cette copie il faut que l'état de l'agent ne soit pas modifié. Pour pouvoir utiliser cette primitive, tous les canaux de communication représentant des connexions orientées vers l'agent à cloner doivent donc être vides pendant que l'état de l'agent est recopié.

Cette primitive de reconfiguration nécessite que le **configurateur** puisse dupliquer l'état d'un agent à distance.

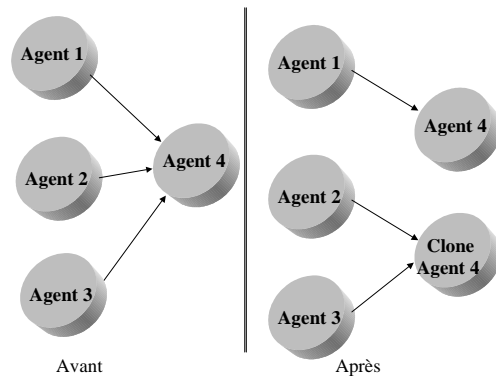


Figure III-3 : exemple d'utilisation de la primitive CloneAgent dans le but de répartir la charge de l'agent 4.

2.1.4 La primitive RemoveAgent

La primitive *RemoveAgent* permet de supprimer un agent dans une application en cours d'exécution.

Pour pouvoir supprimer un agent, le *configureur* doit être sûr qu'il n'y a plus de notification en transit à destination de l'agent à supprimer.

Le configureur doit donc avoir la possibilité de vider ces canaux.

2.2 Les primitives de changements de géométrie

2.2.1 La primitive MoveAgent

La primitive *MoveAgent* doit permettre en cours d'exécution de déplacer un agent sur un autre site. Cette primitive peut être utilisée pour déplacer certains agents de l'application sur de nouvelles machines plus performantes ou pour des besoins d'administration.

Pour ce type de changement, les connexions dirigées vers un agent à déplacer doivent être vides sinon des messages peuvent arriver sur un site alors que l'agent destinataire a été déplacé.

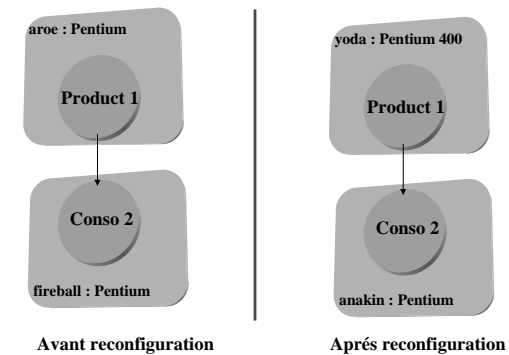


Figure III-4 : exemple d'utilisation de la primitive move pour faire migrer une application sur une plateforme plus performante

3 Gestion de la cohérence de l'application

Une application reconfigurée doit rester dans un état cohérent, c'est à dire qu'une application doit continuer à s'exécuter normalement et à fournir des résultats corrects au vue des primitives de reconfiguration utilisées.

Comme nous l'avons expliqué précédemment, chaque primitive de reconfiguration pose un certain nombre de problèmes qui peuvent nuire à la cohérence de l'application.

Prenons le cas du déplacement d'un agent, les canaux représentant les connexions dirigées vers un agent à déplacer doivent être vides pour éviter que des messages en transit arrivent sur machine alors que l'agent destinataire a déjà été déplacé.

Pour que cette cohérence soit assurée, le configureur doit attendre que les agents à reconfigurer soient dans un *certain état*. Dans l'exemple précédent, le configureur doit attendre que l'agent à déplacer soit dans un état à partir duquel il peut déduire que les canaux dirigés vers cet agent sont vides.

La base de notre algorithme de reconfiguration réside dans la définition de ces états abstraits et de leurs propriétés associées. *L'état abstrait* d'un agent est un état qui peut être calculé par le *configureur* et à partir duquel des propriétés relatives à l'état concret de l'agent peuvent être déduites (par exemple un agent ne recevra plus de notification, un agent n'enverra plus de notification ...).

Nous nous sommes inspirés de CONIC [17] pour définir les *état abstraits* nécessaires permettant d'effectuer une reconfiguration de manière cohérente.

3.1 Etat abstrait d'un agent

Un agent peut avoir trois états abstraits différents :

□ Etat *actif* : Un agent est dans l'état *actif* s'il peut générer des événements, en recevoir et les traiter.

□ Etat *passif* : Un agent dans l'état passif peut recevoir des événements et les traiter mais l'agent n'envoie plus de notification d'événement.

Lorsqu'un agent est dans l'état passif, il est possible de *modifier les connexions* dont il est initiateur (i.e. il est possible d'effectuer l'opération **Rebind**).

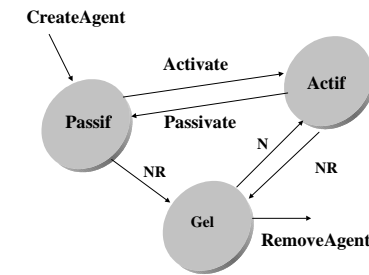
□ Etat *gelé* : Un agent dans l'état gelé n'est pas en train de réagir à une notification **qui n'est pas une notification de reconfiguration** et il ne recevra plus de notifications qui ne soient pas une notification de reconfiguration.

Lorsqu'un agent est dans l'état gelé, il est possible d'effectuer sur lui les ordres de configuration **Move**, **Clone** et **Remove**. En effet, comme les canaux de communication dirigés vers cet agent sont vides et le resteront :

- **Le déplacement** d'un agent peut s'effectuer sans risque que des notifications en transit sur un site arrive alors que l'agent a déjà été déplacé.
- **Le clonage** d'un agent peut s'effectuer sans problème car l'état de l'agent est figé.
- **La suppression** d'un agent peut être faite car l'agent ne sera plus nécessaire pour réagir à des événements étant donné qu'il n'en recevra plus.

3.2 Transitions

La figure suivante illustre les différentes transitions d'états abstraits possibles pour un agent. Les agents n'ont pas la maîtrise de leur état, seul le configurateur a le pouvoir de faire changer un agent d'état car c'est lui qui connaît la configuration d'une application et qui connaît les changements d'état susceptibles d'être valides selon la primitive de reconfiguration. De plus, le configurateur gère l'exécution des ordres de reconfiguration. Notre algorithme de reconfiguration assure que lorsqu'un agent réagit à une notification de reconfiguration, il est dans l'état gelé.



NR : Notification de
N : Notification autres que notification de

Figure III-5 : transition d'état d'un agent

3.3 Algorithme *Passivate*

Rappel :

- *L'ordre de réception des notifications entre deux agents est conforme à l'ordre causal d'émission si:*
 - *Un agent A émet successivement les notifications n1 et n2, alors aucun agent ne recevra n2 avant n1.*
 - *Un agent A1 émet une notification n2, et si un agent après avoir reçu n1, émet une notification n2, alors aucun agent ne recevra n2 avant n1.*
- *Pendant qu'un agent est dans l'état passif, il ne doit pas émettre de notification.*

L'algorithme *passivate* décrit la façon dont un agent va réagir à un ordre « *passivate* » venant du configurateur. Cet algorithme a pour rôle de faire passer un agent de l'état *actif* vers l'état *passif*.

Le problème de cet algorithme est que l'agent peut continuer à réagir aux événements qu'il reçoit et donc, il peut être amené à envoyer des notifications pour achever la réaction à un événement reçu.

Lorsqu'un agent reçoit l'ordre « *passivate* » du configurateur il passe **immédiatement** dans l'état passif et renvoie un acquittement au configurateur. A partir de cet instant, toutes les notifications envoyées par l'agent seront retardées dans la file d'attente en attendant la prochaine activation.

Ce mécanisme permet de bloquer les communications sortantes d'un agent tout en lui permettant de réagir de manière cohérente aux événements qui surviennent.

Notre solution utilise l'*ordre causal* de réception des notifications qui est fourni par le *bus A3*. Notre algorithme **retarde** les notifications envoyées par un agent dans l'état *passif* grâce à une file d'attente fifo présente dans l'agent lui même. Lors de l'activation d'un agent *passif*,

la propriété de causalité nous assure que l'ensemble des notifications seront reçues dans le même ordre.

La figure suivante illustre un ordre *passivate*. Le *configurateur* veut rendre passif l'agent 3, il envoie donc une notification « *passivate* » à l'agent 3. Lorsque l'agent 3 reçoit cet ordre, il devient *passif*. Dès lors, toutes les notifications envoyées par l'agent 3 seront retardées dans la file d'attente. Les agents 1 et 2 peuvent envoyer des événements à l'agent 3 et ces événements seront traités. Mais les notifications que l'agent 3 pourraient envoyer seront retardées.

