Configuration et déploiement d'intergiciel asynchrone sur système hétérogène à grande échelle

Philippe Laumay

▶ **To cite this version:**

Philippe Laumay. Configuration et déploiement d'intergiciel asynchrone sur système hétérogène à grande échelle. Réseaux et télécommunications [cs.NI]. Institut National Polytechnique de Grenoble - INPG, 2004. Français. NNT : . tel-00005409

HAL Id: tel-00005409

https://theses.hal.science/tel-00005409

Submitted on 22 Mar 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : « Informatique : Systèmes et Communication »

préparée au laboratoire LSR-IMAG, projet SARDES,
dans le cadre de l'Ecole Doctorale
« Mathématiques Sciences et Technologies de l'Information »

présentée et soutenue publiquement par

Philippe LAUMAY

le 5 mars 2004

Configuration et déploiement d'intergiciel asynchrone sur système hétérogène à grande échelle

---

Directeur de thèse :

Jacques MOSSIÈRE

JURY

M.	Sacha	KRAKOWIAK	Président
M.	Guy	BERNARD	Rapporteur
M.	Michel	RIVEILL	Rapporteur
M.	Jacques	MOSSIÈRE	Directeur de thèse
M	Luc	BELLISSARD	Co-encadrant
M.	Jérome	D'ANNOVILLE	Examinateur

Résumé

L'émergence des environnements omniprésents pose un nouveau défi aux systèmes informatiques. Les intergiciels asynchrones (*Message-Oriented Middleware, MOM*) sont reconnus comme étant la solution la plus apte à répondre aux besoins de passage à grande échelle, de flexibilité, et d'hétérogénéité des nouvelles applications distribuées. Mais l'implémentation des intergiciels asynchrones actuels reste souvent figée quels que soient les sites d'exécution et l'application (les applications) l'utilisant et sont peu voire pas configurables.

Cette thèse s'intéresse à la configuration et au déploiement des intergiciels asynchrones sur système hétérogène à grande échelle. Elle vise la définition d'un modèle d'intergiciel asynchrone configurable permettant une configuration statique et à l'exécution. Elle a pour objectif d'associer les nombreux travaux dans le domaine de l'asynchrone qui ont mené à la définition des modèles de communication asynchrones et les réflexions menées dans les intergiciels synchrones autour des nouveaux besoins de configuration et d'adaptabilité.

La synthèse de tous nos travaux nous a mené à la création de DREAM (*Dynamic REflective Asynchronous Middleware*), un intergiciel asynchrone adaptable. Les mécanismes de contrôles fournis par Dream ainsi que son architecture permettent de réaliser une configuration en se basant sur les besoins applicatifs et les contraintes imposées par le système.

Nous validons notre prototype par l'implémentation d'un service à événement à base d'agents dans lequel nous utilisons les capacités d'adaptation de DREAM pour ajouter de nouvelles fonctionnalités.

Abstract

A new generation of intelligent and communicating ubiquitous devices is emerging today. It's a challenge for computing systems because it's necessary to meet each entity specific needs. Today, the use of asynchronous communication infrastructures (MOM for Message-Oriented Middleware) is recognized as the only means to achieve these scalability, extensibility and adaptability objectives. But actual MOM implementations are often fixed whatever hardware site or application may be used. Moreover they often provide only a communication API and are not (very) configurable.

This Ph.D. thesis focuses on configuration and deployment of message-oriented middleware on scalable heterogeneous systems. It aims at the definition of a configurable MOM model allowing static and dynamic configuration. The objective is to associate results on asynchronous systems which lead to the definition of asynchronous communication models, and recent works on synchronous middlewares which focus on adaptability.

Our works lead to the creation of DREAM (Dynamic REflective Asynchronous Middleware), an adaptable message-oriented middleware. The control mechanisms and the reflective architecture provided by DREAM allow to base the middleware configuration on applicative needs and hardware constraints.

Our prototype is validated by an implementation of an events service based on agents.

Remerciements

Je tiens à remercier par ces quelques lignes toutes les personnes grâce à qui cette thèse a abouti. Je remercie donc :

Sacha Krakowiak de m'avoir fait l'honneur de présider le jury de thèse.

Michel Riveill d'avoir bien voulu juger ce travail et d'avoir accepté d'encadrer ma soutenance.

Guy Bernard d'avoir accepter de juger ce travail et d'encadrer ma soutenance.

Jérome D'Annoville et l'entreprise SCHLUMBERGER smart cards and terminals (désormais AXALTO) qui m'ont permis d'effectuer mon travail dans les meilleures conditions matérielles possibles.

Jacques Mossière pour ses conseils avisés sur le document et la soutenance de cette thèse

Luc Bellissard pour son rôle de co-directeur de thèse et l'aide qu'il m'a apporté tout au long de cette thèse.

Je tiens aussi à remercier :

Roland Balter, directeur du projet SIRAC, ainsi que Jean-Bernard Stefani, directeur du projet SARDES, de m'avoir permis d'évoluer dans leurs équipes de recherche.

Les « Sardines » : Aline, Vania, Oussama, Nono, Daniel, Baz et Julien pour les bons moments passés en leur compagnie. Slim pour ses conseils durant les derniers mois de rédaction, ainsi que Nicolas Tachker sans qui je n'aurais pas continué.

Je termine, en adressant des remerciements tous spéciaux à :

- Olivier, Raphaël et Rémi pour les nombreuses discussions sur nos doutes, nos espoirs, et nos rêves lors de nos soirées et de nos « pauses ».
- Sandrine pour sa relecture du document et les corrections de mes nombreuses fautes.
- Mes parents, ma soeur et mon frère pour leur affection et leur soutien tout au long de mes nombreuses années d'étude !

Et par dessus tout, je remercie Sonia pour avoir toujours su trouver les mots qu'il faut dans les meilleurs comme dans les pires moments. Pour m'avoir soutenu, encouragé et supporté durant ces trois difficiles années de thèse faites de doutes et de remises en question permanente.

Table des matières

Table des figures

Introduction

Motivations et Objectifs

Les dernières avancées dans les systèmes distribués, mobiles et omniprésents demandent de nouveaux environnements caractérisés par un haut degré de dynamisme. Les grandes variations de ressources disponibles, la connectivité des réseaux et l'hétérogénéité des plate-formes logicielles et matérielles influencent de plus en plus les performances des applications. La croissance attendue de l'informatique omniprésente (*Ubiquitous Computing*) sur les cinq prochaines années va grandement changer la nature des infrastructures systèmes, impliquant une pléthore de petits appareils mobiles tel que les PDA[1] ou les cartes à puce nécessitant des configurations spécifiques en termes de protocoles, qualité de service, etc.

Durant les dix dernières années, de nombreux travaux dans les systèmes distribués ont mis au points la technologie des intergiciels[2] afin de faciliter le développement d'applications réparties à grande échelle. Un intergiciel est une couche logicielle qui se situe entre le système d'exploitation et l'application. Il fournit des solutions réutilisables à des problèmes fréquemment rencontrés lors de la construction de différents classes d'applications. La plupart des infrastructures d'intergiciel disponibles à ce jour sont fondées sur le paradigme *client-serveur*, un modèle de communication synchrone. Mais, si les intergiciels synchrones (et leur extension en ORB, *Object Request Broker*) sont les plus répandus dans le monde des intergiciels, il existe néanmoins un autre paradigme qui s'impose de plus en plus comme étant la solution la plus apte à répondre aux besoins de passage à grande échelle, de flexibilité, et d'hétérogénéité des applications distribuées à grande échelle, il s'agit des intergiciels asynchrones (ou intergiciels orientés messages : *Message-Oriented Middleware, MOM*).

La notion d'interaction asynchrone est un des plus vieux mécanismes de coordination depuis le développement des systèmes bas niveau reliés directement aux mécanismes d'interruptions matérielles. Mais l'absence de structures et de primitives de haut niveau rendait ces systèmes fragiles et peu fiables. Dans les années 70, Les premiers modèles de communication asynchrone à base d'échange de messages ont fait leur apparition. Les années 80 et 90 et le développement grandissant des réseaux ont permis de voir apparaître les travaux autour des notions de queue de messages, d'événements, de système par d'abonnement, etc. qui ont posé les bases des intergiciels asynchrones actuels.

L'utilisation des intergiciels asynchrones dans les environnements omniprésents devient un nouveau défi. Il devient nécessaire de répondre aux problèmes spécifiques de chaque entité participant au système. Mais l'implémentation des intergiciels asynchrones actuels reste souvent figée quels que soient les sites d'exécution et l'application (ou les applications) l'utilisant. De plus ils n'offrent

[1] *Personnal Digital Assistant*, Assistant Personnel Numérique.

[2] Le mot anglais *Middleware* a trouvé sa traduction française en *Intergiciel*. D'autres traductions moins répandues existent, comme par exemple : *Intersticiel*.

souvent qu'une API de communication et sont peu voire pas configurables. Et malgré l'aide au développement que fournissent ces intergiciels, ils manquent de support pour les aspects dynamiques des nouvelles infrastructures informatiques. Les prochaines générations d'applications ont besoin d'intergiciels pouvant s'adapter au changement d'environnement et se configurer en fonction des besoins spécifiques des appareils, du PDA au puissant calculateur.

Notre objectif est de proposer une architecture d'intergiciel asynchrone permettant de répondre à ces nouveaux besoins.

Cadre du travail

Cette thèse a été effectuée au sein du projet SARDES[3] (IMAG-LSR et INRIA Rhône-Alpes) dans le cadre d'une collaboration entre l'INRIA et l'entreprise SCHLUMBERGER SMART CARDS AND TERMINALS .

Les travaux de recherche dans SARDES portent sur la construction d'infrastructures réparties adaptables. En particulier, ils s'intéressent aux environnements à grande échelle dont les ressources de calcul changent dynamiquement et étudient les mécanismes d'adaptation préservant la qualité de service des applications s'exécutant dans de tels environnements. Ils portent également sur l'activité d'administration qui inclut la surveillance des applications à l'exécution et leur reconfiguration lors d'une dégradation de performances.

Dans le cadre de la collaboration avec Schlumberger, nous avons travaillé sur la mise au point d'un prototype d'intergiciel asynchrone sur carte à puce. C'est notamment suite à ces travaux que nous nous sommes orienté vers la définition de notre modèle d'intergiciel asynchrone.

Démarche suivie

Durant ce doctorat, nous n'avons pas réalisé nos travaux de façon linéaire. Nous avons progressé en abordant des points relativement différents mais qui nous ont menés vers une même constatation : Les intergiciels asynchrones actuels reste très monolithiques, ils embarquent un ensemble de fonctionnalités non « débrayables » et ne permettent pas une spécialisation de leur architecture et de leur comportement.

Nous avons, dans un premier temps, abordé le problème du passage à grande échelle d'un intergiciel asynchrone. Nos travaux ont porté sur l'impact de la propriété d'ordonnancement causal lors du passage à l'échelle et ont mis en avant les faiblesses en termes de configuration des intergiciels asynchrones. Ensuite, dans le cadre de la collaboration industrielle, nous avons travaillé sur la mise au point d'un prototype d'intergiciel asynchrone sur un environnement à grande échelle contraint, les cartes à puce. Cette expérience a confirmé les manques de configuration et de configurabilité limitant le portage de l'intergiciel sur carte à puce.

En parallèle, nous avons étudié les différents travaux sur l'évolution des intergiciels synchrones. Ces travaux de configuration des intergiciels synchrones ont mis en avant trois points essentiels que nous avons tenter d'exploiter :

- Offrir la possibilité d'être exécuté de façon minimale (c'est-à-dire un intergiciel offrant un comportement fonctionnel de base sans propriétés inclue "en dur").
- Pouvoir accueillir (statiquement ou dynamiquement lors de l'exécution) de nouveaux comportements fonctionnels ou responsables de la gestion de propriétés non fonctionnelles.

[3] SARDES = System Architecture for Reflexive Distributed EnvironmentS.

– La configuration doit se faire dans le respect des besoins applicatifs et des contraintes du système.

Les modèles d'intergiciels réflexifs étudiés sont une voie efficace pour supporter le développement d'intergiciels flexibles et adaptables répondant aux nouveaux environnements dynamiques.

La synthèse de tous ces travaux nous a mené à la création de DREAM (*Dynamic REflective Asynchronous Middleware*) un intergiciel asynchrone adaptable. Cet intergiciel associe les nombreux travaux dans le domaine de l'asynchrone qui ont mené à la définition des modèles de communication asynchrones et les réflexions menées dans les intergiciels synchrones autour des nouveaux besoins de configuration et d'adaptabilité.

Plan du document

Ce document est organisé en 6 chapitres.

Le chapitre 1 présente une synthèse sur le fonctionnement des intergiciels asynchrones puis une étude de cas de certains intergiciels asynchrones. Ce chapitre a pour but de dresser un état de l'art sur les intergiciels asynchrones sans forcément rentrer dans les détails d'implémentation de chacun. Nous souhaitons montrer que les projets actuels se concentrent essentiellement sur les modèles de communication mais ne se préoccupent pas ou peu des problèmes d'adaptabilité des intergiciels asynchrones, question à laquelle nous tentons de répondre dans les chapitres suivants. Cette étude nous permet de mettre en avant l'aspect monolithique des intergiciels asynchrones.

Le chapitre 2 étudie les différentes méthodes de configuration dans les intergiciels synchrones. Nous nous servons ensuite de cette étude pour trouver les techniques les plus aptes à la mise en place de notre architecture d'intergiciel asynchrone. Puis nous dressons une bref synthèse sur le déploiement des intergiciels.

La configuration d'intergiciels à composants requiert une description précise et non ambiguë des propriétés de la structure et de comportement. Cette description permet de maintenir une représentation explicite des dépendances entre composants et besoins des composants. Dans le chapitre 3, nous étudions des langages permettant de décrire formellement la structure et les propriétés qui caractérisent le comportement d'un logiciel complexe.

Le chapitre 4 expose nos travaux autour de deux aspects distincts qui nous ont permis de soulever les nouveaux besoins d'adaptabilité des intergiciels asynchrones. Tout d'abord nos travaux sur l'impact de la propriété d'ordonnancement causal lors du passage à l'échelle d'un intergiciel asynchrone mettant en avant les faiblesses en terme de configuration des intergiciels asynchrones. Ensuite, le déploiement d'un intergiciel sur un environnement à grande échelle contraint (les carte à puce) a confirmé ces manques de configuration et de configurabilité limitant le portage de l'intergiciel sur carte à puce.

Le chapitre 5 est consacré à la description de notre modèle d'intergiciel asynchrone adaptable : DREAM (***D**ynamic **RE**flective **A**synchronous **M**iddleware*). Nous décrivons l'architecture détaillée de l'intergiciel basée sur le canevas logiciel Fractal ainsi que le langage de description de l'intergiciel (*Middleware Description Language*, MDL). Ce MDL est utilisé par un processus de configuration et de déploiement à grande échelle qui propose une solution pour faciliter la configuration et le déploiement local et global de l'intergiciel.

Avant de conclure ce document, nous exposons rapidement dans le chapitre6, une expérimentation

de DREAM pour la création d'une personnalité de service à événements. Puis nous réalisons une discussion sur notre processus de déploiement à grande échelle.

Chapitre 1

Etude des intergiciels asynchrones

1.1 Introduction

Un intergiciel est une couche logicielle qui se situe entre le système d'exploitation et l'application [18]. Il fournit des solutions réutilisables à des problèmes fréquemment rencontrés lors de la construction de différentes classes d'applications ; citons par exemple les problèmes relatifs à la gestion des communications, à la fiabilité ou encore à la sécurité des applications distribuées. L'intergiciel masque les détails d'implémentation ayant trait à l'hétérogénéité des plates-formes d'exécution au moyen d'un ensemble d'interfaces.

Intergiciels synchrones La plupart des infrastructures d'intergiciel disponibles à ce jour sont fondées sur le paradigme *client-serveur*, un modèle de communication synchrone (l'appel de procédure à distance ou RPC *Remote Procedure Call*). Le mode synchrone est le plus simple des modes de synchronisation, la définition donnée dans [4] est la suivante : « *l'émetteur et le récepteur de la communication doivent être prêts à communiquer avant qu'un envoi ne puisse être effectué* ». L'exemple le plus souvent utilisé pour définir le mode synchrone est la communication téléphonique. L'émetteur ET le récepteur doivent établir une connexion afin d'être tous les deux prêts à communiquer, puis ils parlent chacun à leur tour, attendant que l'autre ait fini pour commencer. Un autre exemple est l'appel de procédure (local ou distant). Lors d'un appel de procédure, l'appelant se met en attente ; une fois la procédure appelante terminée, l'exécution se poursuit chez l'appelant. Le mode synchrone exige que les applications qui communiquent entre elles soient disponibles au moment de l'échange sous peine de perdre purement et simplement la communication. Notons que le type d'intergiciel synchrone le plus répandu est le « courtier à objets » ou ORB (*Object Request Broker*) qui est un intergiciel client-serveur à objets.

Intergiciels asynchrones Même si les intergiciels synchrones (et leur extension en ORB) sont les plus répandus dans le monde des intergiciels, il existe néanmoins un autre paradigme qui s'impose de plus en plus comme étant la solution la plus apte à répondre aux besoins de passage à grande échelle, de flexibilité et d'hétérogénéité des applications distribuées à grande échelle, il s'agit des intergiciels asynchrones (ou intergiciels orientés messages : *Message-Oriented Middleware, MOM*).

Les intergiciels asynchrones peuvent être vus comme une évolution du paradigme des communications par paquets qui prévaut dans les couches basses du modèle des réseaux à couches OSI [72]. Les intergiciels asynchrones offrent aux applications la possibilité de communiquer par échange de messages. Ainsi les applications ne sont plus liées les unes aux autres, et l'émettrice d'un message peut poursuivre son exécution dès que le message est envoyé. Cela présente un intérêt certain pour

de nombreuses applications, par exemple celles qui traitent les données au fur et à mesure qu'elles deviennent disponibles ou encore dans un contexte où des plates-formes sont mobiles ou peu disponibles (dans le cas d'une application à l'échelle mondiale), auquel cas une application n'est pas bloquée par la non disponibilité (déconnecté, occupé, ...) du récepteur.

Il est à noter qu'il est possible de bâtir des applications ayant besoin de réaliser des échanges synchrones (comme par exemple une communication téléphonique) sur un intergiciel asynchrone, le processus émetteur attendant alors la réception d'un acquittement de fin de traitement de la part du récepteur avant de continuer son exécution.

1.2 Description et propriétés des intergiciels asynchrones

Cette section présente une description et une classification des différents modèles et architectures d'intergiciels asynchrones. Cette classification nous semble importante pour comprendre puis discuter des nouveaux défis d'adaptation auxquels doivent faire face les intergiciels modernes.

Dans un article paru récemment sur l'intégration des intergiciels asynchrones dans les transactions distribuées ([87]), les auteurs proposent une classification des architectures d'intergiciels asynchrones en trois modèles. Nous nous sommes inspirés de cette séparation pour présenter une synthèse sur les intergiciels asynchrones. Cette section est donc scindée en trois parties :

- La première est consacrée aux modèles de communication des intergiciels asynchrones, c'est-à-dire aux différents types de délivrance de messages d'un intergiciel asynchrone (section1.2.1).
- La deuxième expose les différentes architectures que peuvent prendre les intergiciels asynchrones pour délivrer un message d'un point à un autre (section1.2.2).
- La troisième introduit les différentes propriétés non-fonctionnelles supportées par les intergiciels asynchrones permettant d'assurer la qualité de service de la délivrance (section1.2.3).

1.2.1 Modèles de communication

Outre les règles de synchronisation, un intergiciel asynchrone suit un modèle de communication qui dicte la manière d'échanger les messages. Les modèles usuels sont la communication par envoi de messages et la communication par file de messages, qui se distinguent par leur degré de fiabilité. Au-dessus de ces modèles, le principe d'abonnement apporte la notion d'anonymat. Enfin, le modèle événementiel, plus proche d'un modèle de programmation, permet d'associer des traitements à des événements, telle que la réception d'un message.

1.2.1.1 Envoi de messages et file de messages

La communication par envoi de messages (*message passing*) représente le modèle de base de la communication dans les intergiciels asynchrones. Elle est généralement non-bloquante, et prend la forme d'un échange de messages entre processus, de la même manière que l'émission d'une lettre (voir figure 1.1).

Dans ce modèle, si le destinataire d'un message ne peut le traiter immédiatement à sa réception, celui-ci est stocké dans une queue de messages qui fait office de tampon (représenté par la boîte aux lettres sur la figure 1.1). Le destinataire doit néanmoins être disponible à la réception d'un message, afin que le gestionnaire de messages ne conserve pas les messages si le destinataire n'est pas accessible (panne machine ou réseau par exemple).

Le *message queuing* ajoute la notion de fiabilité au modèle précédent par l'utilisation d'une file de messages (ou queue de messages). Le rôle de la file d'attente, ici, est d'assurer une fonction de

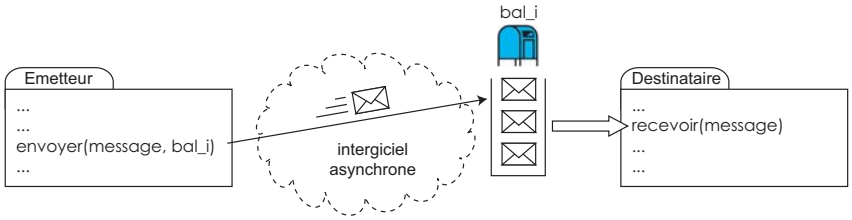

Figure 1.1 – Communication par envoi de messages

sécurisation des messages permettant de conserver leur intégrité quelle que soit la situation : arrêt de l'application destinataire, arrêt du système ou même destruction du support physique contenant la file d'attente. De plus, la représentation symbolique de la queue de messages fiable n'est plus chez le destinataire mais au sein du gestionnaire de messages (voir figure 1.2). Cela s'appelle la persistance des messages.

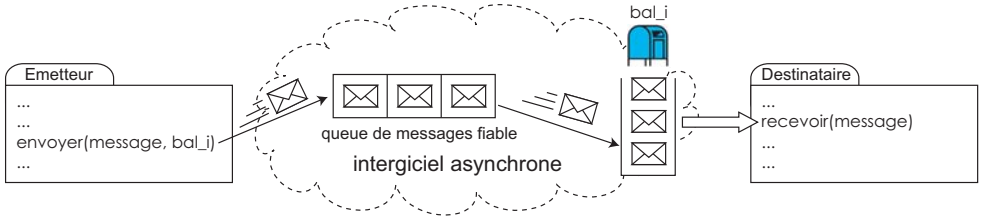

Figure 1.2 – Communication avec file de messages

Dans ce modèle l'application n'accède pas directement aux files d'attente mais fait appel à un gestionnaire de files d'attente par l'intermédiaire d'une interface de programmation, contenant les méthodes `envoyer` et `recevoir`, capables d'accéder aux files d'attente et d'envoyer ou recevoir un message.

La communication avec file de messages est un modèle de communication très utilisé dans les intergiciels asynchrones actuels, tels que IBM Gryphon (voir section 1.3.1.2), IBM MQSeries (voir section 1.3.2.1), VCom, Microsoft MQ (voir section 1.3.2.2) ; c'est d'ailleurs sur ce modèle qu'ont été implémentés les premiers intergiciels asynchrones comme MQSeries d'IBM.

Néanmoins, ces modèles ont leurs limites, notamment si l'émetteur veut envoyer la même information à plusieurs destinataires, auquel cas il doit faire plusieurs envois (un pour chaque destinataire). Une solution est la communication de groupe. Un groupe est un ensemble de récepteurs identifiés par un nom unique qui sont gérés de façon dynamique : les arrivées/départs de membres au sein du groupe peuvent se réaliser pendant le déroulement de l'application. Les intergiciels utilisant ce mode de communication sont Isis, Horus, Ensemble (Université de Cornell) [89, 21]. Mais ce modèle ne répond pas encore à toutes les exigences pouvant être demandées à un intergiciel asynchrone, comme la communication anonyme, c'est pourquoi beaucoup d'implantations actuelles utilisent un modèle de communication par abonnement.

1.2.1.2 Modèle par abonnement

Le modèle par abonnement (*Publish/Subscribe*) est le plus récent des modèles de communication utilisés dans les intergiciels asynchrones (voir [12], [29] et les spécifications de l'API JMS [44]). Il se caractérise par la notion d'anonymat : le message est envoyé à un gestionnaire qui se charge de le renvoyer à ceux qui se sont abonnés à ce type de messages.

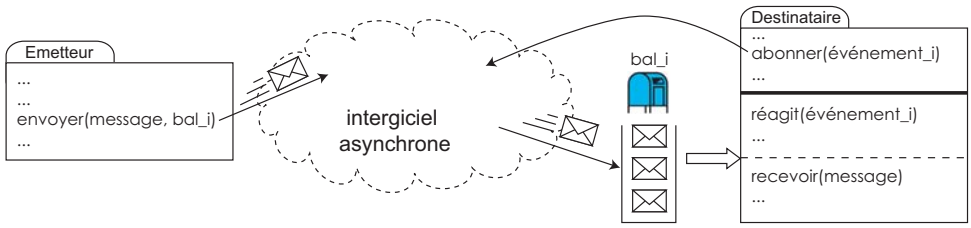

Figure 1.3 – Le modèle par abonnement

Dans les systèmes de MOM par abonnement il y a deux types de participants. Tout d'abord des fournisseurs (*providers*) qui envoient (publient) des événements dans le système, et ensuite des consommateurs (*consumers*) qui s'abonnent (souscrivent) à certaines catégories d'événements du système sur certains critères. Comme on peut le voir sur la figure 1.3, le consommateur d'événements a deux manières de consommer l'événement ainsi reçu. Soit il va le chercher dans sa file locale de message (comme dans l'envoi de messages (*message passing*) ou les files de messages (*message queuing*)), soit il « réagit » à l'événement (voir la section 1.2.1.3 sur le modèle de communication événementiel). Il existe deux principaux critères d'abonnement :

– l'abonnement basé sur le sujet de l'événement (*subject-based*), le plus ancien et le plus répandu ;
– la technique émergente basée sur le contenu de l'événement (*content-based*).

Abonnement basé sur le sujet Les plus anciens et les plus répandus des systèmes par abonnement sont basés sur le sujet (*subject-based*) ([70, 44]) propose un service d'abonnement basé sur des critères d'abonnement sur le sujet (*Topic*). Dans ces systèmes, chaque événement est classé comme appartenant à l'un des types de sujets prédéfinis (aussi appelés groupes, canaux ou intérêts). Les fournisseurs sont donc tenus de donner à leurs événements un des sujets prédéfinis, et les consommateurs doivent s'abonner à l'un de ces sujets fixés. Dans les extensions de ce modèle il est possible de récupérer tous les sujets disponibles à partir d'un site spécifique (un serveur de nom par exemple) afin de sélectionner celui auquel on désire s'abonner. L'abonnement se réalise auprès du système d'abonnement de l'intergiciel (centralisé, réparti, ...). Par la suite, tout événement émis correspondant au sujet demandé sera reçu par l'abonné.

Abonnement basé sur le contenu Une alternative émergente à la technique basée sur le sujet est la technique basée sur le contenu (*content-based*). Ce type d'abonnement apporte une flexibilité supplémentaire pour les abonnés, le choix de filtrage parmi les différentes valeurs des attributs des événements permet de s'abstenir de la pré-définition des sujets [12]. Le principal avantage de l'abonnement sur contenu, par rapport au traditionnel abonnement basé sur le sujet, est qu'il permet aux producteurs d'envoyer de l'information sur le réseau sans s'occuper ni de la connaissance des destinataires, ni de la localisation de ces destinataires. Dans notre exemple, l'abonné basé sur le sujet est forcé de choisir une demande parmi les sujets proposés contrairement à l'abonné basé sur le contenu qui est libre de choisir un critère orthogonal selon la valeur des attributs. Enfin, le critère basé sur le contenu est plus général car il peut être utilisé pour implémenter l'abonnement par sujet alors que la réciproque est fausse. De nombreux systèmes utilisent ce type d'abonnement ([12, 11]) mais ils doivent faire face aux deux problèmes majeurs que l'on rencontre en utilisant ce critère : l'association efficace d'un événement à un grand nombre d'abonnés ; la diffusion efficace des événements à travers un réseau de gestionnaires de messages. Nous n'abordons pas dans cette section les solutions à ces problèmes, celles-ci seront traitées ultérieurement dans le rapport.

Il est à noter qu'il existe une solution légèrement différente du critère *content-based* (notamment abordée dans les environnements de travail tel que Field [77] ou le service de notification Siena [30] ou JEDI [35]), il s'agit d'un critère basé sur un patron d'événement (*pattern-based*). Même si ce modèle est très proche du critère d'abonnement basé sur le contenu, ici, on ne se préoccupe plus des valeurs des différents champs des événements. Le critère devient le modèle, c'est-à-dire le type de l'événement ainsi que le type[1] de ses champs.

1.2.1.3 Modèle événementiel

Le modèle événementiel existe depuis plus d'une dizaine d'années dans les bases de données avec la notion de déclencheur (*Trigger*), puis de règle active E-C-A (Événement, Condition, Action). Le schéma de programmation typique du modèle événementiel est le suivant :

Lorsque	... un événement *E* se produit
Si	... la condition *C* est satisfaite
Alors	... exécuter l'action *A*.

Il est important de comprendre qu'il se crée une association dynamique entre l'événement et la réaction, association qui porte sur le nom de l'événement. Une réaction est un traitement associé à l'occurrence d'un événement, autrement dit une réaction peut être comparée à une procédure qui est appelée lorsque l'événement associé se produit chez le destinataire de l'événement.

Le modèle événementiel utilise explicitement une communication par messages pour envoyer les événements, par envoi ou par file (voir figure 1.4).

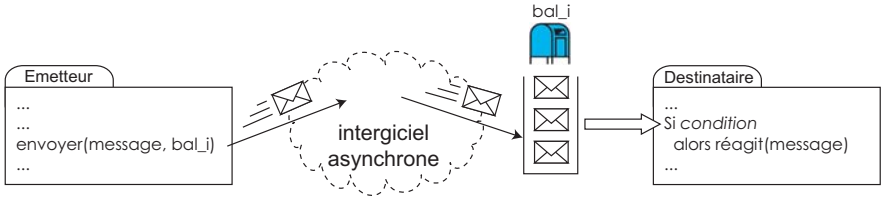

Figure 1.4 – Le modèle événementiel

1.2.1.4 Synthèse des modèles de communication asynchrone

Il est donc possible de classifier les différents modèles comme des couches de communication. L'envoi de message (*message passing*) et les files de messages (*message queuing*) forment la couche la plus basse en fournissant un mode de réception des messages basé sur la « volonté » des destinataires, c'est-à-dire que c'est au destinataire de faire la démarche d'aller chercher son message, c'est ce qu'on appelle le mode *Pull*[2].

Sur cette couche basse se greffent le modèle événementiel et le modèle par abonnement. On a souvent tendance à penser que le modèle événementiel est une couche de communication qui se situe au-dessus du modèle par abonnement. En fait si le modèle par abonnement est une évolution des autres modèles de communication, le modèle par événement est plus un modèle de programmation qu'un modèle de communication. En effet, le modèle événementiel apporte la notion d'envoi d'événement à un (ou plusieurs) destinataire(s) explicite(s) (de 1 vers N). De plus, un système événementiel est basé

[1] Le type des champs se situe au niveau langage de programmation. Ainsi un patron d'événement pourra avoir la forme `[int, bool]`

[2] Du verbe anglais *to pull* : tirer, ramener vers soi.

sur un modèle de réception complètement asynchrone appelé mode *Push*[3], qui consiste à réaliser une réception de manière implicite (la réception d'un événement entraîne l'exécution de la réaction associée). Dans le modèle par abonnement il n'existe pas cette notion de destinataire explicite, l'envoi passe nécessairement par un intermédiaire appelé gestionnaire d'abonnement qui se charge de faire correspondre un événement à son (ses) abonné(s). Il n'y a donc pas de destinataire explicite. Comme dans le modèle événementiel la réception des événements se réalise selon un mode *Push*, mais peut aussi bien se réaliser selon le mode *Pull* comme c'est le cas dans l'interface JMS (les clients viennent prendre périodiquement leurs messages sur le serveur). Une récente et très complète synthèse sur le modèle par abonnement est disponible dans [39].

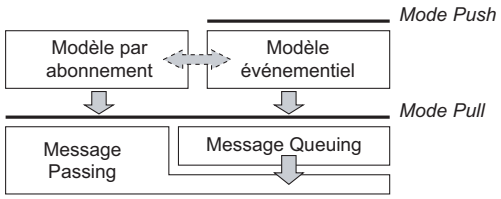

Figure 1.5 – Interaction des différents modèles

Le modèle événementiel est donc un modèle à part entière qui peut se servir du modèle par abonnement comme base de communication mais qui peut tout aussi bien s'appuyer sur les couches inférieures (*message passing* ou *message queuing*). Néanmoins, il apporte en plus la notion de réception purement asynchrone (mode *Push*). De la même manière, le modèle par abonnement peut aussi bien s'appuyer sur le mode *Pull* que sur le mode *Push*. Ces relations sont exprimées sur la figure 1.5.

1.2.2 Architectures et organisation

Nous venons de voir les différentes manières de communiquer à travers un intergiciel asynchrone. Nous allons maintenant aborder les trois types d'architecture possibles des intergiciels asynchrones.

Les messages échangés dans un intergiciel asynchrone empruntent les canaux de communication qui interconnectent les machines sur lesquelles les applications clientes s'exécutent. Il existe trois types d'architecture d'intergiciel asynchrone, le mode centralisé (*hub & spoke*), le mode réparti en point à point ou partiellement maillé (*snow flake*) et enfin le mode réparti en bus (*bus*).

1.2.2.1 Mode centralisé

Cette architecture porte le nom anglais *hub & spoke* car elle se présente comme une roue de vélo, où tous les rayons (*spoke*) sont reliés au même moyeu central (*hub*) (voir figure1.6). Cette architecture est constituée d'un site[4] principal unique qui centralise tous les messages. L'envoi de messages est très simple à réaliser : l'émetteur envoie son message au site central puis celui-ci le transmet dans la boîte aux lettres du destinataire.

Cette centralisation rend l'implémentation de ce type d'architecture assez simple. De plus la mise en place de certaines propriétés comme l'ordonnancement des messages ou la synchronisation devient très facile. Néanmoins, la centralisation du traitement apporte de nombreux désavantages :

[3] Du verbe anglais *to push* : pousser vers qqc.

[4] Dans notre contexte, le mot « site » signifie « instance locale de l'intergiciel asynchrone », c'est-à-dire une machine sur laquelle s'exécute l'intergiciel (ou tout du moins une *partie locale* de l'intergiciel).

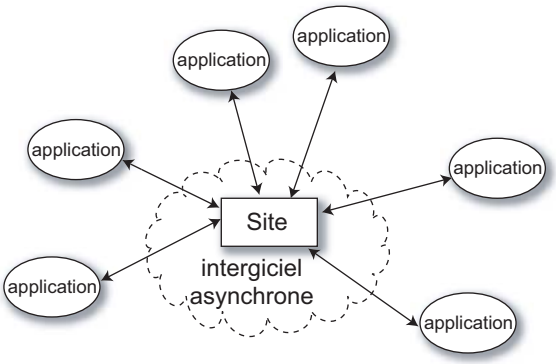

Figure 1.6 – Architecture centralisée

– tout d'abord cela engendre des problèmes de fiabilité : si le site tombe en panne, tout l'intergiciel est en panne et aucun message ne peut circuler ; toutes les applications qui interagissent sont donc bloquées jusqu'au rétablissement du site ;
– ensuite cette centralisation pose de gros problèmes de passage à l'échelle ; en effet, il est assez simple de gérer une trentaine de clients sur le même gestionnaire, mais à partir de plusieurs centaines de clients le site central ne peut plus supporter la charge globale de tous les échanges de messages.

1.2.2.2 Mode partiellement maillé

La deuxième architecture est appelée en anglais *snow flake* à cause de sa topologie en flocon de neige (voir figure 1.7). Contrairement à l'architecture précédente, l'intergiciel asynchrone est distribué sur plusieurs sites distants. Il est donc représenté par un ensemble de sites qui constituent l'intergiciel global, chaque site étant une partie de l'ensemble. Lorsqu'un client (une application) s'adresse au gestionnaire de messages, il s'adresse en fait à l'un des « sites de messages ». L'envoi de messages se réalise de façon transparente pour le client, tout se passe comme s'il n'y avait qu'une seule entité gestionnaire. L'émetteur envoie un message au gestionnaire de messages via un site appelé point d'accès, puis le message est transmis par le site soit directement au destinataire si celui-ci est associé au même site, soit au site auquel est associé le destinataire.

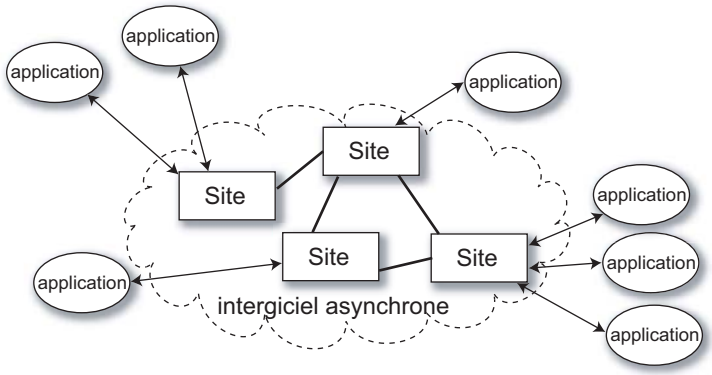

Figure 1.7 – Architecture partiellement maillée

Le gros avantage de cette architecture est la propriété de distribution du service de messages

qui faisait défaut au modèle précédent. Grâce à cette répartition, il devient plus aisé de déployer des applications sur une large zone car la charge générale du gestionnaire de messages est répartie sur l'ensemble des sites, que ce soit la charge CPU ou la charge réseau. De plus, une architecture répartie apporte une fiabilité au système, car si un site tombe en panne, le reste des sites interconnectés peut continuer à fonctionner presque normalement ; l'arrêt de l'un des sites ne pénalise donc pas l'ensemble des applications reliées à l'intergiciel.

La topologie de cette architecture permet de réaliser plus facilement une application à grande échelle que l'architecture précédente. Par contre, surviennent des problèmes liés à la cohérence des données dupliquées dans l'ensemble des sites.

1.2.2.3 Bus

A la manière des deux modèles précédents, le modèle de bus à messages n'est pas forcément une architecture définie au niveau physique. Un bus à messages fait référence à un bus logiciel dans lequel les applications puisent les informations ([70]) ou par lequel les messages s'échangent ([17]). Ce bus logiciel est en fait une vision logique d'une architecture distribuée dont le but est d'acheminer les messages à leurs destinataires. On peut le voir comme une architecture *flocon de neige* qui serait complètement maillée (i.e. où chaque site serait relié à tous les autres, voir figure1.8).

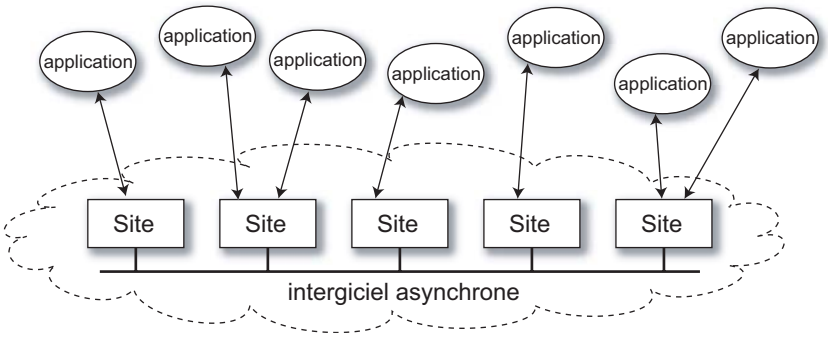

Figure 1.8 – Architecture de type Bus

Un bus logiciel étant mis en oeuvre de façon distribuée, cela implique qu'il existe sur chaque site une entité représentant le bus (un peu comme dans l'architecture précédente). Cette entité appelée bus local est présente dans tous les sites du gestionnaire de messages. Elle est responsable de ses communications locales (au sein d'un même site) mais aussi de l'acheminement d'un message à un client distant. Comme dans l'architecture précédente, l'envoi d'un message à un client distant est complètement transparent pour le client, c'est le bus local qui se charge de transférer le message au bus distant qui lui-même le transmet à son client directement.

L'architecture en bus garde les mêmes avantages que l'architecture en flocon de neige au niveau de la distribution des sites. De plus, elle est plus fiable que la précédente car si un site tombe en panne, il ne « coupe » pas l'intergiciel en deux et le reste des sites peuvent continuer à communiquer (ceux-ci étant tous reliés les uns aux autres). Néanmoins subsistent les problèmes de cohérence dues aux données dupliquées.

1.2.3 Propriétés non fonctionnelles

1.2.3.1 Introduction

On associe souvent les intergiciels asynchrones à leurs caractéristiques de communication : l'asynchronisme des envois de messages, l'architecture distribuée, la communication par abonnement, etc. Mais la plupart des intergiciels asynchrones intègrent aussi des propriétés permettant d'assurer une certaine qualité de service. Dans [87], les auteurs appellent ce modèle transversal le modèle de gestion de fautes des messages (*Message Failure Model*). Nous proposons de le découper en quatre propriétés :

- la propriété de *tolérance aux pannes* permet aux intergiciels de garantir une fiabilité des communications en cas de panne de la machine d'exécution ;
- la propriété de *tolérance aux fautes* permet d'assurer la cohérence de l'exécution des intergiciels ;
- la *sécurité* assure la protection et l'intégrité des informations échangées ;
- l'*ordonnancement des messages* permet d'assurer une cohérence de l'exécution globale des applications clientes.

Les propriétés de fiabilité et d'ordonnancement et leur implantation dans les différents MOM seront traitées dans la suite.

1.2.3.2 Propriétés

Succinctement, les propriétés d'exécution d'un intergiciel distribué se subdivisent en propriétés d'interaction caractérisant le protocole de communication et en propriétés non fonctionnelles. Dans les intergiciels, ces propriétés regroupent notamment les modèles de tolérance aux fautes et de tolérance aux pannes.

Dans la majorité des intergiciels asynchrones actuels, l'entité « gestionnaire de messages » (l'intergiciel dans son ensemble) est vue comme une entité abstraite fiable qui permet de garantir la délivrance des messages. Cette garantie est implémentée grâce à des propriétés de persistance et d'atomicité.

Persistance : La norme CORBA Notification Service ([69]) définit la persistance des messages à partir de quatre types de politique de délivrance : au-mieux, au-plus-une-fois, au-moins-une-fois et exactement-une-fois. Ces politiques de délivrance sont basées sur l'association de la fiabilité des événements et de la fiabilité des connexions. A chacune des ces propriétés peut être associé un niveau de fiabilité qui peut être soit persistant soit non persistant.

- Une politique de délivrance *au-mieux* consiste à avoir une fiabilité non-persistante des événements et des connexions, c'est le cas de SIENA [29] ;
- MQSeries utilise lui une fiabilité des événements (persistance, avec une file de messages) mais une non-persistance des connexions ; c'est une politique *au-moins-une-fois* ;
- Une politique *au-plus-une-fois* consiste à maintenir une fiabilité des connexions (persistance) mais pas des événements ;
- Lorsque l'on applique une fiabilité de persistance des connexions et des événements, on obtient une politique de délivrance *exactement-une-fois* ; c'est le cas de MSMQ [46], et A3 [17, 36].

Atomicité : Cette propriété est une autre composante très importante de la qualité de service car elle permet de garantir une cohérence de l'application. Elle utilise une gestion de transactions, celles-ci permettant des opérations d'extraction et d'insertion de messages dans une même unité logique de traitement. Toutes les opérations d'une séquence sont réalisées au sein d'une transaction ; en cas

de terminaison correcte de la séquence, il y a garantie de la conservation des changements liés à ces opérations, sinon il y a annulation de tous les changements effectués. Le lecteur pourra se reporter à l'article [87], dans lequel les auteurs proposent quatre stratégies pour intégrer les transactions au sein d'un intergiciel asynchrone en fonction de son modèle de communication.

L'atomicité et la fiabilité permettent donc de garantir la délivrance des messages et les traitements, et ainsi d'assurer une qualité de service de l'intergiciel asynchrone.

1.2.3.3 Ordonnancement

Dans les environnements distribués comme les intergiciels asynchrones, le non-déterminisme est très présent, notamment à cause de l'asynchronisme et du contexte d'environnement distribué. Dans n'importe quel gestionnaire de messages, il est impossible de prédire précisément quand un message envoyé par une application sera reçu par une autre. Ceci est particulièrement vrai dans des réseaux à grande échelle (sur Internet par exemple). Un modèle réaliste suppose donc que l'ordre d'arrivée des informations envoyées est complètement arbitraire donc entièrement inconnu. En particulier, les communications de différentes sources vers un destinataire donné peuvent lui arriver dans un ordre indépendant de l'ordre d'envoi de chaque source. C'est donc cette absence de cohérence qui pousse à la nécessité de fournir un mécanisme qui ordonne les messages à l'arrivée.

La plupart des implémentations actuelles d'intergiciels asynchrones ne proposent pas de politique d'ordonnancement des messages, au mieux ils utilisent des canaux d'échange FIFO afin de garantir un ordonnancement entre deux sites [70]. S'ils nécessitent une réelle politique d'ordonnancement, par exemple en raison des besoins des applications qui s'appuient dessus, ils utilisent une architecture centralisée, avec des canaux de communication FIFO entre les clients. L'utilisation d'une telle architecture permet, de fait, de garantir un ordonnancement, car toutes les informations passent par le site central et sont redistribuées au(x) destinataire(s) dans l'ordre d'arrivée.

Obtenir un ordonnancement sur des architectures distribuées est beaucoup plus difficile. Dans les systèmes distribués asynchrones il n'existe pas de borne pour la vitesse relative des processus ni pour le délai de propagation des messages. La communication est le seul mécanisme possible pour la synchronisation de tels systèmes [9]. Mais l'absence d'horloge globale temps-réel oblige à créer un temps logique afin d'ordonner les messages selon un ordre causal la plupart du temps.

L'ordre causal (ou relation de précédence causale) des événements d'un système est un concept fondamental qui permet d'aider à résoudre les problèmes d'ordonnancement dans les systèmes distribués. L'ordre causal utilise le principe des horloges logiques introduit par Lamport dans [56].

Dans un système à horloges logiques, chaque processus a sa propre horloge logique et chaque événement[5] se voit affecter une estampille (égale à l'horloge logique du processus émetteur). On obtient la propriété suivante : « Si un événement a affecte causalement un événement b, alors l'estampille de a est plus petite que celle de b ». Cette propriété essentielle permet d'obtenir un ordre sur les événements en fonction de leur estampille.

Remarque : propriétés des horloges (voir [9], [76]).

Conventions :
- e_i est l'événement numéro i sur un processus ;
- $C(e_i)$ = estampille de l'événement i ;

[5]Un événement est caractérisé par l'envoi d'un message (envoyer(mes), abonner(mess), publier(mess), etc.) ou la réception d'un message (recevoir(mess), réagir(mess), etc.)

– la relation $\rightarrow$ signifie « précède causalement » ;
– la relation $<$ définit la notion d'ordre sur les estampilles[6].

On a alors les propriétés suivantes :

– Propriété d'horloge cohérente : $e1 \rightarrow e2 \Rightarrow C(e1) < C(e2)$
– Propriété d'horloge fortement cohérente : $e1 \rightarrow e2 \Leftrightarrow C(e1) < C(e2)$

Il existe plusieurs types d'horloges logiques : les horloges logiques simples (temps scalaire), les horloges logiques vectorielles (temps vectoriel) et enfin les horloges logiques matricielles (temps matriciel). L'implémentation d'une horloge logique simple ne permet d'obtenir que la propriété d'horloge cohérente, qui ne permet de connaître l'ordre des événements qu'a posteriori et permet à un processus de ne mesurer que sa propre progression. L'utilisation d'une horloge vectorielle permet d'avoir la propriété d'horloge fortement cohérente, qui permet de facilement reconstituer une trace d'exécution globale mais seulement a posteriori[7]. L'algorithme le plus complexe dans le cadre d'ordonnancement de messages asynchrones utilise des horloges matricielles. Celle-ci respecte la propriété d'horloge fortement cohérente, et permet en plus de savoir au moment de la réception d'un événement si celui-ci dépend causalement d'un autre événement qui n'est pas encore arrivé sur le site. Dans ce cas la délivrance du message est retardée (celui-ci est stocké dans un tampon) jusqu'à la réception de l'événement précédent (au sens causal).

1.2.4 Synthèse

Ce chapitre a proposé une vision globale des intergiciels asynchrones, leurs différents modèles de communication, architectures et propriétés.

Pour résumer, il existe quatre modèles de communications, tous basés sur l'envoi asynchrone de messages, qui permettent de faire communiquer des applications fortement découplées dans des environnements hétérogènes :

– la *communication par envoi de messages*, qui permet l'envoi de messages unidirectionnel dans des queues de messages non sécurisées ;
– la *communication par file de messages*, qui possède les même propriétés que la précédente mais apporte en plus la notion de fiabilité des queues de messages (propriété de persistance) ;
– le *modèle par abonnement* qui permet de réaliser un envoi de 1 vers N selon des critères d'abonnement ;
– le *modèle événementiel*, qui se rapproche plus d'un modèle de programmation, suit le modèle des règles actives E-C-A des SGBD.

Il existe trois types d'architecture qui assurent le déploiement des intergiciels asynchrones sur n'importe quel type de topologie réseau :

– la topologie *centralisée* avec un seul site représentant l'intergiciel ;
– la topologie en *flocon de neige* qui est une architecture distribuée représentée par un ensemble de sites partiellement interconnectés ;
– la topologie en *bus* qui est une architecture distribuée représentée par un ensemble de serveurs interconnectés à la manière d'un LAN (c'est-à-dire un graphe de serveurs complètement maillés).