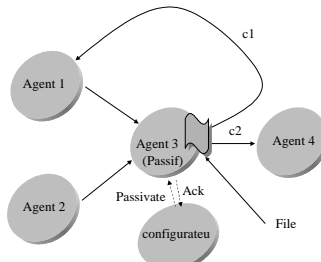


Figure III-6 : illustration de l'algorithme « passivate »

Preuve: Nous devons prouver que l'action de faire passer un agent dans l'état passif ne modifie pas la causalité entre notifications de l'application.

Imaginons le cas suivant : nous avons deux agents A1 et A2 reliés par une connexion *c1*, l'agent A1 est dans l'état passif et une notification *n1* est retardée dans la file d'attente.

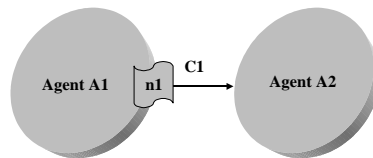


Figure III-8 : scénario choisi

Pour que la notification retardée *n1* casse la causalité des notifications, il faut qu'une notification *nx* dépendante causalement de *n1* arrive à l'agent 2 avant *n1*. Pour que cette notification *nx* soit dépendante de *n1*, il doit exister une chaîne (éventuellement vide) de notifications causalement dépendantes de *n1*. Il doit donc exister une notification *n2* émise par l'agent A1 qui soit causalement dépendante de *n1*. Or cela est impossible car si *n1* est retardée dans la file d'attente, toutes les notifications causalement dépendante de *n1*, émises par A1, sont aussi retardées dans la file d'attente suivant l'ordre fifo d'émission.

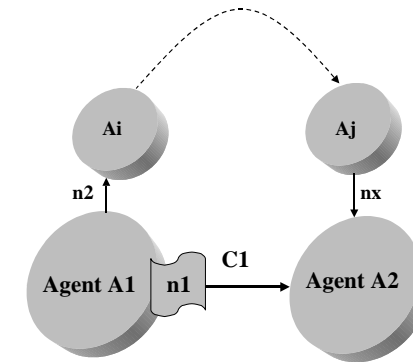


Figure III-9 : illustration de la preuve de la propriété 4

Algorithme détaillé

Soit **A** un agent

Config l'agent configurateur

Etat un attribut de l'agent A représentant l'état abstrait de A.

PassiveAckNot la notification d'acquiescement de l'ordre « *passivate* »

ActiveAckNot la notification d'acquiescement de l'ordre « *activate* »

SendTo(To,notification) une primitive de bas niveau permettant d'envoyer une notification vers l'agent To. Cette primitive ne doit pas être utilisée par un utilisateur.

PostTo(To,notification) une primitive permettant d'envoyer une notification vers l'agent To.

File_A la file d'attente utilisée pour retarder les notifications

- **Sur réception d'un ordre « *passivate* » :**

```

Passivate_react()
{ Etat :=Passif ;
  SendTo(Config, passiveAckNot) }
    
```

- **Sur réception d'un ordre « *Activate* » :**

```

Activate_react()
{Etat :=Actif ;
  Pour tout tuple <To,notif> ∈ File_A
  { SendTo(To,notif) //To doit être réévaluer }
  SendTo(Config, ActiveAckNot) }
    
```

- Sur envoi d'une notification Notif par l'agent A (utilisation de la primitive Post To):

```

PostTo(To,Notif)
{
  Si Etat=Passif insérer n dans File_A ;
  SendTo(To,Notif) ;
}
    
```

3.4 L'algorithme de reconfiguration

L'algorithme de reconfiguration s'appuie sur la définition des états abstraits et des propriétés associées. Comme vu précédemment, pour effectuer un ordre de reconfiguration, le configurateur doit faire passer des agents dans l'état *gelé* et pour cela, il doit aussi faire passer des agents dans l'état *passif*.

On appelle ensemble **CPS** (Change Passive Set) l'ensemble des agents à faire passer dans l'état *passif* pour geler les agents nécessaires à la reconfiguration. Cet ensemble peut être déterminé en fonction de l'ordre de configuration et de la connaissance de la *configuration* de l'application par le configurateur.

Le configurateur détermine l'ensemble **CPS** en fonction de l'agent à geler. Un agent appartient à **CPS** s'il a une connexion vers l'agent à geler ; cette information est connue du configurateur grâce à sa connaissance de la configuration.

Algorithme de reconfiguration

```

{
  Pour chaque ordre de reconfiguration
  {
    A/ Le configurateur détermine l'agent à geler suivant l'ordre de reconfiguration.

    B/ Le configurateur détermine l'ensemble CPS en fonction de l'agent à geler. Un agent appartient à CPS s'il a une connexion vers l'agent à geler, cette information est connue du configurateur grâce à sa connaissance de la configuration.

    C/ Le configurateur envoie un ordre « passivate » à tous les agents appartenant à CPS et attend les acquittements correspondants. Cette étape a pour but de rendre passif tous les agents de CPS. Dès lors, l'agent à geler est en cours de gel.

    Un agent A est en cours de gel lorsque tous les agents qui ont une connexion vers A sont passifs mais qu'il existe encore des notifications en transit à destination de A.

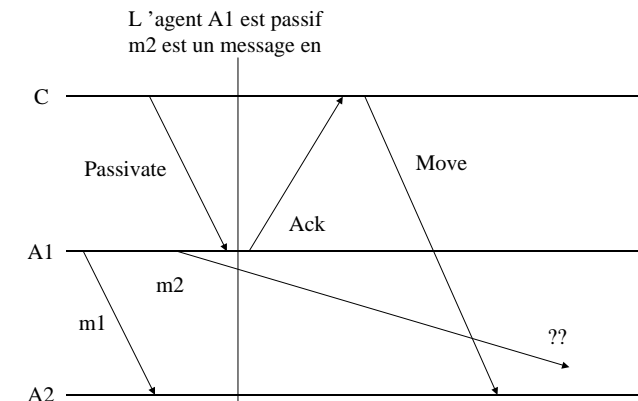
    D/ Le configurateur envoie un ordre de reconfiguration vers l'agent gelé (qui est l'agent à reconfigurer). Lors de l'étape précédente, l'agent à geler est en cours de gel mais, grâce à la propriété de causalité des notifications, nous sommes assurés que l'ordre de reconfiguration ne sera reçu par l'agent à geler qu'il est effectivement dans l'état gelé.
  }
}
    
```

```

}
E/ Le configurateur envoie un ordre « activate » à tous les agents passifs pour les réactiver.
}
    
```

3.4.1 Preuve de l'algorithme

Hypothèse: L'application, illustrée dans le diagramme temporel suivant, est composée de deux agents A1 et A2 et il existe une connexion de A1 vers A2. Le configurateur veut déplacer l'agent A2 sur un autre site. Pour cela, il faut que l'agent A2 soit gelé et l'agent A1 passif. Une fois l'agent A1 *passif*, l'agent A2 est *en cours de gel*.



Il faut prouver que le message **Move** envoyé par le configurateur sera reçu par A2 lorsqu'il est effectivement dans l'état gelé. Une fois la notification **Move** arrivée à l'agent A2, toutes les notifications en transit vers l'agent A2 sont arrivées.

Nous voyons sur la *figure III.8* que l'émission du message **Move** dépend causalement de l'émission du message **Ack**, lequel dépend de tout message en transit émis antérieurement par A1 et donc, dans notre cas, de l'émission du message **m2** (car les messages en transit sont expédiés avant l'émission du message **Ack**).

D'après le mécanisme de causalité défini par le modèle A3, le message **m2** et l'ensemble des messages en transit arriveront avant le message **Move**.

On peut en déduire qu'après l'arrivée du message **Move**, le canal de A1 vers A2 est vide et le restera jusqu'à activation. En effet :

- Les messages en transit ont été envoyés avant que l'agent A1 soit dans l'état passif
- Une fois dans l'état passif, l'agent n'envoie plus de notification au bus (elles sont retardées d'après la définition de l'état passif).

3.4.2 Propriétés associées à l'algorithme de reconfiguration

Propriétés 1 : Un agent peut toujours passer dans l'état *passif* si le configurateur le lui ordonne.

Preuve : Pour être dans l'état passif, un agent ne doit plus déposer de notifications sur les canaux de communications. Or l'algorithme met en attente toutes les notifications que l'agent pourrait envoyer et le *bus A3* nous garantit que l'ordre « *passivate* » envoyé par le reconfigurateur arrivera bien à l'agent destinataire, et ce, malgré l'occurrence de pannes : c'est la propriété de fiabilité de l'environnement **A3**.

Propriété 2 : L'algorithme de gel assure que l'agent à geler finira par passer dans l'état gelé.

Preuve : L'algorithme de gel rend passifs les agents qui ont une connexion vers l'agent à geler.

- D'après la *propriété 1*, les agents finiront par devenir passifs. Lorsque les agents sont passifs, les notifications sont retardées.
- Le *bus A3* assure que les notifications en transit seront reçues.
- Une fois que toutes les notifications en transit sont reçues et que les agents qui ont une connexion dirigée vers l'agent à geler sont passifs, l'agent passera effectivement dans l'état gelé.

4 Discussions

Dans cette section, nous tenterons de prendre du recul par rapport à notre algorithme de reconfiguration et de le comparer avec l'algorithme de **CONIC** qui s'approche le plus de notre solution. Comme les mesures de notre algorithme ne sont pas actuellement disponibles, nous essayerons d'effectuer une première comparaison en terme de nombre de composants passifs et gelés par les deux algorithmes suivant l'ordre de reconfiguration.

4.1 Evaluation de l'algorithme

□ Points forts

- Notre algorithme évite toute incohérence lors d'une reconfiguration.
- Notre algorithme est construit au dessus du bus A3 ce qui lui procure :
 - ♦ **Une propriété de fiabilité** : La propriété de fiabilité permet de fournir un algorithme de reconfiguration robuste (i.e. tolérant aux pannes).
 - ♦ **Une propriété de diffusion causale des notifications** : La propriété de causalité facilite l'algorithme de reconfiguration car il permet de retarder des notifications tout en maintenant l'ordre de réception des notifications cohérents (par rapport à l'ordre causal). De plus, le

configurateur peut envoyer une notification de reconfiguration vers un agent *en cours de gel* contrairement à **CONIC** où le configurateur doit attendre que l'agent ait effectivement gelé avant d'envoyer une notification de reconfiguration.

- Le modèle de communication asynchrone permet d'éviter le problème des sessions dépendantes de **CONIC** car un agent n'est pas bloqué par l'attente d'une réponse à une requête.
- L'algorithme de reconfiguration peut être porté sur d'autres environnements disposant des prérequis suivants :
 - ♦ Existence d'un mécanisme de *diffusion causale*.
 - ♦ Connaissance par un agent externe de la configuration des applications.

□ Points faibles

- Notre algorithme nécessite un mécanisme de diffusion causale des notifications. Ce mécanisme coûteux n'est pas forcément présent dans tous les environnements de programmation. Le mécanisme de causalité est un prérequis au portage de l'algorithme sur un autre environnement.
- Utiliser un mécanisme de communication asynchrone complexifie grandement les mécanismes de traitement d'erreur car lorsqu'une erreur est notifiée, le contexte dans lequel s'est produit l'erreur a peut être changé.
- Le mécanisme de reconfiguration se base sur un agent configurateur qui a une connaissance de la configuration de l'application. La configuration définit les agents participant à l'application, leurs contraintes de placement et les interconnexions entre les agents. Ces informations sont définies *statiquement* lors de la configuration de l'application et sont modifiées par une reconfiguration. Par conséquent, les agents ne doivent pas transmettre des références d'agents et ne doivent pas créer d'agent de leur propre initiative faute de quoi la connaissance de la configuration maintenue dans le configurateur ne correspond plus à la configuration réelle.
- Notre algorithme nécessite l'envoi de notifications de reconfiguration. Actuellement ces notifications sont acheminées de manière identique aux autres notifications. Par conséquent, si dix notifications sont en attente de traitement avant une notification de reconfiguration, cette dernière ne sera traitée qu'après l'exécution des réactions des dix notifications. Une solution serait d'affecter une priorité aux notifications de reconfiguration. Ce point sera rediscuté dans la propositions des perspectives associées à l'algorithme.

4.2 Performance de l'algorithme

Comme actuellement nous ne disposons pas de mesure de notre algorithme, nous essayerons d'effectuer une première comparaison en terme de nombre de composants passifs et gelés par les deux algorithmes suivant l'ordre de reconfiguration.

□ Définitions

Inputs: les inputs d'un agent n sont tous les agents qui ont une connexion vers l'agent n . Le multi-ensemble I_n représente les inputs de l'agent n . On note $|I_n|$ le nombre d'élément de I_n .

□ Application test

Notre application test représente une application répartie constituée d'un ensemble d'agents gérant une bibliothèque répartie. Il existe différents types d'agent :

- L'agent *inter-central* gérant l'ensemble des bibliothèque d'un pays.
- L'agent *central* gérant l'ensemble des bibliothèque d'une ville.
- L'agent *local* gérant une bibliothèque locale à une ville.

L'interconnexion de ces différents agents est la suivante :

- Les agents intercentraux sont fortement connectés entre eux.
- Chaque agent intercentral est fortement connecté aux agents centraux du pays qu'il représente.
- Les agents centraux sont fortement connectés entre eux.
- Chaque agent central est connecté à tous les agents locaux de la ville qu'il représente.
- Chaque agent local au central gérant la ville où la bibliothèque réside.

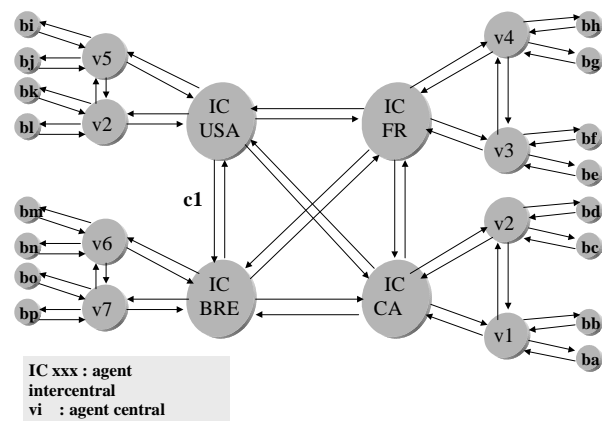


Figure III- : Architecture de l'application test

□ Primitives Rebind

La primitive rebind permet de modifier une connexion existante. Une utilisation de cette primitive serait de reconnecter les connexions dirigées vers l'intercentral du Brésil (tombé en panne) vers un intercentral de remplacement.

Dans le cadre d'utilisation de notre exemple, on effectue une modification de la connexion **c1** à l'aide de la primitive « **Rebind** ». Les points suivants évaluent les performances de l'algorithme de CONIC et du nôtre en terme de nombre d'agents gelés et passifs.

Pour faciliter cette évaluation, on considérera que chaque agent intercentral et central a cinq connexions dirigées vers chacun d'entre eux.

- CONIC

- L'agent **IC_USA** doit être gelé et ces $|I_1|$ inputs doivent être passifs
- On a au total : $|I_1|$ agents passifs + **1** agent gelé.
- **Soit dans le cas de l'application test, $|I_1|=n$, donc n agents passifs + 1 agent gelé**

- A3

Pour n inputs : 1 agent passif

□ La primitive Move

La primitive Move permet de déplacer un agent existant. Une utilisation de cette primitive serait de déplacer l'agent intercentral du Brésil résidant sur une machine A vers une machine B plus performante. Les deux points suivants évaluent ces primitives pour l'algorithme de CONIC et le nôtre.

- CONIC

Il faut geler l'agent **1** et ces $|I_1|$ inputs ($i_{11}, i_{12}, \dots, i_{1n}$), soit $|I_1|+1$ agents à **geler**. Cela nécessite de faire passer $(|I_{11}|+|I_{12}|+ \dots +|I_{1n}|)$ agents dans l'état **passif**.

- **Soit dans le cadre de notre application : n^2 agents passifs et $n+1$ agents gelés.**

- A3

n agents passifs + 1 agent gelé

Cette évaluation permet de conclure que notre algorithme perturbe moins l'application en terme d'agents à rendre passif et d'agents à geler. Ceci laisse présager que notre algorithme aura de meilleures performances d'exécution que CONIC car l'algorithme de

CONIC nécessite plus de temps d'attente étant donné qu'il y a plus d'agent à geler qu'avec notre algorithme.

Notre algorithme a la possibilité de geler moins d'agents grâce au mécanisme de diffusion causale. Cette section nous a permis de présenter une application supplémentaire au sein de laquelle les mécanismes de reconfiguration sont utiles.

Chapitre IV.

Support pour la Reconfiguration dans l'Environnement A3

1 Introduction

L'environnement de développement d'application à base d'agents doit permettre de décrire au travers d'un outil graphique l'architecture d'une application, i.e. les agents nécessaires au fonctionnement de l'application et le schéma de communication entre ces agents. Cet environnement se compose d'un *outil graphique*, d'un agent *configurateur* et d'un agent *administrateur*.