[6]Pour une horloge scalaire, $HL_1 < HL_2$ signifie que la valeur de HL_1 est plus petite que celle de HL_2, et pour les horloges matricielles et vectorielles que chaque élément de HL_1 est plus petit que l'élément homologue de HL_2.

[7]On peut connaître l'ordre causal d'événements au moment de leur réception avec des horloges vectorielles à condition d'utiliser une diffusion fiable des événements, mais à ce principe on préfère l'utilisation d'horloge matricielle.

Enfin, des propriétés assurent la fiabilité et la cohérence des envois de messages. Il existe plusieurs politiques de qualité de service permettant de garantir certaines propriétés sur les intergiciels asynchrones :

- la *fiabilité* sur les connexions et sur les queues de messages permet d'assurer la garantie de délivrance des messages et de tolérance aux pannes ;
- l'*atomicité* permet de garantir une certaine tolérance aux fautes ;
- et une *politique d'ordonnancement* assure une cohérence des échanges de messages.

Grâce à toutes ces caractéristiques, les intergiciels asynchrones sont souvent comparés à une technologie de construction par collage d'applications distribuées (*glue technology*, voir [11]). Cette expression exprime clairement l'intérêt des intergiciels asynchrones, c'est-à-dire l'intégration d'applications en environnements hétérogènes distribués à grande échelle.

Le développement actuel des grands réseaux et l'intégration au sein d'applications réparties de plates-formes hétérogènes et mobiles augmentent encore le degré d'utilisation des intergiciels asynchrones.

Nous allons voir dans la section suivante une présentation des principaux intergiciels asynchrones dans les milieux académiques puis industriels. L'objectif de cette section est d'étudier les avantages et les inconvénients des choix technologiques fait par chaque projet afin de mettre en avant les motivations qui nous ont poussé à notre proposition du chapitre 5.

1.3 Etude de cas

Cette section propose une description rapide de différents projets académiques, c'est-à-dire des intergiciels asynchrones réalisés en milieu universitaire ou dans les centres de recherche & développement de grandes entreprises ; puis de différentes réalisations industrielles, c'est-à-dire des intergiciels asynchrones développés à des fins commerciales par de grandes entreprises. Nos études de cas ne sont pas exhaustives ; bien au contraire, nous nous sommes attardés à ne présenter que les cas les plus représentatifs dans leur domaine. Cette section a pour but de dresser un état de l'art sur les intergiciels asynchrones sans forcément rentrer dans les détails d'implémentation de chacun. Nous souhaitons montrer que les projets actuels se concentrent essentiellement sur les modèles de communication mais ne se préoccupent pas ou peu des problèmes d'adaptabilité des intergiciels asynchrones, question à laquelle nous tentons de répondre dans les chapitres suivants. Cette étude nous permet, enfin, de mettre en avant l'aspect monolithique des intergiciels asynchrones.

1.3.1 Projets académiques

La notion d'interaction asynchrone est un des plus vieux mécanismes de coordination depuis le développement des systèmes bas niveau relié directement aux mécanismes d'interruptions matérielles. Mais l'absence de structures et de primitives de haut niveau rendait ces systèmes fragiles et peu fiables. Dans les années 70, les premiers modèles de communication asynchrone à base d'échange de messages ont fait leur apparition. Les années 80 et 90 et le développement grandissant des réseaux ont permis de voir apparaître les travaux autour des notions de queue de messages, d'événements, de système par abonnement, etc. qui ont posé les bases des intergiciels asynchrones actuels.

Depuis lors, les projets autour des intergiciels asynchrones se sont essentiellement centrés sur la recherche autour des modèles de traitement de messages (abonnement, événement, etc.) ou des modèles de délivrance de message (architectures, propriétés, etc.) avec souvent le même but : le

passage à grande échelle (ou scalabilité). Mais l'implémentation des intergiciels asynchrones actuels reste souvent figée et ils n'offrent souvent qu'une API de communication. Les projets ne se préoccupent pas ou peu des problèmes de configuration et de déploiement de l'intergiciel.

Nous présentons dans cette section quatre projets qui montrent les travaux réalisés autour de ces modèles.

1.3.1.1 JEDI

L'école polytechnique de Milan a développé une infrastructure orientée objet basée sur une communication à événements appelée JEDI (Java Event-based Distributed Infrastructure) [35]. L'architecture de JEDI utilise la notion d'objet actif (*Active Object*, AO). Un AO est une entité autonome en charge d'une tache applicative spécifique. Les AO communiquent entre eux en consommant et produisant des événements. Un événement est un type de message spécifique généré par un AO et notifié aux autres AO selon le modèle par abonnement. Les événements sont acheminés d'un producteur à un consommateur par l'intermédiaire d'une infrastructure spécifique responsable de la dissémination appelée *Event Dispatcher* (ED) utilisant une architecture logique en bus (voir figure1.9).

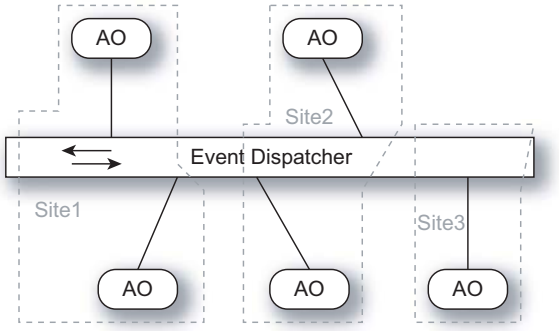

Figure 1.9 – Architecture de JEDI.

Les AO peuvent utiliser deux types d'abonnements : soit l'abonnement basé sur le nom d'un événement (*subjet-based*), soit basé sur un patron d'événement (*pattern-based*). Un patron d'événement est un ensemble de chaînes de caractère ordonnées représentant une expression régulière. Un événement étant composé d'un nom puis de paramètres (= données), un événement e correspond à un patron p ssi les conditions suivantes sont satisfaites :

- Le nom de e est équivalent au nom de p où le nom de p est constitué d'un préfixe puis d'une étoile (caractère `*`) et e et p ont le même préfixe.
- e et p ont le même nombre de paramètres.
- Chaque paramètre de p qui n'est pas égal à "_" est le même que le paramètre correspondant pour l'événement e.

Il existe deux types d'AO : les objets actifs génériques et les objets actifs réactifs. Les premiers gênèrent des événements puis se mettent en attente active (bouclent) sur l'arrivée d'événements provoquant un coût de traitement élevé. Les objets actifs réactifs sont basés sur le modèle événement-réaction et ne sont « réveillés » par le système que lors d'arrivée d'événements. Les objets réactifs ont la faculté de se déplacer d'un site à l'autre, l'ED se chargeant de router les messages vers le nouveau site de l'AO.

JEDI est un projet intéressant car il implémente la notion d'événement par abonnement sur une architecture en bus scalable et permet les deux types d'abonnement basés sur le sujet ou sur le patron ($\approx$ sur le contenu) de l'événement. Néanmoins, il possède sa propre interface de programmation qui

ne permet pas d'interagir avec d'autres intergiciels asynchrones. De plus, les événements échangés ne peuvent être que de type Java simple et il n'est pas possible d'utiliser des types d'événements complexes comme des graphes d'objets. Cela réduit, certes, le coût des messages échangés et donc améliore la scalabilité de l'ensemble mais limite grandement l'interopérabilité et surtout le spectre d'application possible (l'utilisation de chaînes de caractères structurées comme l'XML serait plus adaptée à l'échange d'information).

L'aspect configuration de l'intergiciel n'est pas traité par JEDI. Il n'est pas possible de manipuler sa structure interne pour modifier son comportement. Le bus à événement (ED) est fourni avec l'ensemble de ses propriétés et de ses fonctionnalités sans changement possible. De plus, même si l'architecture hiérarchique des sites (formant le bus global) améliore la scalabilité, elle reste figée et ne peut pas s'adapter à d'autre besoins.

1.3.1.2 IBM Gryphon

Le projet Gryphon d'IBM Research Center a pour but le développement d'un intergiciel asynchrone essentiellement centré sur un gestionnaire de diffusion par abonnement [12]. Leur système utilise un algorithme spécifique permettant d'accélérer la distribution des messages. Cette diffusion est basée sur le principe de l'abonnement basé sur le contenu (voir section 1.2.1.2). Les clients de Gryphon accèdent à l'intergiciel par l'intermédiaire de l'API JMS.

Les motivations de Gryphon sont axès sur trois propriétés : la scalabilité (le passage à grande échelle), la disponibilité et la sécurité.

- Scalabilité : Afin d'améliorer le passage à grande échelle, Gryphon utilise simplement la distribution de serveurs de messages interconnectés. Gryphon a été l'un des premiers à utiliser la répartition sur des sites distants pour répartir la charge dûe à l'augmentation du nombre de client. La connectivité des serveurs est assurée par une architecture *snow flake*.
- Disponibilité : Le système assure une disponibilité du service dans son ensemble en informant automatiquement les clients d'une panne de serveur pour les réorienter vers un autre et cela de façon automatique.
- Sécurité : La sécurité du système est assurée par plusieurs types de connexion possibles des clients. Soit le client utilise un simple mot de passe, soit il utilise une authentification mutuelle sécurisée soit il se connecte avec du symétrique ou asymétrique SSL. De plus, les messages peuvent être cryptés par l'utilisation de méthodes standards (à clés, etc.).

Beaucoup de travaux de recherche ont été modélisés dans cette équipe mais peu ont réellement été développés sur l'intergiciel Gryphon. La plupart des articles de ces dernières années ne présentent que des travaux futurs (comme sur l'utilisation de la réflexivité) mais pas de réalisations concrètes.

Le système Gryphon a été utilisé, avec succès, pour des événements de grande ampleur[8], confirmant son potentiel de passage à grande échelle. Néanmoins, il ne reste « qu'un » intergiciel asynchrone ne proposant que le mécanisme d'abonnement et ne permet pas la combinaison de plusieurs modèles de traitement de messages. Il se cantonne à proposer une implémentation de JMS un peu plus scalable que les autres grâce à son algorithme de diffusion performant. Mais, il ne permet pas de modifier son comportement pour l'adapter à telle ou telle application ou système. L'aspect déploiement n'est pas non plus abordé dans Gryphon et il est nécessaire que les sites soient tous actifs avant de démarrer les échanges de messages.

[8] La diffusion des résultats sportifs en temps réel de l'Open de Tennis US ou pour les jeux olympiques de Sydney 2000.

1.3.1.3 SIENA

SIENA est un intergiciel asynchrone responsable de la distribution d'événements entre applications sur un très grand réseau. SIENA se classe parmi les services à notification d'événements [29], il est responsable de la sélection de notifications auxquelles les clients se sont abonnés puis de leurs disséminations. Il rejoint les travaux de JEDI en poussant plus en avant le mode de communication par abonnement basé sur les patrons d'événements. Mais à la différence de JEDI, il n'y a pas de modèle de composants des clients (les AO), les clients sont, soit des producteurs (*object of interest*), soit des consommateurs d'événements (*interested party*). Ils utilisent des points d'accès sur l'architecture *snow flake* de l'intergiciel.

La force de SIENA réside dans son puissant modèle de données permettant la gestion de l'abonnement basé sur le patron des événements. Un événement (ou notification) est constitué d'un ensemble d'attributs ayant chacun un type, un nom et une valeur. Le type d'un attribut est volontairement un type basique (booléen, entier, etc.) afin de diminuer le coût de transport des notifications et de simplifier le modèle de correspondance.

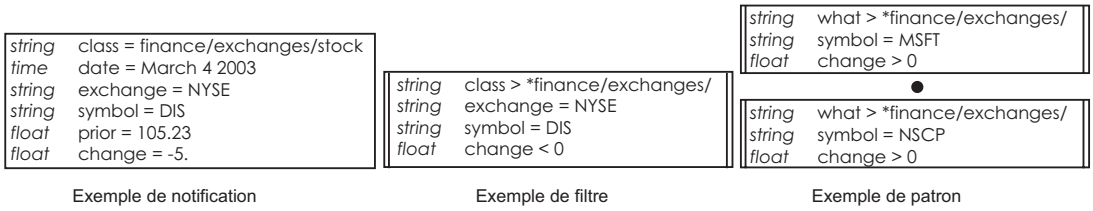

Figure 1.10 – Modèle de donnée de SIENA.

Il existe deux mécanismes d'abonnement : la sélection peut se faire selon un filtre à événement ou selon un patron d'événement (voir figure 1.10). Un filtre est un simple ensemble d'attributs et de contraintes sur la valeur de ces attributs. SIENA introduit l'opérateur $\sqsubset$ pour signifier qu'un événement correspond (ou « couvre ») un filtre. La figure 1.10 montre un exemple de filtre qui correspond à une diminution du prix pour l'action DIS à la bourse de NewYork. La sélection selon un patron est en fait la correspondance d'un événement en fonction d'un ensemble de filtres ordonnés. La figure 1.10 donne l'exemple d'un patron qui permet d'obtenir les événements qui donnent l'augmentation du prix du MSFT suivit par l'augmentation du prix de NSCP.

SIENA propose donc un puissant modèle de données associé à la définition de relations complexes pour la correspondance des événements aux filtres et aux patrons. Néanmoins, ce modèle se limite à l'utilisation de notification ne comportant que des types simples et pas d'objet(s) complexe(s). De plus, la diffusion des messages se fait selon une politique au mieux (*best-effort*) qui ne permet pas de proposer une qualité de service suffisante pour beaucoup d'applications. SIENA ne propose qu'une simple interface de communication pour les applications et ne permet pas l'interopérabilité avec les autres intergiciels, par l'utilisation de la spécification JMS par exemple.

Comme pour les projets précédents, les aspects configuration et déploiement de l'intergiciel ne sont pas abordés dans SIENA. Les travaux se concentrent essentiellement sur le traitement et la délivrance des messages.

1.3.1.4 A3

Cette section présente l'intergiciel asynchrone A3 (Agent Anytime Anywhere [17]), aussi appelé bus à agents, développé au sein du laboratoire SIRAC dans le cadre du Groupement d'Intérêt Economique Dyade. A3 est à la fois un intergiciel asynchrone responsable de l'envoi de messages et un

modèle de programmation d'application à base d'agent.

Le modèle d'agent Les agents sont des objets réactifs autonomes qui se comportent conformément au modèle « événement → réaction ». Dans ce modèle, un événement est représenté par un message typé appelé notification. Lorsqu'un agent reçoit une notification, il doit exécuter la réaction appropriée. L'envoi de notifications représente le seul moyen de communication et de synchronisation entre les agents. Les événements peuvent être envoyés à un agent ou à un « Rôle » qui regroupe un ensemble d'agents[9]. L'envoi de notifications s'effectue au travers d'un bus logiciel qui assure certaines propriétés sur les communications des agents et aux agents eux-mêmes.

Propriétés des agents Les agents A3 sont *persistants* : leur état est régulièrement sauvegardé sur un support persistant. La réaction des agents est *atomique* : une réaction est, soit complètement exécutée, soit annulée (dans ce cas l'agent est rechargé à son état initial).

Propriétés de la communication Au niveau de l'infrastructure de communication, les notifications sont transformées en messages. Cette infrastructure de communication est basée sur un bus logiciel asynchrone qui assure des propriétés d'asynchronisme, de fiabilité (garantie de délivrance et persistance des messages, *Message Queuing*)) et d'ordonnancement (délivrance causal des messages) sur les communications.

Infrastructure d'exécution A3 L'infrastructure A3 est basée sur un bus à messages qui a pour rôle d'acheminer les notifications et de provoquer la réaction de l'agent destinataire. Ce bus à messages est mis en oeuvre de façon distribuée où chaque bus local est représenté par un serveur d'agents (voir figure 1.11).

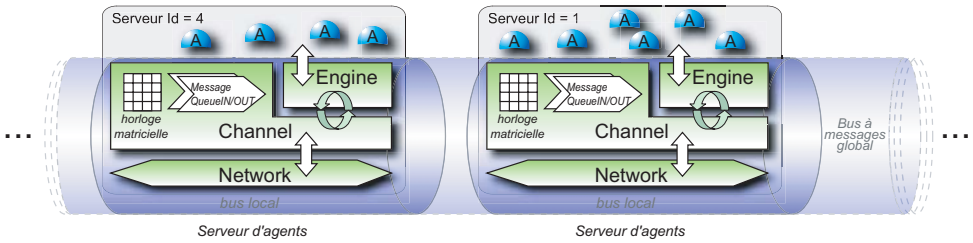

Figure 1.11 – Exemple de serveurs d'agents.

Un serveur d'agents est une machine virtuelle servant d'hôte aux agents. Il a en charge la création des agents, leur exécution et leur communication. Pour cela, chaque serveur d'agents comporte un bus local et une fabrique d'agents. En pratique, un serveur d'agents est inclus dans un processus. Chaque serveur d'agents contient un bus à messages (*Channel*), un moteur d'exécution (*Engine*), et un composant réseau (*Network*). Grossièrement, le bus à messages a la charge de la localisation des agents destinataires d'événements et le composant réseau de la transmission des événements sur le réseau. Quant au moteur d'exécution, il délivre aux agents les événements reçus en appelant leur méthode `react` et garanti les propriétés sur les agents.

L'intergiciel asynchrone A3 propose aussi un ensemble d'outils pour construire des applications distribuées sous forme de composants et de connecteurs. Pour cela il utilise un puissant langage de description appelé OLAN (voir section 3.6).

[9] La définition exacte d'un Rôle peut être trouvée dans [36]

Le projet A3 est donc plus qu'un simple intergiciel asynchrone. Il permet de développer des applications complexes et propose un ensemble de propriétés sur les composants applicatifs et leurs communications. Il est assez proche de JEDI par les concepts d'agent (≈ AO) et de bus à messages (≈ ED) mais a l'avantage de proposer des outils d'aide au développement grâce à une sémantique plus poussée des composants applicatifs et à l'utilisation d'un ADL. La notion d'abonnement selon l'API JMS est fournie par le biais d'agents particuliers implémentés dans JORAM (voir section suivante). De plus, il propose un déploiement simplifié de l'intergiciel par l'utilisation de serveurs « bootstrap » présents sur les sites d'exécution et pouvant mettre en place l'infrastructure de l'intergiciel. Mais la configuration des propriétés et fonctionnalités n'est pas possible et l'intergiciel reste donc monolithique et figé dans sa structure.

1.3.2 Réalisations industrielles

Les intergiciels asynchrones industriels sont, pendant longtemps, restés très peu nombreux car peu de personnes en voyaient l'utilité. Mais l'arrivée des systèmes répartis à grande échelle les ont mis en avant comme une des solutions les plus adaptées et de très nombreuses implémentations sont apparues (BEA MessageQ, DEC MessageQ, TIBCO Rendezvous, FioranoMQ, Sonic MQ, SoftWired iBus/MessageBus,...). La proposition d'interface de programmation standard de SUN (JMS) a permis un essor encore plus important de ces technologies en améliorant leur interopérabilité.

Nous présentons dans cette section les principales implémentations d'intergiciels asynchrones industriels. Tout d'abord IBM MQSeries (devenu aujourd'hui partie intégrante de la suite logiciel WebSphere Business Integration) qui est la plus ancienne réalisation industrielle et la première à montrer que l'utilisation des intergiciels asynchrones est capitale dans l'EAI (*Enterprise Application Integration*[10]) [8]. Nous abordons ensuite Microsoft Message Queuing qui est de plus en plus utilisé dans les entreprises à cause son intégration dans Windows. Une vue d'ensemble des spécifications de Corba Messaging, l'extension de CORBA pour l'invocation asynchrone, sera ensuite abordée. Puis cette section termine sur Java Messaging Service, le standard proposé par SUN pour faire interopérer les différentes implémentations d'intergiciels asynchrones.

1.3.2.1 IBM MQSeries (WebSphere Business Integration)

MQseries est un des plus vieux produits d'intergiciel asynchrone existant. Il a rencontré un grand succès surtout grâce à son portage sur de nombreux systèmes d'exploitation même les plus atypiques. Ses points forts comportent notamment la fourniture d'une interface de programmation commune (MQI API) et un haut niveau de fonctionnalité. MQSeries est construit sur la notion de gestionnaire de messages chargé de gérer les files d'attente et leurs messages pour les applications clientes (voir figure 1.12).

Une queue de messages peut être prédéfinie ou créée dynamiquement (de façon temporaire ou permanente). Une queue de message appartient au même gestionnaire de messages que l'application qui lui est connectée. Le service de transmission de message (*Message Moving Service*) possède une queue de message persistante dans laquelle il stocke les messages en attente de transmission vers d'autres gestionnaires distants. Les différents gestionnaires de message communiquent avec les autres grâce à un programme entre leurs services de transmission appelés « canal de messages » (*Message Channel Agent*, MCA). Les MCA peuvent garantir la fiabilité et la sécurité des données échangées. MQSeries permet de réaliser certaines garanties sur les messages et propose pour cela

[10] Modèle d'intégration, l'EAI est une vision qualitative et organisationnelle pour le système d'information d'une entreprise faisant collaborer les diverses applications.

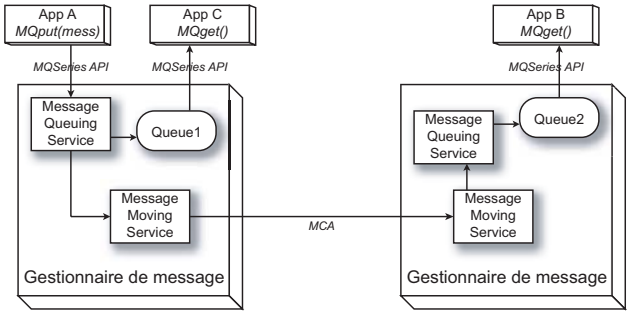

Figure 1.12 – Intergiciel à messages IBM MQSeries.

plusieurs politiques de qualité de service comprenant la persistance des messages, leur synchronisation (ordonnancement), leur sécurité (cryptage).

La réception des messages par une application cliente peut se faire selon les modes *Pull* ou *Push*. Soit l'application décide d'aller, régulièrement, chercher les messages dans sa file d'attente (mode *Pull*) ; Soit elle utilise le mécanisme de déclenchement de MQSeries (mode *Push*, appelé mécanisme de *triggering*). Cependant, MQSeries ne propose pas de mécanisme de diffusion par abonnement (*Publish/Subscribe*).

MQSeries est donc un des intergiciels asynchrones les plus complets du marché. Son interface JMS lui permet de communiquer avec d'autres intergiciels asynchrones (alors que pendant longtemps l'interaction était limitée à l'interface spécifique MQI). Néanmoins, cet avantage se transforme assez vite en défaut lorsqu'il s'agit de réaliser des applications spécifiques sur des systèmes très ciblés. De plus, il reste très statique et la gestion de la distribution décrite ci-dessus reste extrêmement complexe car nécessitant des opérations d'administration coûteuses. Son aspect monolithique le transforme en véritable « usine à gaz » inférant des temps de latence et une surcharge du coût très importante lorsqu'il s'agit de modifier son comportement (distribution,...). L'aspect configuration de MQSeries est limité à quelques interfaces d'administration permettant de modifier son comportement statique local et mettre en place de la qualité de service sur les files de messages. De plus, les mécanismes de configuration ne sont accessibles que par de complexes lignes de commande et MQSeries ne propose pas d'aide à la configuration (par l'intermédiaire de fichier de configuration par exemple).

1.3.2.2 Microsoft MSMQ (.NET Messaging)

Microsoft a intégré dans sa gamme de produits un intergiciel à base de files de messages nommé MSMQ pour *MicroSoft Message Queuing* [46]. Ce service a fait son apparition dans Windows NT4 en octobre 1998 mais son architecture a été complètement modifiée dans sa version 2.0 pour mieux s'intégrer à la plate-forme Windows 2000 puis dans le canevas logiciel .NET.

Les grands principes de cette architecture sont les suivants : les files publiques sont accessibles par n'importe quel type de client, elles sont gérées par les serveurs MSMQ et enregistrées dans un annuaire (*Active Directory*). Chaque serveur gère physiquement un ensemble de files, les émissions de messages vers des serveurs distants utilisent le mécanisme de *Store and Forward*[11]. Il existe deux types de clients : les clients dépendants qui ne peuvent fonctionner que s'ils sont connectés à un serveur MSMQ et les clients indépendants. Les clients indépendants peuvent travailler en mode déconnecté ; les messages sont stockés dans une file locale puis re-synchronisés avec le serveur quand la connexion est disponible (*store and forward*). Il est de plus possible de définir des files privées ; ces

[11] *Store and Forward* = sauvegarde du message en attente de disponibilité du destinataire.

files sont disponibles sur les clients indépendants et ne sont pas publiées dans la topologie du réseau ; on peut les utiliser indépendamment des serveurs MSMQ et on a dans ce cas des communications directes. Une fois installé, MSMQ propose un ensemble d'outils d'administration très puissants permettant de gérer l'ensemble des files. En particulier, des outils graphiques permettent de créer des files ou éditer un message.