L'*outil graphique* facilite la description des interconnexions d'agents. Cet outil effectue diverses vérifications quant à la validité de l'assemblage d'agents et pilote l'installation de l'application sur l'environnement distribué d'exécution. Cette installation est pilotée depuis l'outil graphique et repose sur un ensemble de services contenus dans la machine à agents. Parmi ces services, nous pouvons distinguer :

- L'*agent configurateur* qui a pour rôle d'installer les applications réparties en fonction d'une configuration donnée et de reconfigurer une application en cours d'exécution.
- L'*agent administrateur* qui permet de surveiller l'état des diverses machines à agents disponibles pour l'exécution.

La suite de cette section décrit la hiérarchie des classes d'agents à mettre en place pour répondre aux besoins de la reconfiguration ainsi que les principes de fonctionnement de l'outil graphique, de l'agent configurateur et de l'agent administrateur.

2 Architecture générale

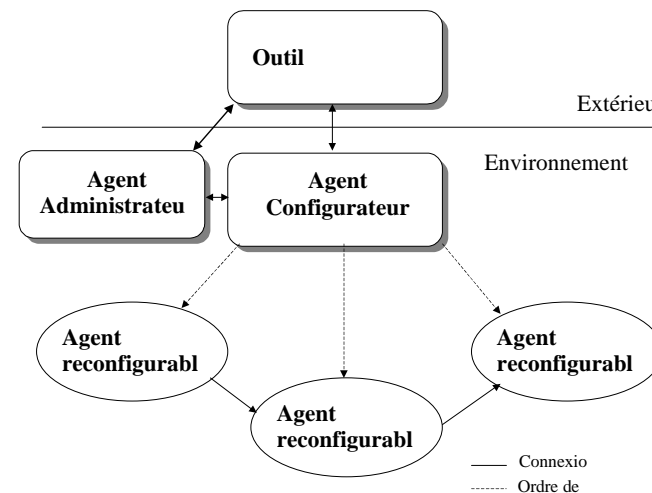


Figure IV-1 : architecture générale

3 Hiérarchie des classes d'agent utilisées

Les mécanismes de reconfiguration nécessitent que les agents réagissent d'une manière adéquate à certains ordres, comme par exemple « passivate ». La classe d'agent définie par les architectes de A3 ne correspond pas entièrement aux besoins de la reconfiguration. C'est pour cette raison que nous avons dû étendre les caractéristiques des agents A3 grâce au mécanisme d'héritage de java. Dans la suite, nous présentons la hiérarchie de classe permettant de répondre aux besoins de la reconfiguration.

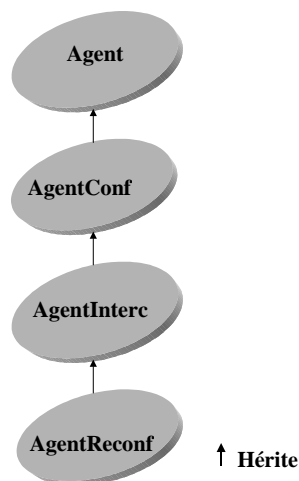


Figure III- : Graphe d'héritage des différentes classes d'agent utilisée

3.1 La classe Agent

Un agent est un objet passif, instance d'une classe qui hérite de la classe *Agent* ; la classe *Agent* définit le comportement commun à tous les agents. Le comportement réactif est mis en œuvre au moyen de la méthode *react* qui définit le comportement de l'agent lors de la réception d'une notification ; cette méthode est appelée par le moteur d'exécution. Chaque classe d'agent est, en particulier, caractérisée par l'ensemble des réactions qu'elle traite dans la méthode *react*. Les agents sont des objets persistants : les données qui composent leur état ont une durée de vie indépendante de toute exécution. Un agent peut être créé automatiquement sur un site distant grâce à la méthode *deploy*.

Cette méthode est intéressante pour le configurateur étant donné qu'il sera amené à créer des agents à distance pour la phase de déploiement. Néanmoins, la classe agent ne fournit aucune méthode permettant de définir les connexions entre deux agents ; les connexions sont câblées directement dans le code des agents.

3.2 La classe Agent Configurable

Les agents configurables sont des agents au modèle de programmation plus évolué. Ils fournissent les primitives de bas niveau permettant de modifier dynamiquement l'état ou la localisation d'un agent.

Parmi ces primitives, les agents configurables fournissent une primitive de duplication. C'est une fonction des agents configurables qui permet de créer un « clone », i.e. un autre agent de même classe et d'état identique au moment de la prise en compte de la demande de duplication. L'état contient toutes les variables publiques ou privées de l'objet.

Le fonctionnement s'appuie sur la réaction provoquée par certaines notifications particulières comme les notifications de configuration. Il existe actuellement deux types de notifications : *SetField*, qui positionne la valeur d'un champ (au sens Java du terme) et *DupRequest*, qui demande le clonage de l'agent sur un site spécifique. Une des particularités de cette classe est que les opérations « duprequest » et « setfield » sont asynchrones. L'initiateur ignore si l'exécution de l'ordre (duprequest ou setfield) s'est bien passée. La classe Agent configurable étend la classe agent pour permettre d'interconnecter deux agents à distance et pour permettre la duplication d'un agent.

3.3 La classe Agent Interconnectable

Cette classe d'agent est celle utilisée pour assembler des agents selon le modèle de configuration **Olan**. Un agent interconnectable est un agent configurable dont un rôle identifie les services ou opérations requis d'un autre composant et qui ne fournit des réactions qu'à un certain nombre d'agents clairement identifiés. En d'autres termes, un agent interconnectable est un agent configurable qui ne répond qu'à un ensemble de notifications provenant d'un ensemble d'agent particulier. Nous pensons que cela permet de sécuriser le modèle car seules les notifications autorisées sont exécutées par les agents concernés.

La seconde propriété de ces agents concerne la synchronisation des opérations de configuration. Les agents interconnectables, lorsqu'ils traitent une notification de configuration, envoient toujours un acquittement de réussite ou d'échec de l'opération à l'agent initiateur de la notification de configuration. De plus, ces agents ne réagissent à aucune autre notification tant qu'une opération de configuration est en cours, i.e. que l'acquittement n'a pas été envoyé. Ils ont ainsi un comportement synchrone par rapport à toute demande de configuration. L'intérêt de cette synchronisation est de faciliter la gestion des erreurs au niveau du client. Nous avons introduit cette classe entre la classe agent configurable et agent reconfigurable car nous pensons qu'elle a des propriétés suffisamment intéressantes (comme la synchronisation des opérations) pour en faire une classe à part. Cette classe pourra être utilisée pour dériver d'autres classes d'agent synchrone, interconnectable mais pas nécessairement reconfigurable. Techniquement, la duplication d'un agent s'effectue très facilement grâce à l'interface de « sérialisation » de Java. En effet, le mécanisme de « sérialisation » Java permet d'obtenir une représentation d'une instance d'un objet Java sous la forme d'une chaîne. Si l'instance sérialisée contient des références vers d'autres instances, ces instances sont aussi sérialisées.

3.4 La classe Agent Reconfigurable

Cette classe d'agent est celle utilisée pour reconfigurer des agents. Cette classe hérite de la classe interconnectable car les agents reconfigurables doivent manipuler la notion de rôle et avoir un comportement synchrone vis à vis des opérations de configuration. De plus, les agents reconfigurables doivent pouvoir réagir correctement aux notifications de reconfigurations « activate » et « passivate » (voir chapitre III) ainsi que la méthode PostTo qui est utilisée pour envoyer des notifications qui seront retardées si le composant est passif. Les agents reconfigurables ont donc la propriété de pouvoir passer dans l'état passif ou actif. Dans l'état actif, l'agent réagit normalement alors que dans l'état passif l'agent continue à recevoir des notifications et à les traiter mais les notifications envoyées par cet agent sont retardées jusqu'à activation. Pour retarder l'envoi des notifications, une file d'attente fifo est utilisée. Lors d'une activation, les notifications en attente dans cette file sont expédiées.

4 L'outil graphique de configuration

Cet outil offre à un utilisateur une interface graphique permettant de construire, de configurer et de reconfigurer des applications par assemblage de composants logiciels. Cet assemblage se fait selon la philosophie et les règles du langage de configuration OCL [LBel]. Cette description est traduite en ordres de configuration qui sont envoyés à un agent configurateur.

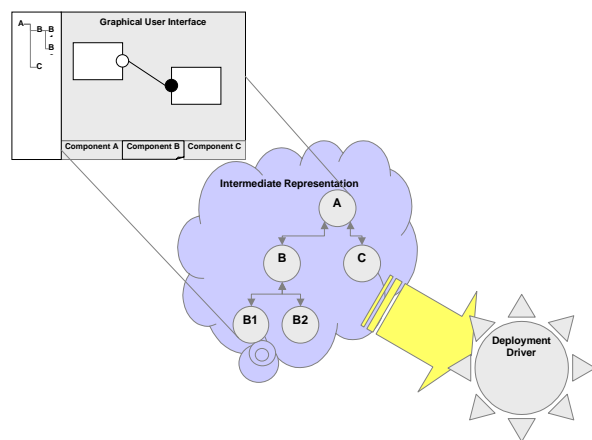


Figure IV-3. Architecture de l'Outil Graphique de Configuration

L'architecture de l'outil graphique est décomposée en trois parties : un module central qui gère la représentation intermédiaire d'une configuration d'application, un module de représentation, de construction et de manipulation graphique de cette représentation

intermédiaire, et un module de liaison avec les structures d'exécution pour assurer le déploiement de l'application.

La représentation intermédiaire est une structure de donnée qui représente la hiérarchie des composants utilisés par l'application. De manière simplifiée, cette hiérarchie est un arbre dont les nœuds sont des composites et les feuilles des composants primitifs (donc en particulier des agents). Il existe plusieurs types de relations au sein de cette hiérarchie : une relation de composition indiquant si un composant est inclus dans un autre composant, et une relation d'interconnexion qui établit l'existence d'une liaison entre des composants contenus dans un même composite. La représentation intermédiaire est le pivot de l'outil car c'est ce qui définit une configuration d'application, ce qui permet la représentation graphique et qui permet le déploiement de l'application.

L'interface utilisateur contient un ensemble de fenêtre permettant la navigation au sein de la structure de l'application. Cette navigation exploite la relation de composition de la représentation intermédiaire.

Le pilote de déploiement est un outil au double rôle. Il effectue la traduction de la représentation intermédiaire en un ensemble de commande que le support d'exécution d'une application à base d'agents AAA est capable d'exécuter pour effectuer le déploiement. Il contrôle aussi le canal de communication entre l'outil et la machine à configuration, garantissant le bon acheminement des commandes de déploiement pour une application donnée.

5 L'Agent configurateur

Dans notre architecture, l'agent configurateur a pour rôle d'exécuter les ordres en provenance d'un outil graphique de configuration. Ces ordres peuvent être de trois types :

- *Les ordres de configuration* : Les ordres de configuration doivent permettre l'installation, la création et l'interconnexion des agents participant à une application distribuée.
- *Les ordres de reconfiguration* : Les ordres de reconfiguration doivent permettre de modifier la configuration courante d'une application en cours d'exécution.

5.1 Analyse des services à fournir

Pour exécuter les ordres provenant de l'outil graphique, le configurateur doit mettre en œuvre les services généraux suivants :

- **Configuration** d'une application.
- **Reconfiguration** d'une application en cours d'exécution.

5.1.1 Le service de configuration d'une application

Les services de configuration doivent permettre l'installation à distance d'une application distribuée. Le déploiement d'une application est réalisé en transmettant au configurateur des ordres de configuration portant sur un ou plusieurs agents. Les agents sont désignés par un nom symbolique, cela permet à l'outil graphique de ne manipuler que des noms d'agent compréhensibles par l'utilisateur. La correspondance entre le nom symbolique d'un agent et son AgentId est maintenue dans le configurateur.

Les méthodes à fournir par le configurateur pour le déploiement sont :

- **La création d'un agent**

CreateAgent (String OCLName, int Site)

La création d'un agent sur un site donné s'effectue en transmettant au configurateur une requête de création définissant la classe de l'agent à créer et son état initial. Un problème de la création d'un agent est celui du passage des paramètres d'initialisation. Une solution aurait été de définir un formalisme permettant de transmettre les type de paramètres les plus communs. Néanmoins ce type de solution n'est pas générique et est très contraignant car il faut définir un formalisme commun. La solution que nous avons adoptée est de transférer le nom de la classe sous la forme d'une chaîne et les paramètres nécessaires comme un tableau d'objet. Un mécanisme générique de java nous permet de créer une instance de la classe initialisée avec le tableau d'objet. Cette solution a l'avantage d'être générique et de ne nécessiter aucun formalisme de passage de paramètres.

- **Interconnexion entre deux agents**

BindAgent(OCLName src, OCLName dest, CONNECT_NAME Cname)

Une fois les agents créés, le configurateur doit permettre des connecter deux agents par la mise en place d'une liaison (agent source, rôle) -> agent destinataire. Cette méthode a pour effet de mettre à jour le rôle de l'agent source pour qu'il référence l'agent destination.

La méthode *setfield* présente dans tous les agents *configurables* (et donc *reconfigurables* par héritage) est utilisée pour modifier le rôle de l'agent initiateur de la connexion. La méthode *setfield* s'appuie sur les méthodes de réflexivité de java qui permettent de modifier les champs d'une instance à l'exécution.

- **Affectation d'un attribut**

SetAttribute (String AttributeName, Object NewVal)

Les agents peuvent être initialisés lors de leur création. Néanmoins, il est intéressant de pouvoir initialiser un agent avec une valeur donnée alors que l'agent a déjà été créé. La méthode *setfield* est utilisée de la même manière que pour l'interconnexion des agents. Pour éviter des problème de sécurité liés à la protection des classes, seuls les attributs non privés peuvent être ainsi modifiés.

5.1.2 Le service de reconfiguration

Le service de reconfiguration d'une application doit permettre de modifier la configuration courante d'une application en cours d'exécution. La réalisation de ces services est grandement facilitée par les services fournis par la classe *d'agent reconfigurable*. En effet, la classe *agent reconfigurable* permet au configurateur de modifier les interconnexions des agents à distance et de dupliquer des agents.

Nous considérons au niveau du configurateur que les ordres contenus dans une session de reconfiguration ont déjà été approuvés par le processus de validation et ordonnés correctement.

Le configurateur doit fournir les services de reconfiguration suivants:

- **Modification des interconnexions entre deux agents**

RebindAgent (String OCLSrc, String OCLDest, String CONNECTName, String NewOCLDest)

Cette primitive permet de modifier la connexion *CONNECTName* reliant initialement les agents *OCLSrc* et *OCLDest*. Le nouveau destinataire d'une connexion est l'agent *NewOCLDest*.

- **Clonage d'un agent**

CloneAgent(String OCLSrc, String OCLClone, int site)

Cette primitive permet de dupliquer l'agent *OCLSrc* sur le site d'exécution *site*. L'agent clone a pour nom *OCLClone*.

- **Migration d'un agent**

MoveAgent(String OCLName, int Newsite)

La primitive *MoveAgent* permet de déplacer l'agent *OCLName* sur un autre site d'exécution : *NewSite*.

- **Supprimer d'un agent**

RemoveAgent(String OCLName)

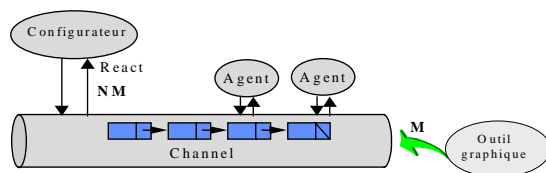
La primitive *RemoveAgent* permet de supprimer l'agent identifié par le nom symbolique *OCLName*.

5.2 Dialogue Client-Configurateur

5.2.1 Accès au configurateur

Dans notre architecture, le configurateur est un agent **A3**, il reçoit des ordres de configuration/reconfiguration en provenance de l'outil graphique. Le mécanisme de communication entre l'outil graphique et l'agent configurateur est particulier étant donné que l'outil graphique n'est pas un agent mais un composant extérieur à la machine à agent.

Le mécanisme de communication entre composants extérieurs et agents s'effectue par échange de messages. Lorsque le *bus logiciel* reçoit un message en provenance d'un composant extérieur, il le transforme en notification et le délivre à l'agent destinataire.



M : message M envoyé par le composant extérieur
 NM : message M transformé en notification NM par le bus logiciel

Figure IV-4: Acheminement d'un message provenant d'un composant extérieur vers l'agent configurateur.

5.2.2 Protocole de communication

Il existe différents scénarios pouvant être mis en place pour gérer le dialogue entre les clients et le configurateur. Le schéma de dialogue entre un client et un configurateur est du type requête/acquittement. Ce schéma n'est pas forcément synchrone mais il indique que le client recevra à un « moment donné » un acquittement correspondant à un ordre envoyé.