L'implémentation de MSMQ est très contraignante. En effet, les serveurs MSMQ doivent absolument être installés sur des machines Windows 2000 Server (ou Windows XP Pro) et au moins l'un des serveurs doit être contrôleur de domaine[12] (PDC, BDC). De plus, les clients doivent être dans le même domaine que le serveur qu'ils utilisent ce qui peut poser des problèmes de configuration ou de déploiement d'applications. Lors de l'installation, si aucun serveur n'est disponible dans le domaine, on ne pourra créer que des clients indépendants fonctionnant avec des files privées locales. La gestion du mode par abonnement n'est pas supportée par MSMQ. Microsoft propose une technologie similaire dans sa gamme d'outils COM+ (plus précisément les événements COM+), même si cette technologie implémente ce mode de communication il ne s'agit pas d'échange de messages mais d'invocation de méthodes. Malgré ces qualités, le Framework .NET n'est pas disposé à offrir un niveau d'abstraction suffisant (à la JMS) en privilégiant une implémentation et des interfaces spécifiques.

Le déploiement de MSMQ est limité par la nécessité, pour les sites, de faire partie d'un même domaine Windows. De plus, comme les autres intergiciels asynchrones, MSMQ est à prendre tel quel et il n'est pas possible de le configurer afin de spécifier une architecture spécifique. Seule l'administration des files de messages est possible et permet de configurer leurs propriétés de qualité de service.

1.3.2.3 Corba Messaging

Les spécifications de CORBA Messaging ([69]) introduisent quelques caractéristiques importantes à CORBA, dont les trois principales sont : l'invocation asynchrone de méthode (*asynchronous method invocation*, AMI), l'invocation indépendante du temps (*time-independent invocation*, TII), et les politiques de QoS sur les messages.

La spécification CORBA AMI définit deux modèles, le *polling* et le *callback* :

– Dans le *Polling Model*, lorsqu'un client fait un appel de méthode, il récupère un objet local spécifique appelé *poller* qui va attendre la réponse de l'objet cible. Le client peut donc régulièrement aller chercher la réponse sur le *poller* (ou se mettre en attente du retour sur le *poller*). Le *Polling Model* est illustré sur la figure 1.13.

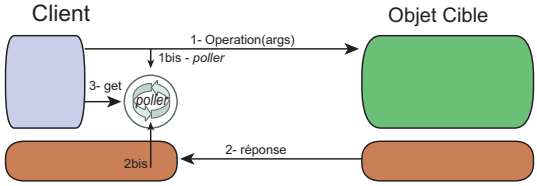

Figure 1.13 – Modèle *Polling* dans Corba AMI.

– Dans le *Callback Model*, le client invoque l'opération sur l'objet et donne une référence d'objet local responsable du retour d'invocation (le *reply handler*) puis reprend la main. Lorsque le serveur retourne, l'ORB client reçoit la réponse et la transmet au *reply handler* approprié fourni par le client. Le *Callback Model* est illustré sur la figure 1.14.

[12] « domaine » au sens « domaine réseau Microsoft Windows »

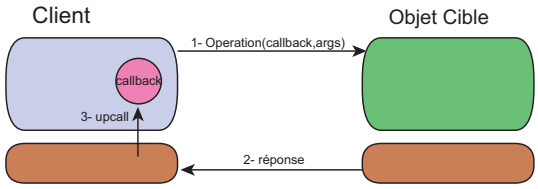

Figure 1.14 – Modèle *Callback* dans Corba AMI.

Les deux modèles AMI ont l'avantage de ne pas nécessiter de thread supplémentaire chez le client, permettant à l'application de gérer plusieurs invocations simultanément.

La spécialisation TII de CORBA Messaging permet de supporter la sémantique du *store and forward*, autrement dit le *message queuing*. En utilisant le TII, les requêtes et réponses sont délivrées lorsque la connexion réseau, le routage, etc. le permettent. Pour cela, TII définit un protocole de routage standard (IRP) pouvant être interfacé avec les routeurs de différents fournisseurs et même des MOM.

Malgré ces différents modes, Corba Messaging reste avant tout un ORB. Son but reste de faire de l'appel de méthode à distance mais ne propose pas de définition plus poussée de la sémantique de délivrance ou surtout de traitement de messages comme dans un véritable intergiciel asynchrone. Seule la notion de QoS sur certaines caractéristiques du TII comme la gestion des queues de messages ou la priorité de message permet de manipuler un peu la délivrance mais ne suffit pas à fournir les propriétés attendues d'un intergiciel asynchrone.

Une implémentation existe sur TAO [7] mais reste pour l'instant figée[13]. Il n'est pas possible de configurer les fonctionnalités de l'intergiciel et le déploiement de l'intergiciel n'est pas pris en charge, chaque site devant être déployé « à la main ».

1.3.2.4 JMS (*Java Messaging Service*)

Sun a choisi vis à vis des éditeurs d'intergiciels asynchrones une politique similaire à celle de JDBC vis-à-vis des éditeurs de bases de données : définir un modèle de programmation basé sur un ensemble d'interfaces portables quelle que soit l'implémentation utilisée. Le but est de normaliser l'accès aux intergiciels asynchrones indépendamment de la structure et de l'architecture interne de l'outil. La spécification de SUN s'appelle JMS pour *Java Messaging Service* [44], elle traite des modes de communication *MessageQueueing* et *Publish/Subscribe* (par abonnement) que nous avons vus précédemment. Elle a été développée par les principaux acteurs de ce domaine (IBM, Tibco, BEA,...) et possède de nombreuses implémentations.

Une brève présentation de JORAM, une implémentation Open Source de JMS dans le cadre du consortium ObjectWeb, est donnée dans la suite.

JMS à travers JORAM JORAM est une implémentation Open source des spécification JMS de SUN. JORAM est développé par la société Scalagent (start-up de l'ancienne équipe A3 présentée dans la section 1.3.1.4) dans le cadre du consortium ObjectWeb. JORAM fournit un intergiciel asynchrone à base de messages construit sur la plate-forme distribuée A3.

JORAM fournit un support complet des spécifications de JMS dans lesquelles il existe deux styles de gestion de messages : le modèle de message point à point (*Point-to-point*) ou le modèle par abon-

[13] L'utilisation de l'ORB reflexif DynamicTAO (voir section 2.2.2.2) avec cette implémentation pourrait néanmoins être intéressante pour la configuration de l'intergiciel.

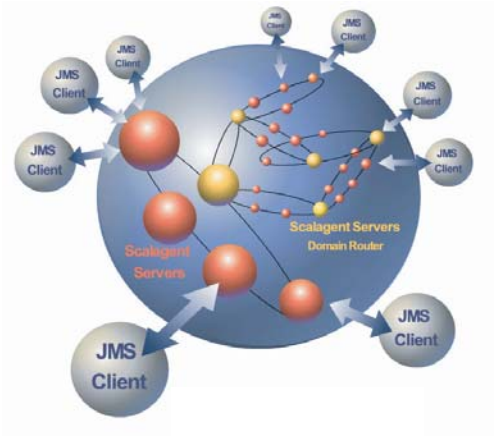

Figure 1.15 – JORAM.

nement (*Publish/Subscribe*). Rien n'empêche une application d'utiliser les deux modes à la fois, cependant JMS se focalise sur les applications qui utilisent l'un ou l'autre. JMS introduit deux termes pour définir les applications clientes de l'API JMS. Il y a les consommateurs qui représentent tous les clients qui reçoivent un message que ce soit en mode synchrone (réception en attente de message) ou asynchrone (réception sans attente). Les clients qui envoient des messages sont appelés producteurs. JORAM utilise JNDI (*Java Naming and Directory Interface*) comme système de nommage pour trouver des clients destinataires, des sujets d'abonnement, etc.

JORAM possède une architecture distribuée de ces serveurs JMS basée sur l'utilisation des serveurs A3. Cette architecture permet, par exemple, de gérer des sujets d'abonnement (*topic*) distribués. Ceci a l'avantage d'alléger la charge sur les serveurs responsables de *topic* très demandés.

Certaines fonctionnalités ne sont pas incluses dans les spécifications de JMS : la tolérance aux fautes, la notification d'erreurs, l'administration, la sécurité et un standard de persistance de messages. De plus, la notion d'événement/réaction n'est pas présente dans les spécifications. Néanmoins l'engouement de l'industrie pour JMS montre bien l'intérêt porté aux intergiciels asynchrones et surtout le besoin d'un standard commun permettant l'interopérabilité entre les différents intergiciels.

1.4 Conclusion

Un des critères que l'on peut appliquer lors d'une analyse ou d'une définition d'une architecture est le degré de liberté (ou de couplage possible) entre plusieurs applications. L'appel de procédure à distance est typiquement une technologie imposant un couplage fort entre les applications parce qu'elle impose aux clients et aux serveurs d'être actifs en même temps et qu'elle ne save pas gérer l'évolution des composants. A l'opposé, les intergiciels asynchrones favorisent le découplage car ils proposent des modes de communication très souples ; les applications ne communiquent plus directement entre elles mais s'échangent des messages par l'intermédiaire d'un médiateur. L'intergiciel asynchrone devient donc un outil indispensable pour l'intégration de système ou d'application moderne mettant en jeu des systèmes de plus en plus hétérogènes, mobiles, et à grande échelle.

Nous pensons qu'il se dégage clairement deux parties bien distinctes dans l'implémentation des intergiciels asynchrones : une partie responsable de la délivrance des messages et une partie en charge du traitement des messages. La partie responsable de la délivrance, ou *modèle de délivrance*, définit les principes fondamentaux du transfert des messages entre les différents sites (l'architecture des sites, le routage des messages, etc.). La partie en charge du traitement des messages définit la manière dont

les messages vont être manipulés avant (ou après) avoir été transférés (modèle de communication, propriétés sur les messages, etc.). Ce *modèle de traitement* des messages définit la sémantique de l'application, il spécifie le modèle de délivrance en fonction des applications. Nous nous baserons sur cette constatation pour notre proposition d'un modèle d'intergiciel asynchrone basé sur le découpage en un *modèle de délivrance* et un *modèle de traitement* des messages.

Insuffisance des solutions actuelles Les intergiciels asynchrones actuels n'abordent pas de façon suffisante les problèmes de configuration et de déploiement de l'intergiciel. Les différents projets présentés, représentatifs de l'ensemble des intergiciels asynchrones, ne permettent pas le changement de la structure des implémentations pour les adapter à des besoins spécifiques (enlever certaines propriétés coûteuses, changer l'architecture de communication des sites, etc.). Nous présentons ci-dessous une grille récapitulative des différents projets et produits[14] en fonction de leur configurabilité, de leur capacité de déploiement et de leur habilité au passage à grande échelle (scalabilité).

Projet/Produit	*Configurabilité*	*Déploiement*	*Scalabilité*
JEDI	-	-	+ +
GRYPHON	-	-	+ +
SIENA	-	-	+ +
A3	-	+	+ +
MQSeries	+	-	+
MSMQ	+	- -	+
Corba Messaging	-	-	-

Comme on peut le voir, les projets de recherche travaillent essentiellement au niveau de la scalabilité. Certain en optimisant leur algorithme de diffusion (Gryphon, SIENA), d'autres en hiérarchisant leur architecture (A3, JEDI). Et seul A3 propose un début de solution de déploiement de l'intergiciel avec un service[15] d'instanciation de serveurs à distance. Les produits industriels (MQSeries, MSMQ) essayent quant à eux de fournir quelques supports de configuration pour le traitement des messages à l'aide d'outils d'administration des files de messages. Mais globalement, peu se préoccupent des problèmes de configurabilité et de déploiement distribué de l'intergiciel.

Presque toutes les implémentations restent donc figées quels que soient les sites d'exécution et l'application (ou les applications) l'utilisant. L'ensemble des propriétés non fonctionelles offertes par l'intergiciel sont donc fournies sur chaque nœud du système. Or cette rigidité dans la composition de l'intergiciel, si elle peut être supportable dans le cadre d'une application légère de petite taille, devient carrément ingérable lors de l'utilisation à grande échelle. Nous verrons dans le chapitre4 comment la propriété de causalité pèse sur la scalabilité de l'intergiciel et les difficultés à configurer l'intergiciel pour alléger ce coût. Les intergiciels asynchrones ne proposent pas de possibilités de configuration et gardent un aspect très monolithique. Ils manquent de mécanismes permettant de contrôler aussi bien leur comportement que leur architecture.

De plus, les intergiciels asynchrones n'offrent souvent qu'une API de communication et ne propose pas la possibilité d'utiliser plusieurs sémantiques en même temps. Chaque projet supporte seulement un ou deux types de sémantique (abonnement, *Push*/*Pull*, etc.) limitant considérablement le comportement applicatif. L'utilisation de plusieurs sémantiques sur un même intergiciel, au lieu de plusieurs intergiciels avec une seule sémantique applicative, permettrait d'alléger la charge sur les sites d'exécution. Depuis l'apparition de la spécification JMS, les intergiciels asynchrones peuvent

[14] Nous n'avons volontairement pas inclus JMS car ses propriétés dépendent de l'implémentation sous-jacente (A3 pour Joram, MQSeries pour WebSphere, etc.).

[15] Cette solution a été réalisée dans le produit de Scalagent, la start-up issue du projet A3.

enfin s'appuyer sur un standard d'interopérabilité, mais de très nombreux projets n'ont pas encore d'implémentation de JMS notamment à cause de la limitation des possibilités sémantiques offertes.

La plupart des intergiciels asynchrones restent donc des « usines à gaz » monolithiques statiques qui ne sont pas configurables et donc non configurés pour satisfaire aux exigences de l'informatique omniprésente.

Tous les problèmes cités ci-dessus sont traités depuis de nombreuses années dans le cadre des intergiciels synchrones. Le chapitre suivant étudie les différentes méthodes de configuration dans les intergiciels synchrones et aborde la notion du déploiement des intergiciels.

Chapitre 2

Configuration et déploiement d'intergiciels

Le chapitre précédent a mis en avant l'intérêt des intergiciels asynchrones pour les applications distribuées à grande échelle. L'emploi d'un intergiciel responsable de la gestion des problèmes d'hétérogénéité, de distribution, etc. est certainement la solution la plus apte à répondre aux besoins croisés de passage à grande échelle, de flexibilité, d'extensibilité et d'hétérogénéité des applications mobiles de l'informatique omniprésente émergente.

2.1 Introduction

L'étude des intergiciels asynchrones du chapitre précédent a mis en avant plusieurs problèmes. Tout d'abord, leur implémentation reste figée quels que soient les sites d'exécution et l'application (ou les applications) l'utilisant. Ils n'offrent souvent qu'une API de communication qui permet un découplage fort avec l'application mais, en contrepartie, rend difficile l'adaptation de l'intergiciel aux besoins applicatifs. De plus ils sont peu voire pas configurables.

Ce chapitre se penche donc sur les travaux autour de la configuration et du déploiement des intergiciels afin de mettre à jour les techniques employées dans des domaines connexes aux intergiciels asynchrones.

La notion de configuration englobe deux activités. Premièrement, *choisir* les éléments constitutifs de l'intergiciel, c'est-à-dire définir l'architecture globale de l'intergiciel, ses interconnexions, mais aussi les propriétés non fonctionnelles des sites, les différents services, etc. Deuxièmement, *placer* cette description sur les sites distribués afin de fournir le support d'exécution aux applications. La première activité est une tâche extrêmement complexe car elle nécessite de prendre en considération les besoins des applications et les contraintes du système. Notre étude et nos travaux se concentrent sur le processus de placement et donc sur les mécanismes qui permettent la configuration de l'intergiciel.

La première section analyse les différentes techniques de configuration existant dans le domaine des intergiciels synchrones basées essentiellement sur l'utilisation de la réflexivité (Open ORB, DynamicTAO). Mais nous abordons aussi d'autres techniques comme l'AOP ou la composition de composants intergiciels (projet Aster).

La deuxième section s'intéresse à la notion de déploiement de l'intergiciel généralement peu abordée pour l'intergiciel lui-même mais plus au niveau applicatif. Le déploiement permet de mettre en place le logiciel dans un système, distribué ou non, pour son utilisation future. Mais, le rôle du déploiement ne se cantonne pas à la simple installation du logiciel ; il couvre aussi d'autres étapes

du cycle de vie du logiciel. Nous essayerons d'analyser les différentes techniques utilisées afin de proposer une solution dans le chapitre 5.

2.2 Configuration

Cette section étudie ce qui est fait dans le domaine de la configuration des intergiciels pour connaître les forces et les faiblesses de chacune des approches. La quasi totalité de ces travaux sont réalisés sur les intergiciels synchrones. Nous essayerons de mettre en avant les avantages de chacune des méthodes afin de choisir la plus adaptée à nos besoins pour les intergiciels asynchrones.

Les méthodes de configuration des intergiciels se distinguent par la configurabilité des intergiciels eux-mêmes mais aussi par les mécanismes mis en jeu permettant la configuration et la reconfiguration à l'exécution. Nous étudions l'utilisation de la réflexivité, une technique de plus en plus utilisée pour la configuration des intergiciels synchrones. Puis, nous présentons son utilisation dans les modèles à composants et dans les ORB. Nous décrivons ensuite la configuration par assemblage de composants intergiciels. La section se clôt, enfin, par une étude des apports de la programmation par aspects pour la configuration des intergiciels.

Nous verrons ensuite dans les chapitres suivants comment tirer parti des travaux menés dans ces domaines pour l'appliquer aux intergiciels asynchrones.

2.2.1 La réflexivité

Cette section présente brièvement le concept de réflexivité en général. Ce concept peut s'appliquer de différentes façons, sur des systèmes très divers, mais il serait trop long, et hors de propos ici, de présenter tous ces domaines d'application. Cette section se contente donc d'en présenter quelques uns seulement, à savoir ceux qui sont utilisés par les travaux de recherche présentés dans la section suivante.

2.2.1.1 Généralités

On admet généralement que le concept de réflexivité a été considéré pour la première fois dans toute sa généralité par Brian C. Smith en 1982 dans [84]. On peut en donner la définition suivante de P. Maes [61] :

> *Un système informatique est dit réflexif lorsqu'il fait lui-même partie de son propre domaine. Plus précisément cela implique que (i) le système a une représentation interne de lui-même, et (ii) le système alterne parfois entre calcul « normal » à propos de son domaine externe et calcul « réflexif » à propos de lui-même.*

L'auteur affirme que la réflexivité permet d'inspecter les comportements internes d'un système ou d'un langage et, en dévoilant son implémentation, de modifier certains comportements pour les enrichir (ajout d'instructions de surveillance, de sécurité, de qualité de service, etc.) ou les adapter (remplacement d'un protocole de communication, ajout de la gestion de la distribution, etc.). Elle va un pas plus loin que Smith en montrant les avantages que procure une vision orientée objet dans la construction d'un système réflexif. Elle développe 3KRS, le premier langage réflexif à objets.

En 1991, Kiczales, Des Rivieres et Bobrow développent CLOS, une architecture réflexive orientée objet. Ils introduisent le concept de protocoles à métaobjets pour désigner l'interface de communication entre les parties fonctionnelles et non fonctionnelles d'une application. Ils soulignent l'importance de l'utilisation d'un langage orienté objet dans la conception d'un système réflexif [51] :

> *Reflective techniques make it possible to open up a langage's implementation without revealing unnecessary implementation details or compromising portability, and object-oriented techniques allow the resulting model of the language's implementation and behaviour to be locally and incrementally adjusted.*

Plus récemment, certains travaux se sont intéressés aux architectures distribuées réflexives, un thème assez peu abordé précédemment. Citons entre autres les travaux de McAffer qui a développé Tj [65], une architecture distribuée réflexive dans laquelle l'accent est mis sur les techniques de passage d'arguments et de valeurs (le « marshalling »), et ceux de Blair qui lui s'intéresse aux « liaisons ouvertes » [23].

2.2.1.2 Définitions

Le principe de réflexivité, peut se résumer en quatre définitions : les deux premières définissent les différents niveaux et les deux suivantes les opérations principales permettant la communication entre un niveau de base et son niveau méta :

- Le **niveau de base** : le niveau de base est un niveau « applicatif » dans lequel évoluent les objets fonctionnels de l'application (objets de base) ;
- Le **méta-niveau** : le méta-niveau est un niveau où les objets systèmes, de contrôle et d'implémentation (méta-objet) évoluent. C'est la partie qui « raisonne » sur les opérations et les structures du niveau de base ;
- La **réification** consiste à construire une représentation formelle d'une structure ou d'une opération du niveau de base, implicite à ce niveau, de façon à ce qu'elle devienne manipulable par le niveau méta ;
- la **réflexion**[1] consiste à répercuter sur le niveau de base les modifications effectuées par le niveau méta sur la représentation formelle du niveau de base, précédemment réifiée.

Comme on peut le voir sur la figure 2.1, le méta-niveau peut lui même être considéré comme un niveau de base et posséder son auto-représentation sous forme de méta-niveau. On appelle ce niveau le « méta-méta-niveau ».

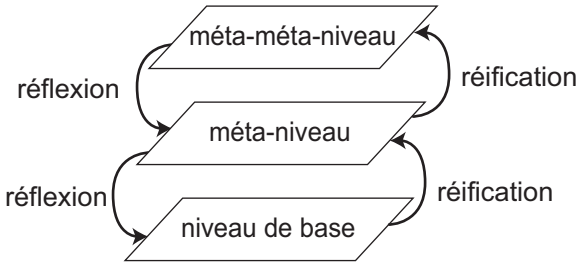

Figure 2.1 – Les principes de la réflexivité.

Le niveau méta manipule en général une représentation formelle des opérations et des structures du niveau de base. Lorsqu'en fait il ne manipule qu'une représentation des opérations, on parle de réflexivité comportementale. A l'inverse, lorsqu'il ne manipule qu'une représentation des structures, on parle de réflexivité structurale.

Afin de simplifier la complexité des architectures à méta-niveaux et fournir une gestion compréhensible des interfaces réflexives, le méta-espace d'un objet est très souvent défini par une

[1] On trouve parfois l'utilisation du mot « intercession » à la place du mot « réflexion ». Dans le contexte de la réflexivité, ces mots signifient la même opération.

architecture de réflexion multi-modèles. Le méta-espace est partitionné en plusieurs méta-espaces distincts (chacun suivant un modèle spécifique) offrant différentes vues de la plate-forme et pouvant être indépendamment réifiés. C'est le cas par exemple du modèle à composants Fractal décrit dans la section suivante.

2.2.1.3 FCF : The Fractal Composition Framework

FCF[2] (*Fractal Composition Framework*) est à la fois une série de concepts définissant un modèle à composants réflexifs et un ensemble d'interfaces de programmation ([27]). Un des objectifs principaux de Fractal est de fournir un support pour la construction et la configuration dynamique de systèmes. Les composants Fractal fournissent donc des (méta)informations qui permettent au programmeur d'accéder (opération appelée *introspection*) et de manipuler (opération appelée *intercession*) dynamiquement à leur structure et à leur comportement.

Fractal se veut suffisamment général pour pouvoir être utilisé de différentes manières :

- comme support à la définition et la configuration ; Fractal est une base pour les différents langages et outils tel que les outils de configuration graphique, les langages de description d'architecture (ADL) et leur compilateur et outils associés, ... ;
- comme modèle général de composition ; Fractal permet de nombreux types de composition tels que la composition structurelle, opérationnelle, comportementale, ... ;
- comme support d'administration ; Fractal peut être étendu aux modèles de composants administrables et utilisé comme une base pour la création d'outils d'administration (supervision, diagnostiques, etc.).

Le cœur de FCF définit les APIs minimales permettant l'élaboration de telles entités. Ces APIs sont basées sur le modèle à composant concret de Fractal présenté dans la suite.

Le modèle à composant Fractal

La spécification du modèle à composant Fractal est motivé par sept exigences essentielles à la réalisation d'un modèle à composant général. C'est l'association de ces exigences qui dicte le modèle à composant général Fractal.

(1) Encapsulation, abstraction, identité La première (très proche des besoins exprimés dans [47]) comprend trois propriétés : l'encapsulation, l'abstraction et l'identité. L'encapsulation signifie qu'un composant doit interagir avec son environnement, uniquement à l'aide de points d'accès et d'opérations prédéfinies. L'abstraction signifie qu'un composant ne doit pas révéler plus que nécessaire les détails de son implémentation aux entités de son environnement. Et enfin l'identité signifie qu'un composant doit avoir une identité permettant de le désigner de façon non ambiguë par rapport aux autres composants.

(2) Composition La deuxième exigence est la possibilité de composition et d'assemblage des composants permettant de créer de nouveaux composants de plus haut niveau. Le modèle ne doit néanmoins pas imposer une sémantique de composition particulière et doit pouvoir laisser la possibilité de gérer aussi bien une composition structurelle qu'opérationnelle par exemple. De plus, la composition doit être explicitement maintenue et modifiable à l'exécution.

(3) Partage L'exigence d'une certaine forme de partage de composants est dictée par les motivations principales de Fractal que sont la description et la construction de configurations de systèmes

[2] Pour plus de clarté, *Fractal Composition Framework* sera dénommé par Fractal dans le reste du document.

(c'est-à-dire l'assemblage de composants) prenant en charge les différentes situations d'infrastructure (logicielle ou matérielle) impliquant le partage et le multiplexage. C'est le cas par exemple dans la gestion de ressources qui requiert la possibilité d'exprimer et de manipuler les dépendances entre les différentes ressources (par exemple, le partage d'un composant processeur entre différents processus, ou le multiplexage de connexions réseaux permettant la gestion d'un protocole réseau de plus haut niveau, ...).

(4) Cycle de vie et déploiement La quatrième exigence est la gestion du cycle de vie et des différents aspects du déploiement des composants (incluant les différentes formes d'installation, d'instanciation, d'initialisation, d'activation, etc.). Cette exigence découle directement du besoin d'automatisation du processus de gestion de systèmes à grande échelle mis en avant dans β4] et [43].

(5) Manipulation d'activités La cinquième exigence est la nécessité de rendre explicite et de supporter la manipulation d'activités mis en œuvre dans un système (processus, thread, transactions, ...) et pouvant impliquer plusieurs composants. Cette exigence se rapporte à la deuxième et la troisième en ce sens qu'une activité peut être perçue comme une composition d'autres activités (i.e. composants) elles-mêmes partagées par d'autres.

(6) Contrôle de comportement La sixième exigence est un raffinement de la deuxième. Elle concerne la possibilité de créer différentes formes de composants de contrôles, c'est-à-dire des composants qui fournissent des capacités d'introspection pour observer un ensemble de (sous)composants et exercer un certain contrôle sur leur exécution (par exemple, interception de messages ou d'invocations afin d'ajouter des pré et post traitements à l'exécution normale).

(7) (Re)Configuration et mobilité La septième exigence est la capacité de reconfiguration et de mobilité des composants (et des assemblages de composants), c'est-à-dire que le contenu et les dépendances de ressources des composants doivent pouvoir évoluer dans le temps (de façon spontanée ou en réaction à diverses interactions de composants). Cette évolution doit pouvoir se faire de façon dynamique à l'exécution.

A ces sept exigences guidant la définition du modèle à composant général de Fractal sont associées les trois propriétés nécessaires à une architecture de composition ouverte : la programmabilité (Les composants doivent offrir des interfaces permettant leur instanciation et leur manipulation dynamiquement), l'adaptabilité (nécessité d'une séparation entre interface et implémentation permettant une adaptation de l'implémentation aux contraintes du système) et l'extensibilité (permettre d'étendre les interfaces initiales en introduisant de nouvelles ou dérivant d'existantes).

Modèle à composants général

Les spécifications de Fractal sont basées sur deux modèles à composants. Le premier, présenté dans cette section est un modèle à composant général de haut niveau. Il sert de base au modèle à composant concret présenté dans la section suivante.

Le modèle général, appelé aussi *kell model* peut être vu comme une généralisation du modèle de référence RM-ODP [47]. Il est basé sur quatre principaux concepts : les noms, les interfaces, les signaux et les *kells* . Nous n'entrerons pas en détail dans ces notions de *kell*-calcul développées dans [85].

On appelle un *kell* un composant de haut niveau, une structure d'exécution conforme au modèle général. Un *kell* est formé de deux parties : une membrane et un plasme. Le plasme d'un *kell* est composé d'un nombre (fini) d'autres *kells*. Comme le montre intuitivement la figure 2.2, un kell peut être vu comme un composant composite avec un certain nombre de sous-composants. Le modèle *kell* est donc un modèle récursif permettant une imbrication arbitraire des *kells*. De plus, un *kell* peut apparaître dans le plasme de différents kells, c'est-à-dire être partagé par plusieurs *kells* différents, c'est alors à la membrane englobante de dicter le contrôle du *kell* entre les plasmes.

La membrane (ou le contrôleur) d'un *kell* renferme le comportement de contrôle spécifique du *kell*. Elle peut intercepter les signaux de ou vers les *kells* du plasme, fournir une représentation causalement[3] liée des *kells* du plasme et superposer un comportement de contrôle au comportement initial des *kells* du plasme. De plus une membrane peut être vue comme un contexte de nom minimal.

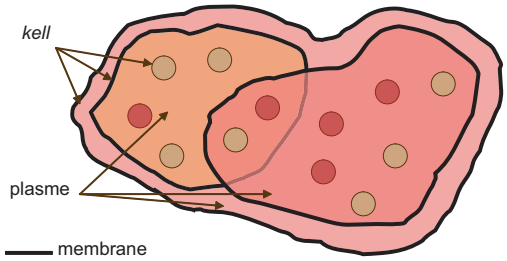

Figure 2.2 – *kells*, plasmes et membranes.

Les *kells* interagissent avec leur environnement grâce à des échanges de signaux par des points d'accès identifiés appelés interfaces. Un signal porte le nom de l'interface visée et des arguments pouvant prendre trois formes : un nom (un nom est un symbole désignant l'interface d'un *kell* dans un certain contexte), une valeur ou un *kell*. Le comportement d'un *kell* est donc défini par un ensemble de transitions spécifiées par des signaux.

Modèle à composants concret

Généralités Le modèle à composants concret introduit des constructions et des restrictions spécifiques au modèle général visant à faciliter son implémentation (notamment en langage de programmation Java). Dans ce modèle, la notion de *kell* est remplacée par celle de composant et les noms par des références sur les interfaces. Comme un *kell*, un composant est une structure d'exécution formée de deux parties : un contrôleur (spécialisation de la notion de membrane) et un contenu (spécialisation de la notion de plasme). Comme on peut le voir sur la figure 2.3, le contenu d'un composant est composé d'un nombre fini d'autres composants (sous-composants) qui sont sous le contrôle du contrôleur englobant. Le modèle est récursif et permet à des sous-composants d'être eux-mêmes composés de sous-composants. La récursivité s'arrête avec les composants primitifs qui encapsulent des objets Java.

Le modèle concret reprend la notion de partage de composant du modèle général. Différents composants peuvent recouvrir un même composant, c'est-à-dire qu'un composant peut être partagé par (être sous-composant de) un certain nombre d'autres composants englobants (voir figure 2.4). Un composant partagé par deux ou plusieurs composants est sujet au contrôle respectif de leurs contrôleurs. Mais la sémantique exacte de la configuration résultante est en générale déterminée par le composant englobant tous les composants de cette configuration de partage.

[3] Toute modification sur un *kell* est répercutée sur son plasme et vice et versa.

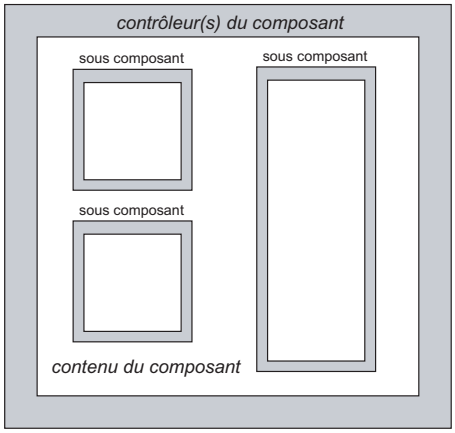

Figure 2.3 – Un exemple de composant du modèle concret (avec trois sous-composants).

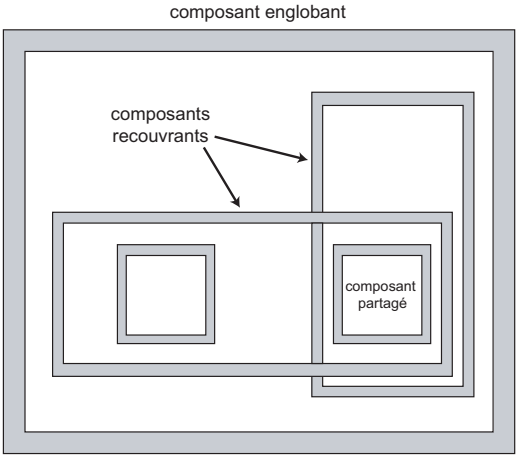

Figure 2.4 – Composant partagé.

Un composant peut interagir avec son environnement à travers des opérations d'invocation sur des interfaces identifiées. Les interfaces correspondent (conformément au modèle *kell* et RM-ODP) à des points d'accès à l'exécution et non pas à des interfaces au sens Java qui sont considérées comme des « types d'interfaces » dans le modèle concret (un type d'interface pouvant être supporté par plusieurs composants alors qu'une interface n'appartient qu'à un et un seul composant). La référence d'une interface (*interface reference*) contient le nom de cette interface, une référence vers l'identité du composant qui la possède et sa position interne ou externe au composant. La visibilité (externe) des interfaces d'un sous-composant est déterminée par le contrôleur du composant englobant. De plus, le contrôleur d'un composant englobant peut avoir des interfaces internes non visibles à l'extérieur mais visible par les sous-composants (voir figure 2.5).

Les interfaces d'un composant peuvent être clientes (émettre une invocation d'opération et recevoir ou non des arguments de retour) ou serveurs (recevoir une invocation d'opération et retourner ou non des arguments). Une liaison (*binding*) est un lien entre une interface cliente et une interface serveur. Elle ne peut être établie que si le type de l'interface serveur est un sous-type de l'interface cliente.

Par analogie avec la notion de membrane du modèle *kell*, le contrôleur d'un composant peut intercepter les invocations, superposer un comportement de contrôle au comportement initial du com-

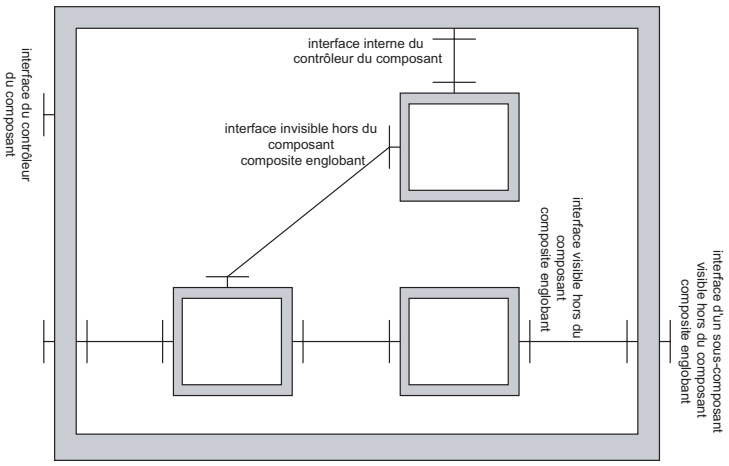

Figure 2.5 – Visibilité des interfaces de composants.

posant et fournir une représentation causale du composant (Il n'y a aucune limitation des capacités de contrôle du contrôleur).

Dans le modèle concret, les opérations des composants sont des interactions basiques entre les interfaces. Une opération peut être une invocation sans retour (correspondant à une émission de signal dans le modèle *kell*) ou avec retour d'arguments (correspondant à une émission et réception de signal dans le modèle *kell*), ceux-ci pouvant être des noms (référence d'interface), des valeurs ou des composants passivés .

Le nommage des composants du modèle concret est défini par une interface spécifique (obligatoirement présente sur chaque composant) désignant de façon non ambiguë l'identité du composant : *ComponentIdentity*. Cette interface permet d'accéder au type du composant et à toutes ses interfaces externes. L'identité d'un composant peut être considérée comme un contexte de nom et une référence d'interface comme un nom de ce contexte de nom.

Système de type Alors que le modèle général donne une vision purement « système d'exécution » dans laquelle les *kell* sont dynamiquement créé par d'autres *kell*, le modèle concret introduit un système de type et des mécanismes, étendus de Java, permettant aux utilisateurs d'instancier les composants à partir de types de composant et de vérifier la validité des assemblages de composants.

Un type d'interface (*InterfaceType*) est constitué d'un identifiant, d'une signature d'interface et de trois booléens indiquant son rôle, sa contingence et son comportement de liaison. Un identifiant est un nom valide dans le contexte du composant possédant l'interface de ce type. La signature d'interface est une collection de signatures de méthodes (comme les interfaces Java standards). Le type d'interface peut avoir un rôle client (interface requise, émettrice d'invocations) ou serveur (interface fournie, réceptrice d'invocations). La contingence d'une interface peut être *optionnelle*, c'est-à-dire non forcément fournie (s'il s'agit d'une interface serveur) ou non forcément liée (s'il s'agit d'une interface client). Mais une interface peut aussi être *obligatoire*, c'est-à-dire qu'elle doit fournir une implémentation (s'il s'agit d'une interface serveur) ou qu'elle doit être forcément liée à l'exécution (s'il s'agit d'une interface client). Le comportement de liaison d'une interface permet au composant de fournir plusieurs interfaces du même type, le comportement est soit un singleton, dans ce cas la liaison sur une interface nommé I retournera une interface nommée I, soit une collection, dans ce cas la liaison sur une interface I retournera une interface de nom I*suffixe*.

Un type de composant est défini par la collection des types d'interface que le composant possède.

Un type d'interface et un type de composant sont invariables et ne peuvent pas changer dynamiquement à l'exécution.

Instanciation Les composants sont instanciés à partir de patrons (*templates*) eux-mêmes créés par des usines à patrons (*template factories*). L'instanciation peut prendre deux formes :

- La création d'un nouveau composant basé sur son type. Elle permet par la suite d'ajouter et de lier ce composant à d'autre composants instanciés (présents à l'exécution).
- L'introduction de patrons de composant à l'intérieur d'autres patrons. Elle permet l'instanciation directe d'une structuration de patron sans ajout ou liaisons postérieures.

Pour simplifier, on peut dire que les usines à patrons sont responsables de « qu'est-ce qui » est instancié alors que les patrons sont responsables de « quand » l'instancier.

2.2.1.4 Julia, une implémentation de FCF

Dans la section précédente, nous avons introduit les motivations et les concepts du modèle à composant Fractal. Nous allons maintenant aborder l'architecture d'une implémentation de Fractal : Julia. Cette implémentation (disponible sur le site web d'objectweb www.objectweb.org) suit le modèle concret de Fractal et ses API. Nous ne rentrerons pas dans le détail du code de Julia et des spécifications de toutes les interfaces qui sont décrites dans la documentation Java car ce n'est pas le but de ce document. Nous nous contenterons d'expliquer simplement les aspects de flexibilité de configuration qui nous serviront par la suite.

Structure des composants Tout composant Julia suit les spécifications du modèle concret de Fractal. Un composant possède donc au moins l'interface de contrôle *ComponentIdentity* qui fournit l'identifiant du composant. Si le composant est un composite il possède en plus l'interface *ContentController* qui permet de manipuler l'ensemble de ses sous-composants. Toute interface additionnelle est facultative. La membrane d'un composant Julia peut contenir deux types d'entités : des objets de contrôle et des objets d'interception. Les objets de contrôle sont accessibles grâce à une référence d'interface, par exemple, la référence d'interface de contrôle *ComponentIdentity* d'un composant permet d'accéder à l'objet de contrôle de l'identité du composant. Cette référence (*ComponentIdentity*) nous permet par la suite d'obtenir les références vers toutes les interfaces externes (clientes et serveurs) du composant qu'elle soit de contrôle ou non. En effet, une référence d'interface de contrôle est une référence d'interface au même titre qu'une référence d'interface fonctionnelle du composant (c'est-à-dire une référence vers le code interne du composant) (voir la figure2.6).

Un objet d'interception (ou intercepteur) est une sorte d'objet de contrôle qui intercepte des appels de méthode. Il permet d'ajouter des pré- et post-traitements avant et après l'invocation sur l'objet délégué. Les objets d'interception sont très utiles pour réaliser de l'adaptation (ajout de traitements spécifiques au support d'exécution, traitements spécifiques à l'appelant, ...) ou de l'observation (statistiques d'invocations du composant, temps d'exécution de la méthode, ...) de composants. Un objet d'interception est associé à un générateur de code ([28]) qui permet de transformer le code natif Java (*bytecode*) d'une classe et optimise l'insertion du code d'interception dans le code du composant. Les objets d'interception peuvent aussi invoquer indistinctement les objets de contrôle. Ceci permet par exemple de bloquer un appel de méthode si le composant n'est pas actif.

De la même manière, les objets de contrôle peuvent s'appeler entre eux permettant de gérer l'interdépendance des différents traitements de contrôle. Par exemple, le contenu d'un composant composite ne peut pas être modifié si le composant n'est pas dans un état passif. En d'autres termes, lorsque la méthode d'ajout de composant est appelée sur l'interface *ContentController*, l'objet de contrôle

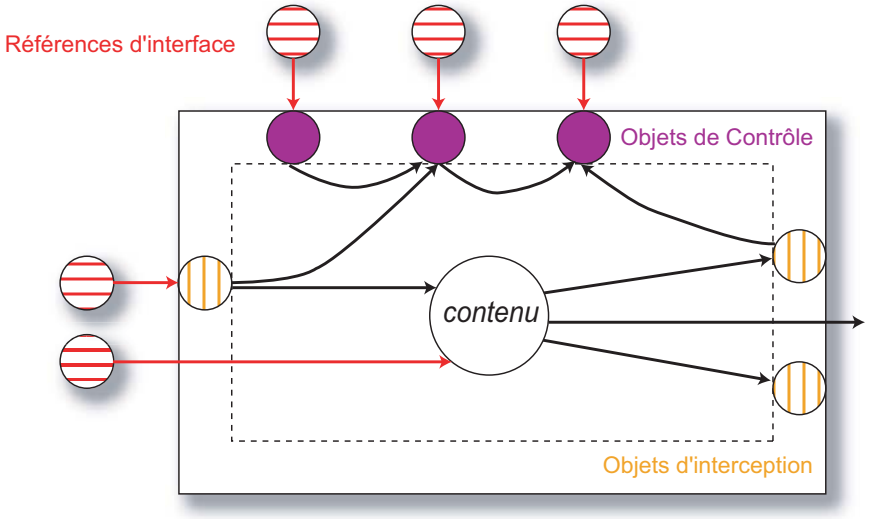

Figure 2.6 – Structure d'un composant Julia.

vérifie auprès de l'objet de contrôle de cycle de vie (*LifeCycleController*) que le composant est bien dans l'état *arrêté*.

Il est important de noter que Julia fournit un certain nombre d'implémentations de contrôleurs. Ainsi il est possible de se servir du contrôleur de contenu présent dans le paquetage (*BasicContentController*) ; celui-ci fournit une implémentation de la gestion du contenu d'un composant composite. Il existe d'autres implémentations comme celle du contrôleur de cycle de vie (*GenericLifeCycleController*) ou le contrôleur de liaison d'un composant (*PrimitiveBindingController*). Bien entendu, il est tout à fait possible de créer ses propres contrôleurs, c'est même le but de Fractal (et donc de Julia) de fournir uniquement l'architecture de développement, libre ensuite de créer ses propres implémentations tant qu'elles suivent le modèle.

Julia.cfg Julia fournit un puissant (mais austère !) fichier de configuration de composants. Ce fichier[4] définit les ensembles de types d'interface de contrôle que doivent supporter les composants, les classes Java à utiliser pour décrire les interfaces, l'ensemble des objets de contrôle implémentant ces interfaces, un ensemble d'intercepteurs et le niveau d'optimisation du code. Ce fichier est utilisé à l'initialisation de Julia pour décrire les types de composants et leurs contrôleurs associés. Une description des contrôleurs à instancier est donc ajoutée lors de la création des patrons de composant. Ceci permet de factoriser la définition des contrôleurs (et de leur instance) de chaque composants.

Le fichier de configuration inclut également un niveau d'optimisation du code. La section optimisation permet de décrire les générateurs de classes permettant de fusionner certaines classes en une seule afin d'optimiser les performances à l'exécution. Il est possible de fusionner les contrôleurs entre eux, les intercepteurs avec les contrôleurs, les intercepteurs et le contenu, ou encore les intercepteurs avec les contrôleurs avec le contenu (plus de détails peuvent être trouvés dans le tutorial de Julia). Ces optimisations permettent de limiter le nombre d'objets Java instanciés et donc d'améliorer les performances du système à l'exécution.

[4] Le lecteur pourra se reporter à la documentation se trouvant sur le site d'ObjectWeb pour plus de précisions.

Langage de Description d'Architecture (ADL) Comme nous l'avons abordé plus haut, la création et l'instanciation des composants se fait par la définition de types de composant ; puis par la création d'usines à patrons permettant de créer des patrons grâce auxquels on peut appeler la création puis l'instanciation de composants ! Cette méthode devient vite complexe lorsqu'on désire créer une composition avec un nombre important de composants. Plutôt que d'utiliser directement l'API Fractal, la méthode la plus simple consiste donc à utiliser l'ADL fourni dans Julia. La version actuelle de Julia définit un ADL assez basique décrit en XML et fournit un outil pour analyser la configuration décrite et créer automatiquement le patron de la configuration. L'outil permet de récupérer l'identité du composant racine de la configuration sur lequel on appelle la méthode d'instanciation qui crée toute la configuration.

Nous proposons dans la section 5.2.4 une version étendue personnalisée de l'ADL de Julia permettant une configuration plus poussée de l'intergiciel grâce à l'ajout de fonctionnalités de gestion de cycle de vie, d'initialisation, etc. De plus, nous tentons de combler l'aspect distribué, absent de l'ADL de base, en proposant une version plus globale de l'ADL.

2.2.1.5 Synthèse sur la réflexivité

Les sections précédentes ont présenté de manière détaillée les principes de réflexivité à travers le modèle Fractal et son implémentation Julia.

La réflexivité en général et l'architecture de composition Fractal plus spécialement sont bien adaptées aux besoins de configurabilité, de configuration statique et dynamique car elles prennent en compte d'une part l'aspect composition (qui permet la construction d'un intergiciel par un ensemble structuré de composants) et d'autre part l'aspect dynamique (qui facilite l'adaptation de l'intergiciel aux changements du système et de l'application). L'aspect composition permettant de construire un intergiciel comme une interconnexion d'autres composants offre un haut degré de configurabilité. De plus, les propriétés de réflexivité fournies par l'architecture Fractal (en particulier, l'accès à des meta-informations permettant de manipuler dynamiquement la structure et le comportement des composants) facilitent la reconfiguration dynamique des composants.

Cette technique nous semble donc très prometteuse. Nous présentons dans la section suivante la manière dont la réflexivité a été appliquée dans les intergiciels synchrones. Les mécanismes fournis permettent de rendre les intergiciels configurables statiquement et dynamiquement.

2.2.2 Les ORB configurables

Cette section présente deux exemples d'ORB configurables, OpenORB et dynamicTao. Ces deux ORB illustrent deux approches de développement de système reflexif et plus précisément d'intergiciel synchrone réflexif. Le développement de DynamicTAO commença avec TAO, une implémentation modulaire de l'ORB CORBA mais complètement statique. Les développeurs ont repris le code de TAO pour y ajouter des caractéristiques réflexives afin de rendre le système plus flexible, dynamique et configurable. Réciproquement, le développement d'OpenORB est parti de zéro en se concentrant sur la mise en place d'une nouvelle architecture d'intergiciel où chaque élément respecte le principe de réflexivité.

Ces deux exemples donnent un bon aperçu des travaux réalisés autour des intergiciels synchrones et plus précisément des ORB afin de répondre au besoin de configuration dynamique à l'aide de la réflexivité.

2.2.2.1 OpenORB

OpenORB [22] a été conçu pour répondre au manque de flexibilité des plates-formes industrielles comme CORBA, DCOM, etc. Les concepteurs d'OpenORB se sont fixés deux objectifs majeurs :

- **configurabilité :** pour que l'intergiciel réponde aux exigences d'une (ou plusieurs) application(s).
- **dynamicité :** pour permettre à l'intergiciel de répondre aux changements des besoins applicatifs et aux changements de l'environnement (bande passante du réseau, pannes, etc.).

Pour atteindre ces objectifs, les concepteurs d'OpenORB utilisent le concept de réflexivité présentée dans la section 2.2.1.

L'architecture d'OpenORB OpenORB est conçu comme un assemblage de composants. Le modèle de composants utilisé a été conçu pour mettre en œuvre les concepts de réflexivité présentés dans le paragraphe précédent.

Chaque composant de l'intergiciel possède un méta-espace causalement[5] lié au composant. Ce méta-espace est composé de quatre méta-objets illustrés sur la figure 2.7.

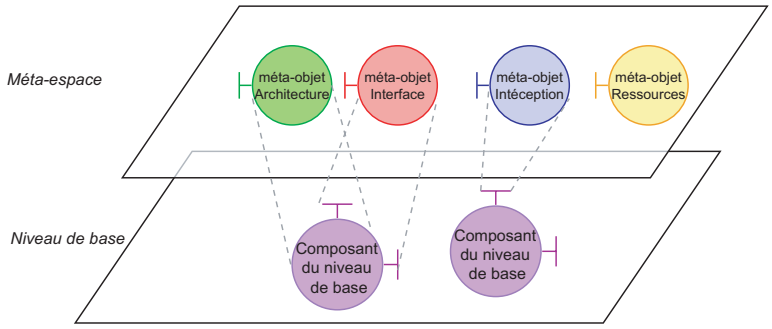

Figure 2.7 – Structure d'un méta-espace dans OpenORB.

1. **Le méta-objets d'encapsulation :** il représente les interfaces du composant, c'est-à-dire l'ensemble des méthodes fournies par le composant.
2. **Le méta-objets de composition :** il représente l'architecture du composant. Dans le modèle d'OpenORB, un composant peut-être composé par un assemblage d'autres composants. Ce méta-niveau sert à décrire de tels composants : un composant est représenté comme un graphe de composants interconnectés.
3. **Le méta-objets environnemental :** il réifie les activités relatives aux interfaces du composant. Ce méta-niveau permet, par exemple, d'insérer des pré- ou post- traitements aux appels de méthodes effectués par et sur le composant.
4. **Le méta-objets de ressources :** il réifie les ressources utilisées par le composant : processus, mémoire, etc.