Premier scénario :

Les clients envoient des ordres un à un. Ils attendent d'avoir l'acquiescement d'un ordre avant d'en lancer un nouveau. Ce scénario est très simple, néanmoins le temps nécessaire à l'envoi des ordres est cumulatif. Cependant, la gestion des erreurs au niveau du client est facilitée car le client est directement « dans le contexte » de l'ordre ayant provoqué l'erreur. Dans ce scénario les ordres envoyés par un client (par exemple, un outil graphique) sont *synchrones*.

Deuxième scénario :

Ici, les clients envoient les ordres et reçoivent les acquittements de manière asynchrone. L'avantage de ce type de scénario est que le temps nécessaire à l'envoi des ordres est grandement amélioré car un client peut envoyer tous les ordres de configuration d'une session avant de recevoir le premier acquittement. Cependant, la gestion des erreurs au niveau du client est complexifiée.

Troisième scénario :

Les clients envoient tous les ordres d'une session dans un seul message et attendent de manière synchrone un compte rendu contenant tous les acquittements. Ce type de scénario devrait réduire le nombre de messages échangés sur le réseau. Il faut bien entendu que le message contenant tous les ordres ne soit pas trop important sans quoi les couches réseau sous-jacentes risquent de morceler le message ce qui réduirait l'intérêt de la solution. Cependant, la gestion des erreurs au niveau du client reste complexe.

Scénario choisi :

Comme l'installation d'une application est un événement transitoire de courte durée dans la vie d'une application, nous avons adopté le premier scénario pour faciliter la gestion des erreurs chez le client au détriment de la rapidité de la configuration/reconfiguration.

5.3 Disponibilité des services de configuration

Le comportement de l'agent configurateur est défini par un automate d'états finis qui décrit quels sont les ordres de configuration acceptés en fonction de l'état de configuration courant. Cet automate permet de vérifier, entre autres, que des ordres de reconfigurations ne sont pas exécutés pendant que l'application est en cours de déploiement.

Les transitions entre états sont telles que la réalisation des ordres de configuration est synchrone : le configurateur n'accepte de traiter un ordre de configuration que lorsque le traitement de l'ordre précédent est terminé.

Ce synchronisme facilite le traitement des erreurs au niveau des clients. Il assure de plus que l'application n'est pas lancée ou reconfigurée tant qu'elle n'est pas complètement déployée.

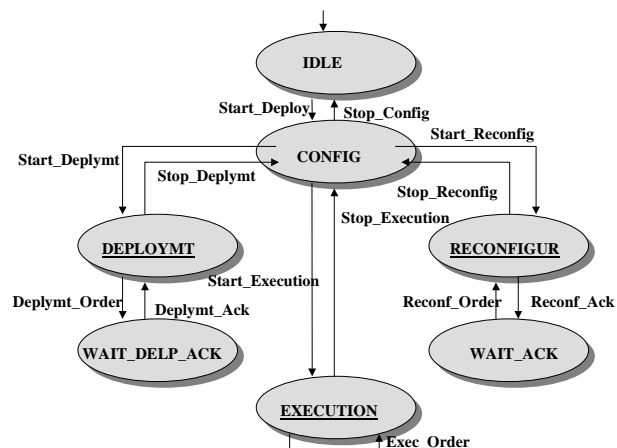


Figure IV-5 : Automate d'état gérant le configurateur

Les différents états de l'agent configurateur sont les suivants :

- L'état *Idle*
Dans cet état, l'agent configurateur est inactif.
- L'état *Deploy*
Dans l'état **DEPLOYMT**, le configurateur peut traiter des ordres concernant le déploiement de l'application. Ces ordres sont Createagent, BindAgent et SetAttribute.
- L'état *Config*
Un configurateur qui passe dans l'état **CONFIGUR** entre dans une *session* de configuration. Pendant cette session, le configurateur communiquera toujours avec le même interlocuteur externe jusqu'à la terminaison de la session (retour à l'état **IDLE**).
- L'état *Reconfigur*
Dans l'état **RECONFIGUR** le configurateur accepte les ordres de reconfiguration. Cet état n'est accessible que lorsque le configurateur est déjà passé dans l'état **DEPLOYMT**.
- L'état *Wait_Ack*
Cet état est utilisé pour gérer le synchronisme des ordres de reconfiguration.
- L'état *Wait_Depl_Ack*
Cet état est utilisé pour gérer le synchronisme des ordres de déploiement.
- L'état *Execution*

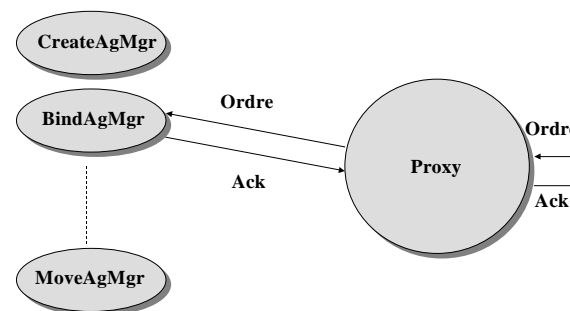


Figure IV-6 : organisation du configurateur en proxy et managers

Lorsque l'application est déployée et installée, elle n'est pas encore active. L'application devient effectivement active lorsque le configurateur est dans l'état **EXECUTION**.

5.4 Architecture détaillée du configurateur

5.4.1 Structure d'exécution du configurateur

Le configurateur est réalisé comme un agent A3 ce qui lui permet de communiquer (via des notifications) avec les agents de l'application à configurer. Cet agent est un agent particulier capable de dialoguer avec un composant externe.

La pièce centrale de l'agent est constituée d'un *proxy* qui a pour rôle :

- D'assurer le respect de l'automate d'états du configurateur.
- De réceptionner les différents ordres relatifs à la configuration de l'application en cours.
- De distribuer ces ordres au « *manager* » concerné.

Un *manager* n'est pas un agent (bien qu'il pourrai l'être) mais un objet java qui encapsule les méthodes nécessaires pour gérer l'ordre qu'il est censé traiter. Par exemple, le manager **BindMgr** est utilisé pour exécuter les ordres de connexions. Il existe donc un *manager* par type d'ordre fourni par le configurateur.

Etant donné que les *managers* sont des objets, il ne peuvent pas recevoir de notifications. Seul le *proxy* est un agent, c'est donc lui qui va recevoir les notifications d'acquiescement provenant des agents participant à la configuration ou de l'agent *factory* (l'agent *factory* participe au déploiement des agents). Le proxy va propager les notifications qu'il reçoit vers le manager géant l'ordre courant en appelant une méthode de ce manager.

Cette architecture a été adoptée pour sa modularité et ses facilités d'évolution. En effet, chaque ordre et le proxy peuvent évoluer indépendamment. De plus, l'ajout d'un ordre est indépendant de l'écriture du proxy. Enfin, le code du configurateur est éclaté en fonctionnalités faciles à comprendre et à faire évoluer.

5.4.2 Structures de données du configurateur

Le configurateur doit maintenir les informations concernant la configuration courante de l'application car ces informations sont exploitées par le mécanisme de reconfiguration. La configuration doit donc être conservée dans une structure spécifique. Cette structure est construite par le configurateur lors du déploiement de l'application. Pour faciliter la réutilisation et l'évolution de cette structure, nous l'avons conçu comme un objet de la classe configuration.

La classe configuration

La configuration est représentée par un graphe orienté dans lequel chaque sommet représente un agent et chaque arête orientée d'un sommet A vers un sommet B représente une connexion de A vers B.

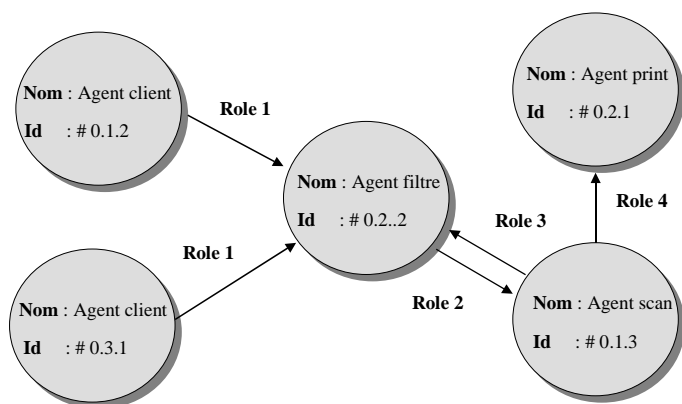
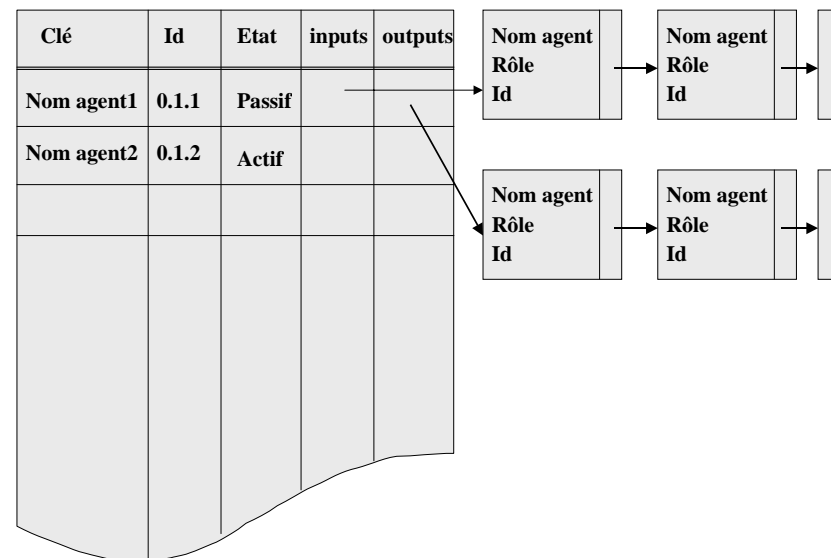


Figure IV-7 : exemple de configuration

Techniquement ce graphe est implémenté pour des raisons de performance sous la forme d'une table de hashcode dont la structure est la suivante :



La clé de la table de hashcode est le nom des agents, l'identificateur de l'agent est nécessaire pour que le configurateur puisse envoyer des notifications vers l'agent destinataire. L'état permet de savoir si un agent est actif ou passif. Enfin, les inputs et les outputs permettent de connaître les connexions associées à un agent. Cette structure contient un peu de redondance d'information mais elle permet des temps d'accès très rapides aux informations lors d'une reconfiguration.

5.5 Les managers

Un *manager* est un objet java qui encapsule les méthodes nécessaires pour gérer un ordre de reconfiguration. Chaque ordre de configuration/reconfiguration est associé à un manager spécifique. Certains managers sont détaillés dans les points suivants :

□ Le CreateManager

Le manager *CreateManager* a pour rôle de gérer l'ordre de création d'agent. Son fonctionnement s'appuie sur la *méthode de déploiement à distance* des agents. Lorsqu'un agent est créé et déployé, le manager met à jour la structure de donnée représentant la configuration.

□ Le BindManager

Ce manager permet d'établir une connexion entre deux agents. Pour interconnecter deux agents, le manager utilise la *méthode setfield* présente dans tous les agents reconfigurables. Cette méthode permet de modifier dynamiquement les références

vers un agent. La structure de données de la configuration est mise à jour de la façon suivante :

- Pour l'agent I initiateur de la connexion : l'agent destinataire est ajouté dans la liste des outputs de l'agent I.
- Pour l'agent D destinataire de la connexion : l'agent initiateur de la connexion est ajouté dans la liste des inputs de l'agent D.

□ *Le MoveManager*

Ce manager gère le déplacement d'un agent A. D'après l'algorithme de reconfiguration présenté dans le chapitre III, pour effectuer cet ordre, les inputs de l'agent A doivent être passifs et l'agent A doit être gelé.

Pour obtenir les informations relatives aux inputs de A, le manager utilise la structure de donnée représentant la configuration.

L'agent est déplacé en utilisant les méthodes de *duplication* et de *suppression* d'agents présentes dans tous les agents reconfigurables.

L'ordre de reconfiguration MoveAgent ne nécessite pas la modification de la structure de donnée représentant la configuration car la structure et la géométrie de l'application ne sont pas modifiées.

6 Conclusion

La mise en œuvre du support pour notre algorithme de reconfiguration dynamique a permis de montrer la faisabilité de l'algorithme et de commencer à l'expérimenter . Cette mise en œuvre rend possible l'expérimentation et l'évaluation d'application dans le cadre de la reconfiguration dynamique.

Ma contribution à la mise en œuvre de l'environnement de développement d'application répartie que nous voulons fournir réside dans le développement de la classe des agents reconfigurables et dans la conception du support pour la reconfiguration dynamique.

Le point sur le travail restant à effectuer est le suivant : l'outil graphique est en cours de développement ainsi que certaines applications (dont notamment une application de courrier électronique). Une fois ces deux derniers points effectués, nous aurons complété notre environnement de construction d'application répartie et nous aurons une application réelle utilisant nos mécanismes de reconfiguration et développée à l'aide de notre environnement.

Chapitre V.

Conclusion

1 Problématique

La problématique à laquelle nous avons cherché une solution est celle de *la reconfiguration dynamique d'applications réparties*.

La reconfiguration dynamique d'application répartie peut se définir comme l'ensemble des changements que l'on peut apporter à une application répartie sans arrêter son exécution. Ces changements peuvent être de différentes types :

- **Les changements de structure**

Les changements de structure modifient la configuration structurelle d'une application répartie, c'est à dire la description de l'ensemble des composants et de leurs interconnexions. Les changements structuraux de base sont : l'ajout , le retrait de composant et la modification des liaisons entre composants

- **Les changements de géométrie**

Les changements géométriques altèrent uniquement le placement des composants en terme de site d'exécution. Intuitivement ce type de changement ressemble au problème de la migration de processus si l'on assimile un composant à un processus.

- **Les changements de mise en œuvre**

Les changements d'implémentation ne changent ni la structure de l'application ni le placement, ni l'interconnexion des composants mais modifient le code des composants et éventuellement restructurent leurs données sans modifier leur interface.

- **Les changements d'interface**

La reconfiguration dynamique d'application répartie permet de réduire les coûts liés à l'arrêt de l'application en cours d'exécution pour effectuer les opérations de maintenance, réinstaller une nouvelle version de l'application ou faciliter l'évolution d'une application.

L'objectif de mon travail est de fournir une approche générale permettant de résoudre les différents type de changements associés à la *reconfiguration dynamique d'application répartie*. Ainsi que des outils systèmes permettant de supporter ces changements.

2 Contexte

La classe d'application répartie à laquelle nous nous intéressons peut être décrite comme un assemblage d'agents interconnectés par le bus logiciel **A3** qui met œuvre un modèle de communication par événement. La conception d'application répartie peut tirer partie des propriétés assurées par l'environnement **A3** qui sont :

- Exécution transactionnelle des réactions associées à une notification.
- Diffusion causale des notifications.

La description d'une application est réalisée à l'aide du langage de description d'architecture **OCL** [[4]] permettant de séparer l'assemblage des agents de leur programmation. Un tel langage permet de construire une application avec une vision macroscopique des composants logiciel requis et de leur manière d'interagir et de communiquer dans un environnement réparti. Le contexte de ce travail repose sur les outils de description d'architecture d'application **OLAN** et du modèle d'exécution **A3**.

3 Contribution

3.1 Description d'un algorithme de reconfiguration

Le principal résultat du travail effectué dans le cadre du DEA est d'avoir décrit et prouvé un algorithme pour la reconfiguration dynamique permettant d'effectuer les changements de structure et de géométrie. Cet algorithme est original car :

- Il est le fruit d'une synthèse de trois approches étudiées dans la section état de l'art. En effet, notre algorithme utilise :
 - Une notion similaire aux états abstraits de **CONIC** pour déterminer le moment où une reconfiguration peut s'effectuer de manière cohérente.
 - Un modèle de communication asynchrone basé sur un bus logiciel similaire à **POLYLITH** permettant de faire abstraction du problème des requêtes dépendantes.
 - Des mécanismes transactionnels permettant de fournir un algorithme de reconfiguration robuste de manière similaire au système **ARGUS**.
- Il s'appuie sur *un mécanisme de diffusion causale* des notifications.

3.2 Mise en œuvre de l'algorithme de reconfiguration

Un résultat secondaire est celui de la mise en œuvre de l'algorithme de reconfiguration sur la plateforme **A3**. Cette mise en œuvre permet de montrer la viabilité de notre algorithme dans le cadre d'un environnement réel.

Un certain nombre d'applications tests sont en cours de développement dans le but d'évaluer de manière plus précise les performances de l'algorithme.