Mécanismes de configuration OpenORB offre d'intéressantes capacités de configuration et reconfiguration. Sa structure réflexive à composants et la quantité d'informations réifiées permet de changer dynamiquement les fonctionnalités offertes par l'intergiciel et permet de contrôler ses consommations de ressources. L'architecture basée sur l'association d'un méta-espace à chaque objet offre une granularité très fine pour l'emploi de la réflexion.

[5] Toute modification du niveau de base est répercutée sur le méta-niveau et vice versa.

2.2.2.2 Dynamic TAO

DynamicTAO [55] est une extension de l'ORB TAO [81] permettant la reconfiguration du moteur interne de l'ORB et des applications l'utilisant. Dans DynamicTAO, les *ComponentConfigurator* (*CC*) représentent les relations de dépendance entre les composants de l'ORB et entre l'ORB et les composants applicatifs. Chaque (*CC*) stocke un graphe orienté de dépendance des composants de l'ORB et des applications (voir figure 2.8).

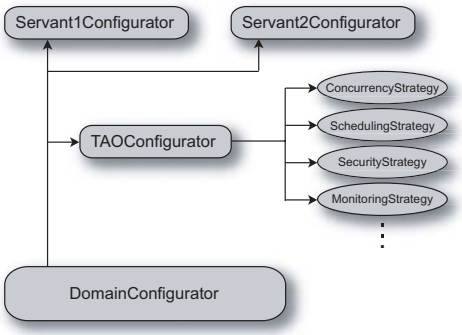

Figure 2.8 – Les composants de configuration de DynamicTAO.

Lorsqu'une requête de remplacement d'un composant C arrive, l'intergiciel examine les dépendances entre C et les autres composants (de l'ORB et applicatifs) en utilisant le (*CC*) de C. Les développeurs de l'intergiciel peuvent donc utiliser cette caractéristique pour écrire du code qui garantit la cohérence de l'ORB en cas de reconfiguration dynamique.

DynamicTAO exporte des méta-interfaces pour charger et décharger les modules dans le système d'exécution et pour inspecter et changer la configration de l'ORB. Les méta-interfaces sont disponibles aux développeurs dans l'optique de tests et de mises au point pour réaliser de la maintenance de serveurs, mais aussi à tous les composants logiciels qui veulent inspecter ou reconfigurer la structure de l'ORB.

Un des principaux buts de DynamicTao est d'avoir un noyau ORB minimal s'exécutant continuellement même pendant la mise à jour dynamique de nouvelles stratégies de l'ORB ou des servants. Et grâce à la vérification et le maintien de la cohérence de l'ORB, il permet de garantir toute reconfiguration dynamique.

2.2.2.3 Synthèse sur les ORB configurables

L'utilisation des techniques de réflexivité dans les intergiciels synchrones a été réalisée avec succès. Elle a permis de les rendre facilement configurables statiquement mais aussi à l'exécution. Les mécanismes fournis permettent de modifier la structure interne de l'intergiciel en changeant tel ou tel composant. De ces projets ressort une caractéristique intéressante : la minimalité de l'intergiciel qui consiste à fournir un « noyau » minimal qui permet de configurer l'intergiciel à sa convenance (en fonction de ses besoins et contraintes) tout en fournissant une continuité d'exécution de l'intergiciel ou partie de l'intergiciel (pendant la reconfiguration d'un seul « bout » de l'intergiciel). Nous essayerons de reprendre cette idée pour l'appliquer à notre modèle d'intergiciel asynchrone.

La section suivante présente une autre technique de configuration : la programmation orientée aspect. C'est un outil puissant pour la prise en charge de propriétés non fonctionnelles dans un intergiciel permettant de séparer sa fonctionnalité de ses propriétés.

2.2.3 Utilisation de l'AOP pour la configuration des intergiciels

La programmation par aspects (AOP[6]) [53] a récemment été introduite comme solution aux problèmes de gestion des traitements non fonctionnels dans les intergiciels.

2.2.3.1 La notion d'aspect

Dans l'AOP, les propriétés non-fonctionnelles sont désignées sous l'appellation d'aspect. L'approche propose leur modularisation au moyen d'une programmation indépendante et séparée. Son objectif principal est d'isoler les traitements dispersés mais identiques et ainsi de faciliter la manipulation d'un aspect donné. La construction d'une application est ainsi basée sur l'entrelacement (*weaving*) du code des modules indépendants qui implémentent les aspects.

2.2.3.2 Principe

Pour développer une application en utilisant l'AOP, un programmeur dispose de plusieurs outils :

Un langage de composants pour décrire un programme à base de composants qui implémentent les fonctionnalités du système.

Un langage d'aspects pour implémenter les aspects désirés.

Un « tisserand » ou *weaver* qui admet en entrée le code des aspects et le code des composants et fournit en sortie un programme combinant les deux. Pour rendre cette combinaison possible, il faut que les programmes à base de composants et les programmes implémentant les aspects aient des points de rencontre (*join points*) définis par le développeur de l'application.

2.2.3.3 Projets

Dans [26], la programmation par aspects est utilisée pour séparer les propriétés fonctionnelles d'un intergiciel de son code métier. Les auteurs ont développé leur intergiciel en utilisant AspectJ [52], un langage de programmation par aspects. Cela permet de coder les aspects fonctionnels de l'intergiciel dans un langage de programmation classique et les propriétés non fonctionnelles séparément sous forme d'aspects. Le langage AspectJ joue ensuite le rôle du tisserand : il insère les aspects définissant les propriétés non fonctionnelles dans le code fonctionnel de l'intergiciel. On obtient ainsi un intergiciel configuré fournissant un certain nombre de propriétés non fonctionnelles spécifiques.

2.2.3.4 Synthèse sur l'AOP

La programmation par aspects est un outil puissant pour la prise en charge de propriétés non fonctionnelles dans un intergiciel. Cette technique permet de séparer clairement la fonctionnalité de l'intergiciel de ses propriétés non-fonctionnelles. Parmi les nombreux exemples d'aspects, on peut citer le traitement des erreurs, la persistance,... On comprend à la vue de ces exemples que programmer un aspect nécessite de faire des modifications sur l'ensemble du programme, ce qui diminue sa lisibilité et augmente, de ce fait, sa complexité. De plus, l'implémentation des aspects est très dépendante de l'implémentation de l'intergiciel. Les aspects codés sont donc difficilement réutilisables. D'autre part, les personnalisations faites à l'intergiciel sont statiques (tout ajout, suppression ou modification d'un aspect nécessite une recompilation de l'intergiciel). Il n'est donc pas envisagé de pouvoir reconfigurer dynamiquement l'intergiciel comme dans le cas des approches réflexives.

[6] *Aspect Oriented Programming*.

La section suivante est un peu différente des précédentes car elle ne présente pas vraiment des mécanismes de configuration mais plutôt la manière dont on peut configurer un intergiciel. Il nous semblait important de présenter des travaux sur la *manière* de configurer afin de fournir les meilleurs *moyens* dans notre proposition.

2.2.4 Composition de *composants intergiciels* : le projet Aster

Aster [49, 54] vise à faciliter le développement des applications distribuées ayant des exigences de qualité de service différentes. Les concepteurs d'Aster partent du principe qu'un système distribué doit être capable de supporter l'exécution d'un grand nombre de classes d'applications (des applications multimédias aux applications massivement parallèles). Pour atteindre cet objectif, l'intergiciel qui supporte l'exécution d'une application doit pouvoir être configuré de manière à répondre aux exigences de celle-ci.

Aster permet au programmeur de composer ses applications à l'aide de composants logiciels déjà existants. Il donne, en outre, la possibilité de spécifier en logique du premier ordre les propriétés de qualité de service nécessaires à chaque composant et se propose de composer un intergiciel par assemblage de composants intergiciels (*middleware components*) pour fournir ces propriétés de qualité de service.

2.2.4.1 La théorie sous-jacente

Un des enjeux d'Aster a été de déterminer un langage formel de spécification des fonctionnalités et des possibilités d'interactions des composants intergiciels. Une configuration à haut niveau a été choisie : une méthode formelle a été définie pour permettre de raisonner sur l'adéquation entre des besoins applicatifs et des fonctions des composants intergiciels. Précisons les éléments constitutifs de l'environnement Aster et les méthodes formelles utilisées.

Les éléments constitutifs d'Aster

- **Langage d'interconnexion** Le langage d'interconnexion sert à décrire l'application distribuée qui va s'exécuter sur l'intergiciel. Ces applications sont représentées par un assemblage de composants logiciels. Le langage d'interconnexion sert à décrire ces composants logiciels et leurs interconnexions. Pour cela, il permet de décrire :
  1. les interfaces des composants logiciels ;
  2. une application en termes d'interconnexions d'interfaces.
- **Bus abstrait** L'exécution des applications Aster repose sur l'utilisation d'un intergiciel synchrone ou bus logiciel. Cet intergiciel est obtenu en ajoutant un certain nombre de composants intergiciels à un intergiciel de base présent sur les différents sites. L'ensemble des composants intergiciels adjoints forme le *bus abstrait*.
  Ces concepts sont représentés sur la figure 2.9 : le Bus Abstrait spécialise un ORB présent sur les différents sites. L'intergiciel formé par ces deux couches héberge l'exécution d'une application distribuée représentée par un ensemble de composants logiciels.

Spécification des propriétés d'exécution Les propriétés d'exécution fournies par les composants intergiciels ou requises par les composants applicatifs sont exprimées en logique du premier ordre. Elles dénotent les interactions entre composants distants du point de vue de la qualité de service mise en œuvre. Elles sont donc définies en termes de prédicats de base caractérisant les actions de communication sous-jacentes (i.e. *send* et *receive*) et l'occurrence d'une défaillance (i.e. *failure*).

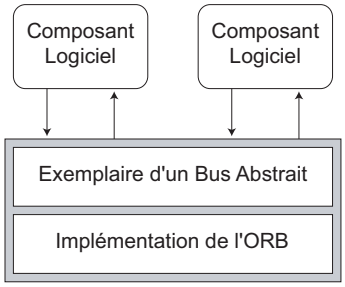

Figure 2.9 – Structure d'une application Aster

Exemple : Le bus abstrait met en œuvre deux sémantiques de défaillance : *au plus une fois* (ou *At-Most-Once*) et *meilleur effort* (ou *Best-Effort*). Soit C_1 et C_2 deux composants applicatifs et req une requête émise par C_1 à destination de C_2, les propriétés correspondantes sont formellement définies comme suit :

$$Best\text{-}Effort(C_1, C_2, req) \equiv$$
$$send(C_1, C_2, req) \wedge (\neg failure(C_1, C_2, req) \Rightarrow receive^{+}(C_1, C_2, req))$$

$$At\text{-}Most\text{-}Once(C_1, C_2, req) \equiv$$
$$send(C_1, C_2, req) \wedge$$
$$((\neg failure(C_1, C_2, req) \Rightarrow receive(C_1, C_2, req)) \wedge$$
$$(failure(C_1, C_2, req) \Rightarrow (\neg receive^{+}(C_1, C_2, req) \vee receive(C_1, C_2, req))))$$

Spécialisation par appariement de spécifications formelles Etant donné la spécification d'exigences et, dualement, de comportements implémentés par les composants logiciels, on peut construire un système spécialisé par appariement de spécifications formelles. On peut définir deux types d'appariement, suivant que les deux spécifications doivent être équivalentes ou non. Soit $\mathcal{P}_1$ et $\mathcal{P}_2$, deux ensembles de comportements où un comportement est défini par la conjonction de propriétés d'exécution, on définit :

– **Appariement exact**

$$\mathcal{P}_1 \lhd_{exact} \mathcal{P}_2 \equiv \forall P_2^i \in \mathcal{P}_2 : \exists P_1^j \in \mathcal{P}_1 \mid P_1^j = P_2^i$$

– **Appariement spécialisé (ou *plug-in*)**

$$\mathcal{P}_1 \lhd_{plug\text{-}in} \mathcal{P}_2 \equiv \forall P_2^i \in \mathcal{P}_2 : \exists P_1^j \in \mathcal{P}_1 \mid P_1^j \Rightarrow P_2^i$$

A partir de cette définition de l'appariement de spécifications, on peut sélectionner un bus logiciel correspondant aux exigences de l'application. Soit $\mathcal{R}_A$, l'ensemble des exigences d'une application, et $\mathcal{P}_A$, l'ensemble des comportements implémentés par les composants de A. Un bus logiciel B implémantant l'ensemble de comportements $\mathcal{P}_B$ et exigeant $\mathcal{R}_B$ peut être choisi pour supporter l'exécution de A si et seulement si :

$$(\mathcal{P}_B \cup \mathcal{P}_A) \lhd_{plug\text{-}in} (\mathcal{R}_B \cup \mathcal{R}_A) \tag{2.1}$$

2.2.4.2 La construction d'une application

Une application répartie se construit avec l'environnement Aster en cinq phases [50] :

1. **spécification :** description de l'application en terme d'interconnexion de composants logiciels et spécification des besoins en qualité de service de chaque composant à l'aide du langage formel Aster,
2. **compilation :** vérification statique des types et génération de descripteurs de composants,
3. **construction de système distribué :** identification des composants intergiciels nécessaires à la satisfaction des besoins en qualité de service des différents composants de l'application,
4. **génération :** production du code d'interface entre la brique logicielle créée et l'intergiciel existant et production d'un *makefile*,
5. **production :** compilation standard par l'intermédiaire du *makefile*.

Comme représenté sur la figure[7] 2.10, ces différentes phases sont implémentées au moyen de trois outils :

- **le sélecteur** qui recherche les composants intergiciels nécessaires à la satisfaction des besoins de l'application,
- **le générateur** qui produit une application exécutable en réalisant l'interconnexion des composants de l'application avec les composants intergiciels choisis par le sélecteur,
- **le compilateur** qui est un compilateur standard.

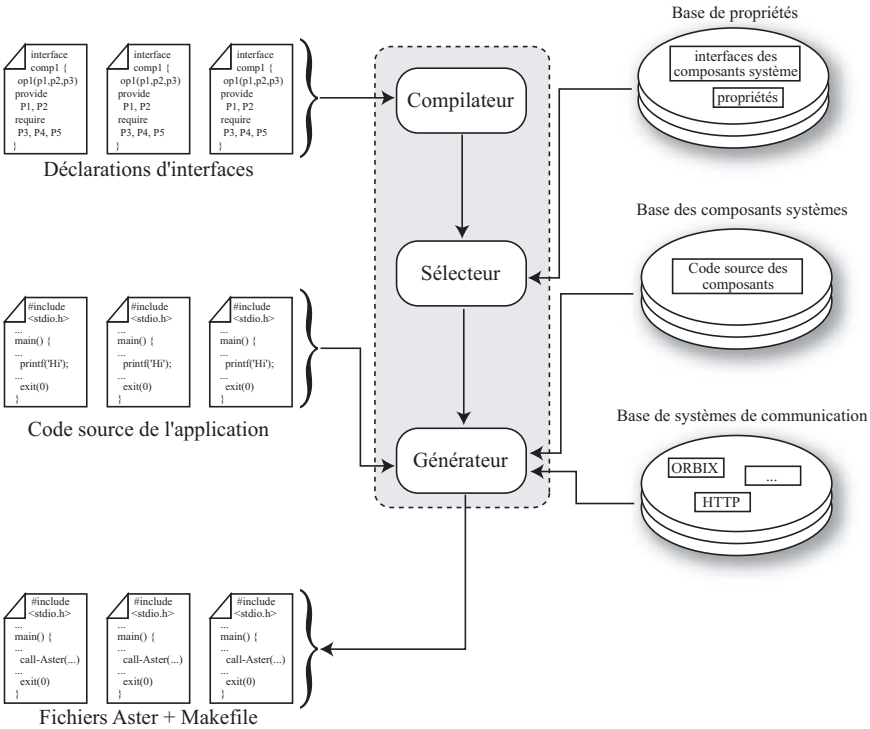

Figure 2.10 – Architecture de l'environnement Aster

[7] Cette image est une reproduction d'une image parue dans [48].

2.2.4.3 Exemple d'utilisation

Dans [20], Aster est utilisé pour construire un système distribué de gestion de fichiers à partir de plusieurs systèmes centralisés de gestion de fichiers. Pour ce faire, l'application a besoin d'un composant système appelé *localisateur* qui permet de sélectionner un des systèmes centralisés de gestion de fichiers. Les besoins de l'application sont formalisés ainsi que les propriétés offertes par le composant système responsable de la localisation. Le système vérifie ensuite que la relation2.1 est respectée. En adjoignant ce composant système au bus abstrait, on obtient alors le bus logiciel désiré.

2.2.4.4 Synthèse sur la composition de composants intergiciels

Le projet Aster introduit un concept intéressant qui est la configuration d'un intergiciel par composition de composants intergiciels. La spécification formelle en logique du premier ordre des exigences des composants d'une application et des composants intergiciels disponibles permet de déterminer l'ensemble des composants intergiciels nécessaires à la satisfaction des besoins d'une application.

Cependant, comme les concepteurs le soulignent dans [48], l'environnement Aster ne garantit la correction sémantique de l'intergiciel généré qu'à la condition que les spécifications des composants intergiciels soient correctes. Or le formalisme employé est assez complexe et donc difficile d'utilisation pour le développeur d'applications réparties. Il conviendrait donc de construire des outils permettant de vérifier la correction de l'implémentation des composants intergiciels par rapport à leur spécification.

En outre, on peut reprocher à Aster de ne pas considérer l'aspect *performance de l'intergiciel* dans le processus de personnalisation. Aster procède indépendamment pour chaque composant de l'application. Dès lors qu'un composant requiert une propriété non fonctionnelle, il est adjoint un ou plusieurs composants intergiciels au bus de communication pour satisfaire cette propriété. Or il peut être utile d'étudier une propriété dans son ensemble pour optimiser sa gestion. C'est notamment le cas de la prise en charge de propriétés complexes comme l'ordonnancement des messages.

De façon générale, on peut reprocher à Aster sa méthodologie axée sur l'intégration d'intergiciel complet pour satisfaire quelques propriétés. L'utilisation d'un intergiciel configurable à base de composants associé à la méthode de configuration éviterait ainsi les pertes de performances liées à l'agglomération d'intergiciels monolithiques.

2.2.5 Bilan sur la configuration

Le chapitre précédent a mis en avant l'aspect monolithique des implémentations actuelles d'intergiciels asynchrones. Dans cette section, nous nous sommes attardés à analyser les travaux autour de la configuration des intergiciels en général. Il est ressorti plusieurs points intéressants. Tout d'abord que la plupart des travaux actuels sont concentrés sur la configuration des intergiciels synchrones et plus spécialement les ORB. Peu de travaux se sont intéressés à configurer les intergiciels asynchrones et donc encore moins à les rendre configurables.

Les travaux de configuration des intergiciels synchrones ont mis en avant trois points essentiels qu'il nous faudra exploiter :

- Offrir la possibilité d'être exécuté de façon minimale (un intergiciel offrant un comportement fonctionnel de base permettant la reconfiguration dynamique), comme c'est le cas de dynamicTAO.
- Pouvoir accueillir (statiquement ou dynamiquement lors de l'exécution) de nouveaux comportements fonctionnels ou responsables de la gestion de propriétés non fonctionnelles. Nous pensons que cela peut être facilité par l'utilisation des techniques de réflexivité (comme dans OpenORB).

– La configuration doit se faire dans le respect des besoins applicatifs et des contraintes du système comme le montre les travaux du projet ASTER.

L'AOP est une technique très intéressante mais qui ne nous a pas convaincus, notamment à cause de ses difficultés à réaliser des reconfigurations dynamiques. Par contre l'approche d'OpenORB basée sur des méta-niveaux horizontaux combinée avec l'utilisation de modèles à composants réflexifs comme FCF nous semble tout à fait apte à répondre au besoin de configurabilité et de configuration des intergiciels asynchrones.

Trouver la bonne technique de configuration est essentiel mais la configuration n'est qu'une étape dans le processus plus complet qu'est le déploiement. La section suivante aborde les différentes notions du déploiement d'intergiciel.

2.3 Déploiement

Il est communément admis que le déploiement est une étape importante dans l'ingénierie logicielle. En effet, le déploiement permet de mettre en place le logiciel dans un système, distribué ou non, pour son utilisation future. Mais le rôle du déploiement ne se cantonne pas à la simple installation du logiciel ; il couvre aussi d'autres étapes du cycle de vie du logiciel.

Le déploiement est une activité complexe, encore mal maîtrisée. En effet, il s'avère particulièrement difficile de déployer des applications réparties à grande échelle. Cette complexité a plusieurs causes :

– l'hétérogénéité des environnements considérés : les applications réparties à grande échelle s'exécutent sur des machines aux caractéristiques et aux ressources très différentes.
– le nombre croissant d'entités à déployer : avec l'explosion de l'Internet et la croissance du nombre d'appareils performants pouvant s'y relier (assistant personnel, téléphone mobile, etc...), les applications réparties mettent en jeu un nombre important d'entités.

De nombreux travaux ont été faits sur le déploiement des applications, mais la plupart du temps, le processus utilise un intergiciel comme support de déploiement pour installer, configurer et démarrer les composants applicatifs. En revanche, peu de travaux existent sur le déploiement des intergiciels eux-mêmes. Or, si la première partie de ce chapitre a clairement mis en avant les besoins en terme de configuration des intergiciels, elle n'explique pas le problème essentiel de la mise en place de ces configurations complexes sur des architectures à grande échelle. Les nouvelles contraintes de mobilité, d'hétérogénéité et de scalabilité auxquelles doivent faire face les intergiciels rendent leur processus de déploiement d'autant plus délicat et nécessitent d'autant plus son automatisation.

Cette section met en avant les travaux sur le déploiement, principalement réalisés sur les applications (et les services) et essaye d'en tirer les différents besoins en termes de mécanismes.

2.3.1 Déploiement dans les services

Ces dernières années, un certain nombre de plates-formes de services sont apparues. C'est notamment le cas des spécifications industrielles comme OSGI. *Open Services Gateway Initiative* (OSGi) [71, 31] est une alliance indépendante qui a pour but de définir et de promouvoir des spécifications ouvertes pour la livraison de services administrables dans des réseaux résidentiels, véhiculaires et autres types d'environnements. Dans OSGI la sémantique et le comportement d'un service sont définis par son interface. La plate-forme OSGI se divise en deux parties : une architecture d'administration de services d'une part et la spécification des services standards d'autre part. La grande force d'OSGI

réside dans son modèle de programmation à base de services et sa gestion du déploiement de services. Les services sont assemblés en paquetages (appelés *bundle*), qui représentent l'implémentation des services et leurs ressources ainsi qu'un concept logique utilisé par l'architecture pour organiser son état interne (dépendance avec d'autres services, état de déploiement,...). Néanmoins, OSGI n'offre pas une vision distribuée de la plate-forme. De plus, il n'y a pas de moyen de décrire une architecture globale comme une interconnexion de composants. Ceci limite donc grandement la gestion de services distribués.

Un nouveau paradigme de service a fait son apparition depuis peu, il s'agit des *Web Services*. Les Web Services se résument en (1) une façon d'enregistrer et de retrouver des services [1], (2) un mécanisme de transport pour accéder à un service [25] et (3) un langage de description des paramètres d'entrées/sorties sur un service [33]. Néanmoins, les Web Services ne sont pas une plate-forme de service mais juste une évolution du RPC classique appliquée aux nouvelles technologies du Web. Ils ne répondent pas aux problématiques d'interconnexion, d'architecture globale et de déploiement de services distribués.

2.3.2 Déploiement applicatif

Le processus de déploiement fait référence à toutes les activités et leurs interactions lors du cycle de vie d'une application. Ces activités concernent le versionnement (multiples configurations), l'installation, la désinstallation, la mise à jour, l'adaptation, l'activation et la désactivation d'une application. Dans notre cas, nous nous intéressons plus particulièrement aux activités de versionnement et d'installation, dans le cas d'applications basées sur des modèles à composants. Les langages de description d'architectures (voir le chapitre 3) ont tiré profit de ces structures à composants pour faciliter le déploiement.

Dans la suite nous présentons brièvement un exemple de processus de déploiement applicatif à base de composants. Cet exemple donne un bon aperçu des techniques utilisées pour le déploiement des applications à grande échelle. Ensuite, nous prendrons l'exemple du déploiement d'un modèle à composants : CCM.

2.3.2.1 Un exemple : déploiement sur la plate-forme Scalagent

La plate-forme Scalagent fournit un ensemble d'outils de construction, de configuration et de déploiement d'applications. Tous ces outils sont basés sur l'utilisation de l'ADL OLAN (voir section 3.6) et utilisent comme support d'exécution l'intergiciel asynchrone A3 (voir section 1.3.1.4). Dans le modèle Scalagent, chaque composant fonctionnel (appelé SCBean) possède une coque de contrôle appelé SCContainer qui fournit des interfaces de contrôle pour la configuration et le déploiement (interfaces de liaison, de gestion du cycle de vie et de configuration d'attributs). Le modèle permet la hiérarchisation d'une configuration d'application sous forme de composites appelés SCController qui possèdent un référentiel de leur architecture interne. La figure 2.11 résume ces notions (une description plus complète du modèle peut être trouvée dans [63]).

Le service de déploiement peut activer une application alors que tous ses composants ne sont pas encore installés. Dans ce cas, seules certaines parties de l'application fonctionnent. Le reste de l'application est intégré au fur et à mesure de l'accomplissement des installations restantes.

Le SCController offre une interface de contrôle spécifique utilisée par les outils pour déployer et (re)configurer le SCController. Elle permet d'ouvrir une session de déploiement ou de (re)configuration. Dans les deux cas, cette ouverture de session prend en paramètre un fichier ADL dans lequel est indiqué la nouvelle configuration interne ($\approx$ la suite d'opérations à effectuer sur le SCController) : ajout et/ou suppression d'un SCContainer ou d'un SCController, interconnexion entre

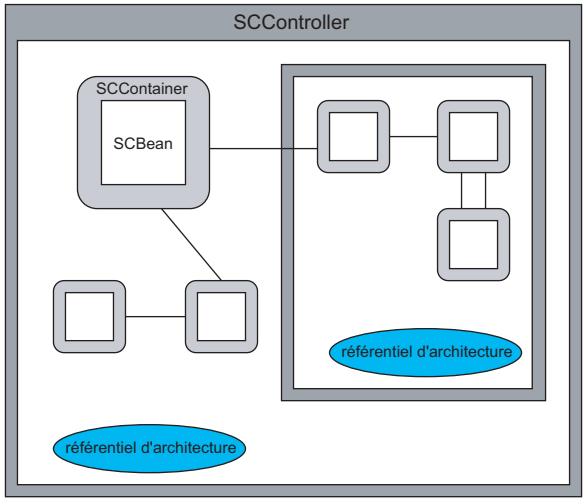

Figure 2.11 – Exemple d'architecture dans le modèle Scalagent.

entités encapsulées, lien entre le SCController et un composant interne, positionnement d'attributs, etc.

Processus de déploiement Le déploiement d'une application fait un usage intensif de la description ADL. Un service présent dans la plate-forme exploite cette description pour créer les différents SCControllers correspondant aux composites de l'ADL. Ce service offre une API à travers laquelle il est possible de créer des sessions de déploiement prenant en paramètre le fichier ADL. Ce dernier est analysé pour créer le SCController de niveau le plus haut (c'est-à-dire l'application) qui va lui-même créer les SCContainers encapsulés, provoquer le déploiement des SCControllers encapsulés, effectuer les liaisons et configurer les propriétés des entités encapsulées. Ce processus est exécuté récursivement par l'ensemble des SCControllers.

L'aspect hiérarchique du modèle de composants présente un double intérêt :

– le processus de déploiement peut être parallélisé en découpant la tâche globale en un ensemble de tâches de niveau inférieur effectuées par les SCControllers.
– il est possible de définir l'ordre d'installation, d'activation et de liaison des composants. Ceci permet, entre autre, de pouvoir activer des parties de l'application avant même que tous les composants ne soient déployés.

Le processus de déploiement est décrit sous la forme d'un modèle de tâches de déploiement qui s'enchaînent selon un workflow. Les tâches sont asynchrones et prennent en compte les erreurs (avec remonté de la progression du déploiement et des erreurs pour chaque session). En cas d'erreur (ou de panne), l'initiateur du déploiement (un outil, un administrateur, etc.) a la possibilité d'effectuer des tâches compensatrices pour résoudre les problèmes. Soit tout est défait, soit il tente une reconfiguration d'une partie de l'application (changement de sites d'exécution, nouvelles liaisons, ajout/suppression de composants, etc.). Plus de détails peuvent être trouvés dans [75].

2.3.2.2 Déploiement dans CCM (*Corba Component Model*)

Le modèle de déploiement de CCM s'appuie sur l'utilisation de paquetages de composants déployables et composables. Dans le cadre des composants CORBA, un paquetage est une entité regroupant un ensemble d'implémentations de composants ainsi qu'un (ou plusieurs) descripteur(s)

de composants. Il existe deux principaux types de paquetages de déploiement décrits à l'aide des descripteurs suivants :

- le descripteur de paquetage de composants. Il décrit la structure d'un composant : héritage, interfaces supportées, ports, etc. Il permet de déployer un composant « seul » ;
- le descripteur d'assemblage de composants. Il permet de déployer de manière simple des composants dépendants les uns des autres.

Ces différents descripteurs sont basés sur le langage OSD. CCM propose un scénario de déploiement, basé sur ces packages et leurs descripteurs, assurant l'installation cohérente des applications CCM. Dans ce modèle, le déploiement est réalisé par un ORB et par des outils de déploiement. Ces outils utilisent les services de l'ORB pour réaliser le transfert, l'installation, l'assemblage, l'instantiation et la configuration des applications sur les machines des clients.

2.3.3 Déploiement des intergiciels

Cette section reprend deux intergiciels étudiés précédemment (Aster et OpenORB) qui, comme la plupart des intergiciels, ne possèdent pas de processus de déploiement à proprement parler, mais qui fournissent des méthodes pour installer l'intergiciel qui sont les prémices d'un outil de déploiement.

2.3.3.1 Déploiement dans le système Aster

Le système Aster ne fournit pas de mécanismes de déploiement à proprement parler. En effet, aucun outil n'est proposé pour mettre en place l'intergiciel sur les différents sites. De plus, Aster est un système statique : les traitements effectués pour déterminer l'ensemble des composants intergiciels nécessaires sont faits avant le lancement de l'application et il n'est pas possible de modifier la structure de l'intergiciel en cours d'exécution. Il n'existe donc aucun mécanisme d'adaptation, ni de mise à jour de l'intergiciel.

Néanmoins, on peut considérer que le système Aster fournit les prémices d'un outil de déploiement car il permet de déterminer l'ensemble des composants intergiciels nécessaires sur les différents sites. Les phases de spécification des comportements désirés et de sélection des composants intergiciels nécessaires peuvent être considérées comme les étapes préliminaires du déploiement.

2.3.3.2 Déploiement dans OpenORB

OpenORB ne fournit pas de support pour le déploiement. Les différents composants formant l'ORB doivent être déployés indépendamment sur les différents sites de l'application. De plus, contrairement à Aster, OpenORB ne fournit pas d'aide à la configuration de l'intergiciel. C'est au développeur de l'application de déterminer l'assemblage de composants nécessaires à la satisfaction des besoins de l'application.

Comme il est souligné dans [24], il est regrettable qu'il n'existe pas un processus de déploiement qui pourrait mettre en place l'intergiciel, opérer les reconfigurations nécessaires, tout en préservant l'intégrité de l'intergiciel. En effet, une des contraintes imposées par le dynamisme d'un intergiciel est qu'il est nécessaire de garantir que les reconfigurations effectuées sur une partie de celui-ci ne vont pas nuire au reste du système. C'est le rôle d'un processus de déploiement de garantir le respect de cette intégrité.

2.3.3.3 Déploiement dans Scalagent

Un début de solution est proposé par Scalagent. Un processus veilleur est présent sur tous les sites qui peuvent potentiellement recevoir un serveur d'agent. Ce processus a la charge de lancer

les processus de l'intergiciel avec la bonne configuration définie statiquement (ports d'écoute TCP, protocoles réseaux, etc.). Il reçoit une description ADL des serveurs d'agents à lancer, les instancie et active les processus. Mais il n'existe pas de composition possible de l'intergiciel à l'aide de briques élémentaires, il s'agit uniquement de configurer des fonctions préexistantes dans le *jar* (complet) de l'intergiciel.

2.3.4 Bilan sur le déploiement

De notre étude, il apparaît que les problèmes de déploiement sont de plus en plus pris en compte par des outils et par des spécifications de modèles à composants. Les outils essayent de résoudre et d'automatiser plus particulièrement les problèmes d'installation et de maintenance, alors que les spécifications essayent d'aider ces outils en résolvant en amont une partie du travail (descriptions, ...). En spécifiant explicitement les contraintes à respecter pour le déploiement, ces spécifications améliorent les activités de déploiement (en terme de dépendance, de localisation physique des composants, ...). Ainsi certains problèmes de déploiement sont pris en compte dès la phase de développement de l'application et non lors de l'installation elle-même. Un autre avantage est l'effort de standardisation des descriptions.

Un aspect intéressant de la plate-forme Scalagent réside dans l'utilisation d'un modèle de composants hiérarchiques basé sur l'ADL OLAN. Celui-ci permet de structurer les applications construites, ce qui présente un double intérêt : d'une part, cela permet de diviser et de répartir la connaissance de l'architecture selon un principe de localité : les référentiels d'architecture conservent, à chaque niveau, uniquement les informations requises par ce niveau. D'autre part, cela permet de distribuer les processus globaux, comme le déploiement, qui font usage de l'architecture de l'application.

2.4 Conclusion

Ce chapitre avait pour but d'étudier les différents travaux autour de la configuration et du déploiement des intergiciels afin de n'en retirer que la « substantifique mœlle » pour les intergiciels asynchrones. Nous avons étudié dans un premier temps les principes de réflexivité qui proposent des mécanismes de contrôle multiples par le biais de méta-niveaux. Puis, nous avons constaté que la mise en place de ces techniques dans les intergiciels synchrones donne des résultats intéressants en terme de configurabilité et de contrôlabilité des intergiciels.

Presque toujours concentrés sur les intergiciels synchrones, ces travaux ont fait ressortir de nombreux points qu'il nous faut prendre en considération :

- Exécution de l'intergiciel dans un « noyau » minimal permettant la reconfiguration dynamique sans bloquer le déroulement des applications (ou le moins possible).
- Configurabilité de l'intergiciel. C'est-à-dire, posséder une architecture suffisamment souple pour pouvoir accueillir (statiquement ou dynamiquement lors de l'exécution) de nouveaux comportements fonctionnels ou responsables de la gestion de propriétés non fonctionnelles.
- Spécification de configuration sous une forme descriptive en fonction des besoins applicatifs et des contraintes du système.
- Nécessité de mécanismes de contrôle facilitant la configuration et le déploiement de l'intergiciel.

L'étude menée au début de ce chapitre nous montre que l'approche basée sur l'utilisation de modèle à composant réflexif comme FCF est tout à fait apte à répondre à ces besoins.

Malgré la puissance des mécanismes, peu de projets expliquent leur méthode de configuration. Ces mécanismes, aussi puissants soient-ils, nécessitent une représentation explicite des dépendances entre composants et des *besoins* des composants. Il nous semble pourtant essentiel d'accompagner

la configuration d'une description d'interfaces et d'une description des propriétés structurelles et comportementales de l'intergiciel. L'utilisation de langage de description d'architecture permet de décrire de façon hiérarchique l'ensemble d'une configuration.

Le chapitre suivant étudie les langages de description permettant de décrire formellement la structure et les propriétés qui caractérisent le comportement d'un logiciel complexe.

Chapitre 3

Description de configuration : utilisation des ADL

Le chapitre précédent a mis en avant les mécanismes de configuration et de déploiement utilisés dans les intergiciels synchrones. Mais ces mécanismes, aussi puissants soient-ils, nécessitent la maintenance d'une représentation explicite des dépendances entre composants et besoins des composants. C'est pourquoi la configuration d'un intergiciel est généralement accompagnée d'une description d'interfaces et d'une description de ses propriétés structurelles et comportementales, laquelle est souvent donnée au moyen d'expressions d'un langage naturel. La configuration d'intergiciels à composants requiert toutefois une description précise et non ambiguë des propriétés de la structure et du comportement.

Dans la suite, nous allons étudier des langages permettant de décrire formellement la structure et les propriétés qui caractérisent le comportement d'un logiciel complexe.

3.1 Introduction

Les langages de description n'ont pas été conçus initialement dans le cadre de la configuration d'intergiciel mais dans celui de la programmation constructive [38]. Cependant, ils permettent d'obtenir des modèles d'application tout à fait adéquats pour la modélisation d'applications réparties dont nous comptons nous servir pour la configuration d'intergiciel.

La programmation constructive, dénommée aussi programmation par composition de logiciels, se focalise sur la notion d'architecture des applications. Elle permet de séparer la mise en œuvre des applications en deux niveaux distincts : le premier niveau correspond à la mise en œuvre des briques logiciels de bases de l'application par les programmeurs. Le second niveau correspond à l'assemblage et à l'intégration de ces briques de bases pour former l'architecture de l'application. Ce type de macro programmation permet de modéliser de manière compréhensible et synthétique l'architecture d'une application et permet de séparer clairement la programmation des composants de leurs liaisons. Ces langages de description vont nous permettre de répondre aux questions suivantes : Comment modéliser l'architecture d'un intergiciel ? Et comment configurer puis déployer un intergiciel ?.

3.2 Qu'est-ce qu'un langage de description ?

La description d'une architecture logicielle repose sur l'emploi d'un ADL (*Architecture Description Language*) qui comprend un ensemble de notations pour définir les composants, connecteurs et configurations.

Un composant peut être soit primitif, soit complexe, un composant complexe correspondant à une configuration de composants. Un composant primitif se rapporte à un programme, mis en œuvre au moyen d'un langage de programmation quelconque. Sa définition au moyen d'un ADL comprend la déclaration de son interface, c'est-à-dire la liste des opérations fournissant un service pour d'autres composants ou requérant un service d'autres composants, et la déclaration du fichier d'implémentation associé.

Un connecteur peut également être soit primitif, soit complexe, il est alors décrit en termes de composants et de connecteurs existants. Toutefois, la définition de connecteurs complexes n'est pas toujours intégrée dans les ADL existants. Un connecteur primitif correspond à un protocole de communication offert par la plate-forme d'exécution cible. Par exemple, dans un environnement Unix, on trouvera au moins un connecteur correspondant au *pipe*. La définition d'un connecteur primitif au moyen d'un ADL comprend la déclaration de son interface et l'indication de la mise en œuvre du protocole correspondant.

Enfin, la définition d'une configuration consiste à interconnecter un ensemble de composants de manière à lier les opérations requises par les composants aux opérations offertes par d'autres composants. Ces interconnexions sont en outre réalisées par l'intermédiaire de connecteurs qui fixent les protocoles de communication entre les différents composants.

3.2.1 Caractéristiques communes des ADL

Il existe divers ADL, chacun ayant des capacités différentes. Cependant, tous les ADL ont un certain nombre de concepts en commun [42] :

Les composants représentent l'unité de base du système. On trouve parmi eux des clients, des serveurs, des bases de données, des objets, des filtres, ...

Les connecteurs sont le moyen de connexion des composants. Ils gèrent la communication et la synchronisation entre les composants. Parmi les connecteurs, on trouve des protocoles client-serveur, des envois de messages, des appels de procédures locaux ou à distance, ...

Le système décrit est un graphe dont les nœuds sont les composants et les arcs les connecteurs.

Les propriétés représentent les informations sémantiques sur le système, les composants et les connecteurs.

Les contraintes imposent des propriétés que doit respecter le système, soit un de ses composants ou soit simplement un connecteur : par exemple, le nombre de clients admissibles d'un composant serveur.

Les ADL existants diffèrent par l'exploitation qui est faite de la description de l'application. En effet, la recherche autour des ADL s'est concentrée sur deux points :

- **la génération d'un exécutable et son déploiement.** C'est le cas d'ADL comme UniCon [82], Olan [14] ou C2 [66] qui utilisent la description de l'application pour automatiser son processus de déploiement.
- **l'analyse du système.** C'est le cas d'ADL tels que Rapide [60] ou Wright [5] qui permettent au développeur de l'application de spécifier le comportement dynamique des différentes entités du système. Ces ADL utilisent la description de l'application pour modéliser et analyser les scénarios envisageables.

3.3 UniCon

UniCon [82] est un langage de description d'architectures dont un des buts est de construire des systèmes à partir d'éléments architecturaux déjà existants. Il est organisé autour de deux concepts

symétriques : les *composants* et les *connecteurs*. Un système est constitué de différents composants interconnectés par des connecteurs.

3.3.1 Les composants

Les composants sont spécifiés par une interface. Un composant peut être primitif ou composite. Un composant primitif correspond aux unités de compilation des langages de programmation classiques et aux autres objets que l'on trouve dans la couche utilisateur : fichiers, processus, etc. Un composant composite est défini par un ensemble de composants reliés par des connecteurs.

Une interface de composant comprend :

– le type du composant qui correspond à la nature de sa mise en oeuvre (c'est-à-dire bibliothèque C, processus Unix, etc.),
– des *players* qui sont les points d'interaction du composant avec les connecteurs.

3.3.2 Les connecteurs

Les connecteurs définis dans UniCon sont aussi bien définis explicitement (pipe, RPC) qu'implicitement : par exemple, des interactions indirectes entre composants dues à l'accès à une ressource partagée.

Les connecteurs sont définis par leur type, par des propriétés qui spécialisent le type et par des *rôles* qui sont les points d'interaction du connecteur avec les composants.

3.3.3 L'assemblage

L'assemblage entre les composants se fait au moyen des connecteurs. Cela consiste en un appariement de rôles et de players.

3.3.4 Les outils

– *Le compilateur* : une fois construit, le système est analysé par le compilateur UniCon. Celui-ci vérifie que les players sont connectés aux rôles duaux et que les composants composites répondent à leur spécification. Le compilateur génère ensuite le code qui permet de relier les players des composants aux rôles des connecteurs.
– *L'outil graphique* : UniCon fournit un outil graphique qui facilite la construction de l'application. Le développeur peut même ajouter des spécifications pour documenter son architecture. Celles-ci ne seront pas analysées par le compilateur.

3.3.5 Limites d'UniCon

Une limite majeure d'UniCon est que le déploiement d'une application répartie en termes de processus et de sites doit être fait explicitement par le programmeur. Imaginons qu'un composant définissant un module (bibliothèque logicielle) soit réutilisé dans une autre application pour en faire un composant serveur accessible à distance. Le concepteur de l'application doit redéfinir le composant UniCon correspondant en le typant différemment, en redéfinissant son interface et en modifiant son architecture initiale afin de changer les connecteurs et toutes les interconnexions attenantes. Cette opération n'est pas immédiate, le changement de l'architecture du système distribué sous-jacent et de l'application même impose un travail important de « réingéniérie » (restructuration).

3.4 Rapide

3.4.1 Présentation

Rapide [60] est à la fois un langage de description d'architectures et un ensemble de langages de spécification dont le but est de décrire et de simuler des systèmes informatiques, tant au niveau matériel que logiciel. Les architectures Rapide ne peuvent pas être transformées en code exécutable. Un système est modélisé comme un ensemble de composants interconnectés qui peuvent générer des événements ou y réagir. La simulation d'une architecture Rapide génère une collection d'événements partiellement ordonnés appelée *poset*. Un poset permet de savoir :

- quels événements se produisent et à quel moment,
- quels événements sont causalement dépendants d'autres et réciproquement.
- quels événements sont causalement indépendants d'autres.

3.4.2 Les outils de vérification

Les posets générés pouvant être complexes à interpréter, Rapide fournit d'autres outils aux développeurs. Rapide définit ainsi un langage formel de définition de contraintes dans lequel le développeur peut spécifier les contraintes que doit vérifier son système. Un vérificateur permet ensuite de vérifier que le système modélisé ne viole pas ces contraintes. Les contraintes que l'on peut spécifier sont relatives à l'ordonnancement des événements dans le système.

3.4.3 Avantages et inconvénients

L'intérêt de Rapide par rapport aux autres langages étudiés est de fournir des réponses au problème de la description de la dynamique d'une application. Il est ainsi possible de vérifier la validité des architectures par des techniques de simulation de l'exécution d'une application.

Cependant, Rapide ne permet pas de générer une application et donc ne permet pas de la déployer, ni de l'administrer.

3.5 Wright

Comme les ADL étudiés précédemment, Wright [5] structure les systèmes à l'aide de composants et de connecteurs reliant les composants. Cependant, contrairement à UniCon, Wright est un langage purement descriptif orienté vers la vérification des protocoles entre les composants, plutôt que sur la correction fonctionnelle de l'architecture globale. Il n'est pas utilisé pour construire ou déployer le système décrit. Il sert principalement à modéliser et à analyser le comportement dynamique du système et porte son accent sur la vérification formelle du système.

Pour ce faire, Wright définit le comportement des composants et des connecteurs dans un calcul proche de CSP[1].

Nous allons brièvement présenter les éléments constitutifs de l'architecture Wright à l'aide d'un exemple simple : un système client-serveur (figure 3.1).

Pour décrire l'architecture complète d'un système, les composants et connecteurs d'une description Wright sont combinés dans une configuration. Une configuration est une collection d'instances de composants reliées au moyen de connecteurs. La configuration du système est décrite en trois étapes.

[1] CSP (Communication Sequential Processes) décrit le comportement d'une architecture logicielle à travers un modèle algébrique de processus [45].

```
System ClientServer
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s : Server
    c : Client
    cs : C-S-connector
Attachments
    c.request as cs.client
    s.provide as cs.server
end ClientServer
```

Figure 3.1 – Un système client-serveur simple décrit dans le langage Wright.

3.5.1 Etape 1 : Définition des composants et des connecteurs (*System*)

Cette partie de la description de l'architecture permet la définition des composants et des connecteurs.

- **Un composant** est caractérisé par :
  - ◇ **une interface** qui est constituée d'un ensemble de *ports*. Chaque port représente un point logique d'interaction du composant avec son environnement. Le protocole suivi par le port est exprimé en CSP. Dans la figure 3.1, représentant la description d'un exemple composé de deux composants, l'un serveur et l'autre client, le client et le serveur n'ont qu'un seul port, mais, en général, un composant peut en avoir plusieurs.
  - ◇ **une spécification de composant** (*spec*) qui spécifie la fonction du composant en CSP : elle consiste en la description des interactions entre le composant et ses ports.
- **Un connecteur** détermine les interactions entre plusieurs composants : un « pipe », par exemple, représente un flot de données séquentiel entre deux filtres. Un connecteur est défini par :
  - ◇ **un ensemble de rôles** qui spécifient le comportement attendu de chacune des parties liées par le connecteur. Dans l'exemple retenu, le connecteur client-serveur illustré sur la figure 3.2 a un rôle client et un rôle serveur.
  - ◇ **un protocole** (*glue*) qui régit les communications entre les rôles reliés par ce connecteur. Dans l'exemple du connecteur client-serveur, le séquencement des activités par le protocole de communication impose la séquence suivante : le client émet une requête, le serveur traite la requête, le serveur fournit un résultat, le client obtient le résultat. Ce protocole est décrit en CSP.

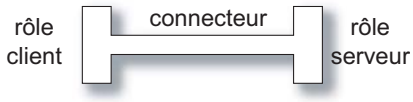

Figure 3.2 – Exemple de connecteur Wright.

3.5.2 Etape 2 : Les instances de composants et de connecteurs (Instance)

Cette partie de la description de l'architecture spécifie les entités qui vont réellement composer le système. Dans l'exemple de la figure 3.1, il n'y a qu'un seul serveur (*s*), un seul client (*c*) et un seul connecteur (*cs*).

3.5.3 Etape 3 : Les liaisons entre ports et rôles (Attachments)

Dans la phase finale, les instances de composants et connecteurs sont combinées pour décrire quels sont les ports de composants attachés aux rôles des connecteurs. Dans l'exemple, le port de requête du client (*c.request*) se voit lié au rôle de client du connecteur (*cs.client*), alors que le port de fourniture du serveur (*s.provide*) est attaché au rôle de serveur du connecteur (*cs.server*). Cela signifie que le connecteur *cs* va coordonner le comportement des ports *c.request* et *s.provide*.

A partir de cette description en trois parties, Wright génère un modèle CSP qui peut être analysé par un vérificateur de théorème externe. Celui-ci peut analyser différentes propriétés du système comme l'adéquation entre les ports des composants et les rôles des connecteurs, la nécessité de connecter certains ports d'un composant pour lui permettre de fonctionner correctement ou encore l'absence d'interblocage (boucle infinie, Deadlock) dans l'architecture.

3.5.4 Synthèse sur l'ADL Wright

Wright est un langage architectural qui se concentre sur le comportement du système. Celui-ci est caractérisé en termes d'événements significatifs qui peuvent prendre place dans les calculs des composants et des interactions entre ces composants (les connecteurs).

Wright met l'accent sur la spécification du comportement dynamique des applications. Il offre un formalisme qui permet non seulement de vérifier si les ports et les rôles sont compatibles du point de vue de l'échange de paramètres, mais aussi s'ils sont compatibles du point de vue modèle d'exécution des communications. Il permet ainsi une analyse des interblocages éventuels.

Cependant, Wright n'est pas dédié à la production d'une image exécutable de l'application. De plus, son modèle algébrique de processus est bien adapté à la description d'applications simples, mais il se complexifie dès lors que les applications considérées sont plus importantes.

3.6 Olan

Olan [14, 15, 13, 16] est un environnement de programmation qui vise à faciliter le développement, la configuration et le déploiement d'applications réparties construites par assemblage de composants hétérogènes. Son langage de description d'architecture, OCL (Olan Configuration Language), repose sur un modèle d'assemblage de composants logiciels. Nous allons présenter dans ce paragraphe le langage de description OCL.