3.3 Conclusions générales

L'algorithme de reconfiguration semble permettre de tirer un certain nombre de résultats généraux concernant la reconfiguration dynamique d'applications réparties :

- Le modèle de communication asynchrone permet d'éviter le problème des sessions dépendantes rencontrées avec l'utilisation des modèles de communications synchrones. Il en résulte un gain de performances significatif en terme de composants devant être stoppés durant les phases de reconfiguration.
- L'utilisation d'un mécanisme de diffusion causale des notification permet :
 - De faciliter le vidage des canaux de communication.
 - D'assurer un ordonnancement correct des notifications en attente lors de la réactivation d'un agent.
 - De minimiser les perturbations de l'application, en terme d'agent dont l'exécution est gelée.
- L'utilisation d'un langage de description d'architecture (**ADL**) permet de faciliter la modification des interconnexions. En effet, la définition des interconnexions n'est pas incluse dans la mise en œuvre d'un composant ce qui permet d'avoir un contrôle externe des communications entre agents. De plus, l'ADL fournit un canevas pour la validation des changements, car le fait d'avoir la connaissance de l'application permet de vérifier la faisabilité et la validité d'une action de reconfiguration.

4 Perspectives

Les perspectives à court terme de ce travail de DEA réside dans l'extension de l'algorithme de reconfiguration existant et dans son expérimentation et plus précisément des points suivants :

- L'évaluation plus précise de l'algorithme de reconfiguration. Cela comprend :
 - L'expérimentation de l'algorithme sur des applications réelles dont NetWall[25] le logiciel de pare-feu.
 - L'évaluation de l'algorithme à l'aide du modèle LTS[19], basée sur la modélisation du comportement à l'aide de graphes de transition.
- L'extension de l'algorithme pour donner une **priorité** aux notifications de reconfiguration. En effet, actuellement, les notifications de reconfigurations ont la même priorité que les autres notifications. Par conséquent, l'exécution d'une notification de reconfiguration peut être retardée pour permettre l'exécution de notification antérieure.
- La comparaison de notre algorithme à d'autres algorithmes existants, et notamment à un algorithme de reconfiguration mis en œuvre par notre équipe sur le support **CORBA**.

Au vue de l'algorithme de reconfiguration un certain nombre de perspectives à long terme peuvent être émises. Ces perspectives résident dans les points suivants :

- L'adaptation de l'algorithme de reconfiguration pour certains types d'application ayant des contraintes de performance importante, comme par exemple les applications temps réelles.
- La conception d'un algorithme permettant la validation des changements.
- L'étude des processus initiateurs d'actes de reconfiguration[2]. Dans notre approche, c'est un opérateur humain qui les initie, avec le contrôle de l'ADL pour vérifier si ce changement est syntaxiquement correct. D'autres utilisations possibles sont envisageables, comme des règles d'administration associées à l'application qui permettent de piloter dynamiquement l'utilisation des mécanismes de reconfiguration. Ce travail offre les bases d'un grand nombre d'applications.

Bibliographie

- [1] R. Balter and al., « Architecture and Implementation of Guide, an Object Oriented Distributed System », *Computing Systems*, Vol.4(No.1), 1991, pp. 31-67.
- [2] M. R. Barbacci, D. L. Doubleday, M. J. Ardner, R. W. Lichota, « Durra : a Structure Description Language for Developing Distributed Application », *Software Engineering Journal*, Vol. 8(No.2), Mars 1993, IEE and BCS, London, UK, pp. 83-94.
- [3] M. R. Barbacci, D. L. Doubleday, C. B. Weinstock, « Application Level Programming », *Proceedings of the 10th Int'l Conference on Distributed Computing Systems*, 1990, pp. 458-465.
- [4] L. Bellissard, S. Ben Atallah, F. Boyer, M. Riveill, « Distributed Application Configuration », *Proc of the 16th IEEE Int'l Conference on Distributed Computing Systems (ICDCS'96)*, IEEE Computer Society Press, Hong Kong, Avril 1996, pp. 579-585.
- [5] L. Bellissard, F. Boyer, M. Riveill, J. Y. Vion-Dury, « System Services for Distributed Application Configuration », *Proc of the 4th Int'l Conference on Configurable Distributed Systems (ICDCS'98)*, IEEE Computer Society Press, Annapolis MD, USA, May 1998, pp. 53-60.
- [6] C. Bidan, V. Issarny, T. Saridakis, A. Zarras, « A Dynamic Reconfiguration Service for CORBA », *Proc of the 4th Int'l Conference on Configurable distributed systems (ICDCS'98)*, IEEE Computer Society Press, Annapolis MD, USA, Mai 1998, pp. 35-42.
- [7] A. D. Birrell et B. L. Nelson, « Implementing Remote Procedure Calls », *ACM Transactions on Computer Systems*, Vol.2(No.1), Février 1984, pp. 39-59.
- [8] T. Bloom, M. Day, " Reconfiguration and module replacement in Argus : Theory and Practice", *Software Engineering Journal*, Vol.8(No.2), IEE and BCS, London UK, Mars 1993, pp. 102-108.

- [9] T. Chandra, S. Toueg, « Unreliable Failure Detector for Reliable Distributed Systems », *Journal of the ACM*, mars 1996, pp.225-267.
- [10] K. M Chandy, L. Lamport, « Distributed Snapshots : Determining Global States of Distributed Systems », *ACM Transaction on Computer Systems*, Vol.3(No.1), Février 1985, pp. 63-75.
- [11] R. P. Cook, I. Lee, « DYMOs : A Dynamic Modification System », *SIGPLAN Notices*, ACM Press, Vol.18(No.8), 1983, pp. 201-202.
- [12] F. De Remer, H. H. Kron, « Programming-in-the-large vs. Programming-in-the-small », *IEEE Transactions on Software Engineering*, Vol.SE-2(No.2), IEEE Computer Society Press, Juin 1976, pp. 114-121.
- [13] K. Goudarzi, J. Kramer, « Maintaining node consistency in the face of dynamic changes », *Proc. of the 3rd Intn'l Conference on Configurable Distributed Systems (ICDCS'96)*, IEEE Computer Society Press, Annapolis MD, USA, Mai 1996, pp. 62-69
- [14] M. Herlihy, B. Liskov, « A value Transmission Methode for Abstract Data Types », *ACM Tans on Programming Languages and Systems*, Vol. 4(No. 4), Octobre 1982.
- [15] C. Hofmeister, J. Purtilo, « Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for Replacement », *Proc. of the 13th International Conference in Distributed Computing Systems (ICDCS'93)*, IEEE Computer Society Press, 1993, pp. 101-110.
- [16] C. R. Hofmeister, J. M. Purtilo, « Surgeon : a Packager for Dynamically Reconfigurable Distributed Applications », *Software Engineering Journal*, Vol.(No.), IEE and BCS, London UK, Mars 1993, pp. 95-101.
- [17] J. Kramer, J. Magee, « Dynamic Configuration for Distributed Systems ». *IEEE Transactions on Software Engineering*, Vol.SE-11(No.4), Avril 1985, pp. 424-436,.
- [18] J. Kramer, J. Magee, « The Evolving Philosophers Problem : Dynamic Change Management », *IEEE Transactions on Software Engineering*, Vol.16(No.11), IEEE Computer Society Press, Novembre 1990, pp. 1293-1306.
- [19] J. Kramer, J. Magee, « Analysing Dynamic Change in Software Architectures : A case of study », *Proc of the 4th Intn'l Conference on Configurable Distributed Systems (ICDCS'98)*, IEEE, Annapolis MD, USA, Mai 1998, pp. 91-100
- [20] B. Liskov, « Distributed Programming in ARGUS ». *Communication of the ACM*, Vol.31(No.3), Mars 1988, pp. 300-312.
- [21] I. Oueichek et X. Rousset de Pina, « Dynamic Configuration Management in the Guide Object-Oriented Distributed Systems », *Proc. of the 3rd Int'l Conference on Configurable Distributed Systems (ICDCS'96)*, IEEE Computer Society Press, Mai 1996, pp. 28-35.

Conclusion

- [22] J. M. Purtilo, « The POLYLITH Software Bus », *ACM TOPLAS*, Vol.16(N.1), Janvier 1994, pp. 151-174.
- [23] M. E. Segal, O. Frieder, « On-the-fly Program Modification : Systems for Dynamic Updating », *IEEE Software*, Vol.(No.), IEEE Computer Society Press, Mars 1993, pp. 53-65.
- [24] J. M. Smith, « A Survey of Process Migration Mechanisms », *SIGOPS*, Vol.22(No.3), Juillet 1988, pp. 28-40.
- [25] F. Soinne, « Netwall Version 3.2 », Bull Whitepaper, available at <http://www-frec.bull.com/ospbuhp.htm>.
- [26] M. Theimer, B. Hayes, « Heterogeneous Process Migration by Recompilation », *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991, pp 18-25.