3.6.1 Modèle de composants

Olan permet de décrire les architectures d'applications dont les abstractions de base sont les composants, entités d'intégration et de structuration de logiciels et les connecteurs, entités de gestion de la communication entre composants. Le concept principal véhiculé par le modèle d'assemblage est le composant.

Les dépendances fonctionnelles du composant avec le monde extérieur sont exposées au niveau de l'interface. Celle-ci permet de décrire de manière complète l'accès aux opérations et aux structures de données fournies par le composant.

L'interface peut aussi contenir des définitions d'attributs publics : ce sont des variables typées qui permettent de rendre accessibles au niveau du langage certaines données contenues initialement dans le code intégré et qui peuvent éventuellement changer en cours d'exécution. Certains attributs sont prédéfinis (tout composant possède par exemple un « site de chargement »), tandis que d'autres sont définis directement par le réalisateur.

3.6.2 Modèle d'interaction

Connecteur Les connecteurs sont des objets qui contrôlent les interconnexions de composants et spécifient le protocole requis à l'exécution. De façon similaire aux notions de rôles introduits dans UniCon, chaque connecteur Olan spécifie l'ensemble de composants dont il accepte la connexion.

Connexion Les connexions sont les communications effectives qui prennent place entre les composants. Elles peuvent être vues comme des instances d'un type de connecteur avec une implémentation spécifique dont les sources et destinations sont spécifiées.

3.6.3 Évolution dynamique

Des aspects dynamiques de la configuration d'une application peuvent être exprimés dans un programme OCL. Deux types d'instanciation existent : l'instanciation paresseuse et l'instanciation dynamique.

Instanciation paresseuse L'instanciation paresseuse permet de déclarer des composants qui ne seront pas tout de suite instanciés. Ces composants sont créés lorsque le service qu'ils fournissent est sollicité. Ce type de mécanisme ne permet d'instancier qu'un seul composant par interconnexion, contrairement à l'instanciation dynamique.

Instanciation dynamique Par opposition à l'instanciation paresseuse, de multiples instances peuvent être crées dynamiquement à partir d'une seule clause d'interconnexion.

3.6.4 Système à l'exécution

Les supports d'exécution d'Olan sont des plates-formes standards, possédant un service de déploiement. Parmi ces plates-formes, citons CORBA ou encore la plate-forme d'exécution A3 étudiée au chapitre 1.

3.6.5 Synthèse sur Olan

La description que le composant primitif fournit au travers d'une interface est homogène et indépendante de l'entité encapsulée, de son langage de programmation et de sa plate-forme

d'exécution. Un composant peut être utilisé dans n'importe quelle architecture à condition de satisfaire ses besoins fonctionnels et son modèle d'exécution. Ceci est une amélioration importante par rapport à UniCon où le langage de programmation n'était pas transparent au niveau des composants. De plus, OCL est plus souple en termes de définition de composants, son typage est moins contraignant. Il garde malgré tout la possibilité d'effectuer un grand nombre de vérifications sur l'architecture, en particulier avec les connecteurs et les règles d'interconnexion qui y sont attachées.

De plus, Olan offre un environnement complet de construction d'applications. Toutes les étapes du cycle de vie d'une application distribuée sont représentées, incluant la construction, l'installation, le déploiement et la configuration du système.

3.7 Conclusion

La communauté travaillant dans le domaine des architectures logicielles, sous-domaine du génie logiciel, a proposé des langages de description d'architectures (ou ADL pour *Architecture Description Language*), c'est-à-dire des langages qui fournissent le moyen de décrire formellement la structure et les propriétés qui caractérisent le comportement d'un logiciel complexe. Les ADL, couplés avec les modèles à composants, permettent la description de configuration sous forme de composants, connecteurs, ports, rôle, etc.

Nous pouvons distinguer deux types principaux d'ADL :

- OLAN et Unicon font parties de ceux qui permettent de générer une application mais qui ne prennent pas ou peu en compte les propriétés non fonctionnelles des applications.
- Rapide et Wright font partie de ceux qui permettent de modéliser et d'analyser le système en prenant en compte le comportement des applications[2] mais qui ne permettent cependant pas de générer des exécutables.

3.7.1 Limitations

Les ADL semblent avoir atteint leurs limites quant à la génération d'application. En effet, les outils de génération actuels savent générer une configuration initiale mais peuvent difficilement s'accommoder d'une reconfiguration dynamique de l'application qui est pourtant de première importance pour les systèmes distribués complexes actuels. Le problème est qu'un ADL embarque le lien entre l'architecture du système et son implémentation mais pas l'inverse. Un changement dans l'architecture implique un changement de l'implémentation qui est regénérée mais les changements dans l'implémentation ne sont pas reflétés dans l'architecture. De plus, le déploiement d'une configuration est souvent fait « d'un bloc », le changement d'un simple composant implique le remplacement de l'ensemble d'une configuration. Alors que la mise à jour partielle permettrait de prendre en compte les contraintes de disponibilité de certaines applications (notamment les application distribuées à grande échelle en environnement mobile). Cette dernière solution est proposée par Scalagent, qui présente un modèle hiérarchique très intéressant car il permet de répondre aux contraintes de scalabilité et de distribution. Nous nous inspirerons de ce modèle pour notre processus de déploiement à grande échelle présenté dans le chapitre 5. De plus, nous nous attarderons à lever certaines limites décrites dans notre langage de description d'intergiciel présenté dans le chapitre5.

3.7.2 Besoins pour les intergiciels asynchrones

Tous les ADL que nous avons présentés dans ce chapitre ont été conçus dans un but : la construction d'application (et le plus souvent d'application centralisée). Peu de travaux ont porté sur l'utilisa-

[2] Plus précisément : prennent en compte le comportement des composants de l'application.

tion d'un ADL pour la création d'une configuration d'intergiciel et encore moins pour les intergiciels asynchrones. Or, le besoin de plus en plus important de processus de déploiement (voir section2.3) nécessite l'utilisation d'ADL permettant de décrire les différentes phases d'initialisation ainsi que la gestion du cycle de vie des différents composants de l'intergiciel.

Nous proposerons dans le chapitre 5 un langage de description d'intergiciel (le MDL pour *Middleware Description Language*) qui répond aux besoins de composition, de configuration et de déploiement de notre modèle d'intergiciel asynchrone.

Dans le chapitre suivant, nous revenons sur l'ensemble des travaux que nous avons effectués et qui ont abouti à la description d'un nouveau modèle d'intergiciel et de son langage de description associé.

Chapitre 4

Configuration d'intergiciel asynchrone à grande échelle

Depuis quelques années, le développement des réseaux à grandes échelle (WAN, Wide-Area Networks) et, plus récemment, des périphériques mobiles omniprésents (PDA, carte à puce,...), offrent de nouveaux défis pour le développement des applications à grande échelle. Comme nous l'avons vu dans les chapitres précédents, le développement de systèmes logiciels distribués est simplifié par l'existence d'intergiciels qui traitent les problèmes récurrents d'hétérogénéité et d'interopérabilité et qui fournissent des services résolvant les problèmes de gestion de la distribution (services en charge de propriétés non fonctionnelles).

Ce chapitre présente nos travaux sur la configuration des intergiciels et met en avant les problèmes auxquels nous avons été confrontés.

4.1 Introduction

Pendant longtemps les intergiciels ont été développés en mettant en avant toujours plus de fonctionnalités. Mais cette fuite en avant soulève désormais de nombreux problèmes lors de l'utilisation des intergiciels sur des environnements à grande échelle. L'aspect monolithique des intergiciels asynchrones existants pose de gros problèmes de performances si l'on ne prend pas en compte l'hétérogénéité des sites d'exécution et des applications.

Le but de ce chapitre est de mettre en avant l'impact du manque de configurabilité, de configuration et de mécanismes de déploiement des intergiciels asynchrones. Pour cela, nous prenons deux exemples que nous avons abordés durant notre doctorat. Tout d'abord nos travaux ont porté sur l'impact de la propriété d'ordonnancement causal lors du passage à l'échelle d'un intergiciel asynchrone et ont mis en avant les faiblesses en termes de configuration des intergiciels asynchrones. Puis, le déploiement d'un intergiciel sur un environnement à grande échelle contraint (cartes à puce) a confirmé ces manques de configuration et de configurabilité limitant le portage de l'intergiciel sur carte à puce.

Ce chapitre reprend ces travaux dans les sections 4.2 et 4.3 et synthétise les résultats dans la section 4.4. Cette synthèse nous mène à la définition d'un modèle d'intergiciel asynchrone adaptable.

4.2 Un exemple de propriété : l'ordonnancement causal

Les intergiciels asynchrones actuels sont de nature « boîtes noires », ils embarquent un ensemble de fonctionnalités non « débrayables » et ne permettent pas une spécialisation de l'architecture et du

comportement. C'est notamment le cas de la propriété d'ordonnancement causal des messages que nous présentons ici. Cette propriété peut être très importante pour les applications distribuées car elle permet de garantir une certaine cohérence de l'application. En garantissant qu'une « conséquence » ne pourra jamais précéder sa « cause », la délivrance causal facilite le développement d'applications distribuées asynchrones à grande échelle dans lesquelles il est extrêmement difficile de contrôler la cohérence de l'exécution.

De nombreux algorithmes existent pour définir l'ordre causal. Le plus intéressant se base sur l'utilisation d'une horloge logique matricielle qui garantie la *délivrance* causale des messages. Cette matrice a une taille dépendante du nombre de processus (ou sites) mis en jeu dans l'application. La taille de la matrice augmente de façon quadratique en fonction du nombre de sites et les coûts associés augmentent de la même manière (comme nous le verrons dans les expérimentations de ce chapitre).

Ainsi, les intergiciels asynchrones qui proposent cette propriété subissent l'augmentation quadratique des coûts. Nous verrons dans la suite qu'il est possible de configurer cette propriété pour réduire ces coûts mais que l'aspect non-configurable des intergiciels asynchrones rend cette configuration difficile à mettre en place, alors que l'utilisation d'un intergiciel configurable aurait permis de mieux gérer la causalité.

4.2.1 Rappel sur la causalité

L'ordre causal (ou relation de précédence causale) des événements d'un système est un concept fondamental qui permet d'aider à résoudre les problèmes d'ordonnancement dans les systèmes distribués. L'ordre causal utilise le principe des horloges logiques introduit par Lamport dans [56].

Processus
Nous raisonnons sur un ensemble fini de processus $\mathbb{P} = \{p_1, \ldots, p_k\}$.

Evénement
Un événement peut être l'envoi, la réception d'un message par un processus ou un événement local à un processus. On note $e_i(p_j)$ un événement e_i se produisant sur le processus p_j. On note $\mathbb{EV}$ l'ensemble des événements survenant dans le système. Cet ensemble est fini.

Message
On note $\mathbb{M}$ l'ensemble des messages échangés dans le système.

Emission et réception de messages
On note $E_{M_i}(p_j)$ (resp. $R_{M_i}(p_j)$) l'émission (resp. la réception) du message M_i par le processus p_j. L'émission et la réception sont deux cas particuliers d'événements.

Dépendance causale
L'ordre causal est défini en utilisant la relation « *happens before* », elle-même définie par [56]. La dépendance causale d'un événement $e_k(p_l)$ par rapport à un événement $e_i(p_j)$ est noté $e_i(p_j) \rightarrow e_k(p_l)$ et intervient lorsqu'une des trois règles suivantes est vérifiée :

- **Règle (r1)** :
  $p_j = p_l$ et $e_i(p_j)$ se produit avant $e_k(p_l)$.
- **Règle (r2)** :
  $\exists M_k$ tel que $(e_i(p_j) = E_{M_k}(p_j)) \wedge (e_k(p_l) = R_{M_k}(p_l))$

– **Règle (r2)** :
$\exists(p_n, e_m(p_n)) \in \mathbb{P} \times \mathbb{EV}$ tel que $(e_i(p_j) \rightarrow e_m(p_n)) \wedge (e_m(p_n) \rightarrow e_k(p_l))$.

Ordonnancement causal des messages Un protocole **ordonne** causalement les messages échangés entre les processus d'un système lorsqu'il garantit que les processus **délivrent** les messages qu'ils reçoivent suivant leur « ordre de dépendance ». La délivrance des messages doit respecter les règles suivantes :

– Si un processus p envoie successivement les messages M_1 et M_2, alors aucun processus ne délivrera M_2 avant M_1.
– Si un processus p envoie un message M_1 et si un autre processus après avoir reçu M_1 envoie un message M_2, alors aucun processus ne délivrera M_2 avant M_1.

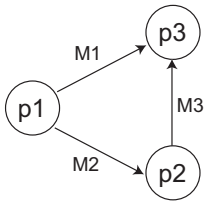

Figure 4.1 – Le principe de délivrance causale.

La figure 4.1 illustre le principe de la causalité. Dans cet exemple, le processus p_1 envoie un message M_1 au processus p_3, puis un message M_2 au processus p_2. Sur réception du message M_2, le processus p_2 envoie un message M_3 au processus p_3. La propriété d'ordonnancement causal assure que le message M_1 sera délivré à p_3 avant le message M_3.

Algorithme Soit n le nombre de sites, sur chaque site S_i on définit l'horloge matricielle comme une matrice $H_i[1..n, 1..n]$ initialisée à 0. Soit une composante quelconque de H_i : $H_i[j, k]$ est la vision de S_i de l'état du canal de communication de S_j vers S_k (c'est-à-dire le nombre de messages envoyés sur le canal du site j au site k). $H_i[i, *]$ et $H_i[*, i]$ sont évidemment à jour sur S_i, les autres composantes représentent la connaissance par S_i de l'état global. Lors d'un envoi de message m par S_i, de S_i vers S_j, H_i définit l'ensemble des envois de messages dont le message m dépend causalement. Le protocole d'envoi de message fonctionne comme suit :

A. Sur le site S_i, lorsqu'un message est envoyé sur le canal de communication $C_{i\rightarrow j}$ (canal de S_i à S_j) on exécute une incrémentation de l'horloge :
| $H_i[i, j] := H_i[i, j] + 1$
B. Chaque message M porte comme estampille H_M l'horloge matricielle H_i du site émetteur après la mise à jour. A la réception d'un message (M, H_M) envoyé par le site S_i, le site récepteur S_j retarde la délivrance de M jusqu'à ce que la condition suivante soit satisfaite :
| $\forall\, k \in [1..n], k \neq i H_M[k, j] \leq H_j[k, j]$ et $H_M[i, j] \equiv H_j[i, j] + 1$
La première condition ($\forall k \in [1..n], k \neq i H_M[k, j] \leq H_j[k, j]$) permet de vérifier qu'aucun site ne nous a envoyé un message qu'on aurait dû recevoir avant celui-ci (au sens causal) (voir le diagramme temporel de la figure 4.2).
La deuxième condition de délivrance ($H_M[i, j] \equiv H_j[i, j] + 1$) nous permet de vérifier la propriété de canaux FIFO (voir le diagramme temporel de la figure 4.3).
C. Après délivrance, l'horloge matricielle est mise à jour afin d'actualiser sa connaissance des autres canaux en se servant de la connaissance du site émetteur (valeur de H_M) :
| $Pour\ k = 1..n$ et $l = 1..n\ H_j[k, l] := Max(H_j[k, l], H_M[k, l])$
Une fois cette mise à jour effectuée, il est nécessaire de vérifier si un ou des messages en

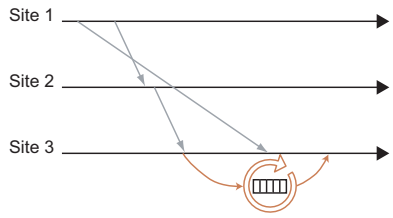

Figure 4.2 – Le message en *avance causale* est stocké pour une délivrance ultérieure.

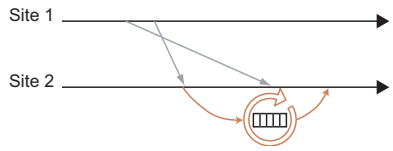

Figure 4.3 – Le message en *avance causale* est stocké pour une délivrance ultérieure.

attente peuvent être délivrés. Un message est pris en tête du tampon et le cycle reprend à partir de l'étape B.

4.2.2 Impact de la causalité

Comme on a pu le voir dans le chapitre 1, l'ordonnancement des messages peut prendre plusieurs formes. L'ordonnancement FIFO est le type d'ordonnancement qui est le moins coûteux car souvent il découle de l'utilisation de couche réseau comme TCP/IP qui intègre une gestion optimale de cette propriété. Par contre l'utilisation de l'ordonnancement causal introduit des problèmes significatifs lors du passage à l'échelle parce que l'augmentation linéaire du nombre de participants accroît de façon quadratique le coût de la gestion de la causalité[1].

Le facteur principal de perte de performance est lié au coût de *buffering* [32]. Par *buffering* il faut entendre tout ce qui se rapporte aux coûts de stockage (en mémoire et sur supports persistants) des horloges logiques et des messages en attente. Lorsque le nombre de participants augmente, le nombre de messages causalement dépendants augmente lui aussi. Le nombre de messages référencés dans des dépendances causales et qui n'ont pas été délivrés augmente proportionnellement en fonction des références. En conséquence, les besoins en *buffering* augmentent de façon quadratique avec le nombre de clients et augmentent les coûts de la même manière.

De nombreuses optimisations existent pour réduire le coût lié à l'ordonnancement comme l'envoi de messages par paquet, le débrayage de la gestion de l'ordonnancement du niveau système au niveau applicatif, le partitionnement des participants en groupes distincts suivant la topologie de l'application ou du réseau. Les sections suivantes proposent certaines solutions basées sur le découpage en domaines des participants.

4.2.3 Protocole d'ordonnancement causal : notion de domaines

La gestion de l'ordonnancement causal est très coûteuse car elle implique une croissance quadratique des coûts et entraîne une baisse de performance qui n'est pas acceptable. Il est donc nécessaire de trouver des moyens de limiter ces effets. Il existe plusieurs solutions pour restreindre ces coûts, chacun de ces moyens porte sur la gestion de *domaines de causalité*. Un domaine de causalité est un ensemble d'entités dans lequel on conserve la propriété d'ordonnancement, chacun de ces domaines

[1] Voir les évaluations réalisées dans la section 4.2.6.

étant relié aux autres. L'avantage de cette solution est d'alléger dans chacun de ces domaines le coût lié à l'ordonnancement causal. Néanmoins il est nécessaire de vérifier que si la causalité est respectée dans chacun des domaines alors elle est vérifiée sur l'ensemble de ces domaines interconnectés. Intuitivement il semble naturel de penser que le respect de la causalité dans chaque domaine respecte la causalité de façon globale. Les définitions et le théorème de la *causalité par transitivité* est explicité dans la section suivante, il nous démontre que cette propriété est vraie à la condition qu'il **n'existe pas de cycle** dans l'interconnexion des domaines.

4.2.4 La causalité par transitivité

4.2.4.1 Définitions

Soit $\mathbb{P} = \{P_1, ..., P_U\}$ un ensemble de processus, $\mathbb{D} = \{D_1, ..., D_V\}$ un ensemble de domaines, et $\mathbb{R}$ un sous-ensemble quelconque de $\mathbb{P} \times \mathbb{D}$. La relation $\in$ définie par $p \in d \Leftrightarrow (p, d) \in \mathbb{R}$ décrit la *répartition* des processus dans les différents domaines.

TRACES

Définition 1 *Une* trace *d'exécution est définie par la donnée de :*
- *un ensemble* $\mathbb{M} = \{m_1, ..., m_n\}$ *de messages*
- *deux fonctions* src *et* dst *de* $\mathbb{M}$ *dans* $\mathbb{P}$*, telles que* $\forall m \in \mathbb{M}, src(m) \neq dst(m)$
- *pour chaque processus* p*, un ordre total strict* $<_p$ *défini sur le sous-ensemble des messages émis ou reçus par* p *(défini par* $\mathbb{M}_p = \{m \in \mathbb{M} | src(m) = p \vee dst(m) = p\}$*).*

$src(m)$ désigne naturellement le processus émetteur de m, et $dst(m)$ le processus récepteur de m. $m <_p m'$ signifie que l'émission (ou la réception) de m par p a lieu avant l'émission (ou la réception) de m' par p. Cette définition exclue volontairement la diffusion de messages, l'envoi simultané de plusieurs messages, la réception simultanée de plusieurs messages... Une trace peut être représentée par un diagramme temporel comme celui de la figure 4.4.

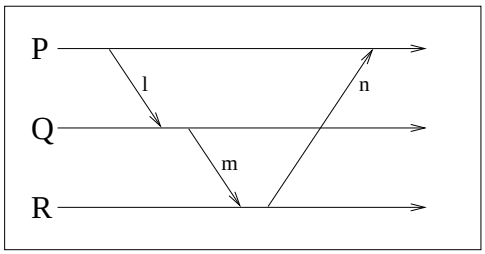

Figure 4.4 – Représentation d'une trace

Définition 2 *On dit qu'une trace est* valide *si elle vérifie :* $\forall m \in \mathbb{M}, \exists d \in \mathbb{D}, src(m) \in d \wedge dst(m) \in d$*.*

Définition 3 *La* restriction *d'une trace à un domaine* d *est la trace définie par :*
- *l'ensemble des messages dont la source et la destination appartiennent à* d *:* $\mathbb{M}_d = \{m \in \mathbb{M} | src(m) \in d \wedge dst(m) \in d\}$
- *les fonctions* src_d *et* dst_d *obtenue par restrictions des fonctions* src *et* dst *à* $\mathbb{M}_d$
- *pour chaque processus* p*, l'ordre total* $<_{d_p}$ *obtenu par restriction de l'ordre total* $<_p$ *à* $\mathbb{M}_d$*.*

CAUSALITÉ

Définition 4 *On dit que* m dépend causalement *de* n*, et on note* $n \prec m$*, si l'une des conditions suivantes est vérifiée :*

- m *et* n *sont émis par un même processus* p*, et* m *est émis après* n *:* $\exists p \in \mathbb{P}, src(n) = p \wedge src(m) = p \wedge n <_p m$
- m *est émis par un processus ayant précédemment reçu* n *:* $\exists p \in \mathbb{P}, dst(n) = p \wedge src(m) = p \wedge n <_p m$
- *il existe un message* l *tel que* $n \prec l$ *et* $l \prec m$*.*

Définition 5 *On dit qu'une trace est* correcte *si elle est valide et si la relation* $\prec$ *définit une relation d'ordre partiel sur les messages.*

Cette définition permet d'exclure les traces qui ne peuvent pas se produire dans la réalité, comme dans la figure 4.5, où un message « remonte dans le temps ».

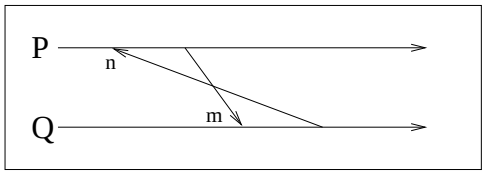

Figure 4.5 – Une trace incorrecte

Définition 6 *Une trace correcte* respecte la causalité *si, pour chaque processus* p*, l'ordre de réception des messages est compatible avec l'ordre causal :* $\forall p, m, n \in \mathbb{P} \times \mathbb{M} \times \mathbb{M}, (dst(m) = p \wedge dst(n) = p \wedge n \prec m) \Rightarrow n <_p m$*.*

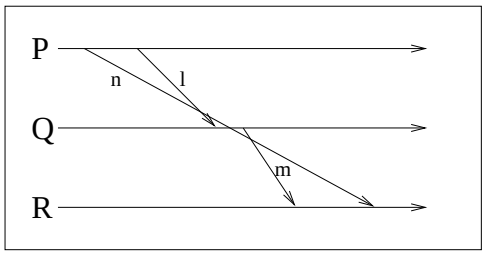

Figure 4.6 – Violation de la causalité

Définition 7 *On dit qu'une trace correcte* respecte la causalité dans le domaine d *si la restriction de cette trace au domaine* d *respecte la causalité.*

Par exemple, la trace de la figure 4.7 respecte la causalité dans D_1 mais pas dans D_2.

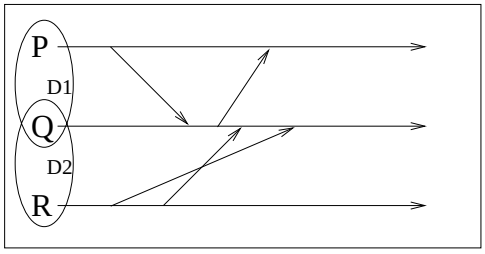

Figure 4.7 – Respect de la causalité dans un domaine

CHEMINS

Définition 8 *Un* chemin de p_1 à p_c *est une suite non vide* $[p_1, ..., p_c]$ *de processus tels qu'il existe, pour chaque couple de processus consécutifs, un domaine contenant ces deux processus :* $\forall i < c, \exists d \in \mathbb{D}, p_i \in d \wedge p_{i+1} \in d$.

Définition 9 *Soit* $C = [p_1, ..., p_c]$. *Alors, par définition, la* longueur *de* C *est égale à* c, *sa* source *est égale à* p_1, *et sa* destination *est égale à* p_c.

Définition 10 *Un* chemin direct de p_1 à p_c *est un chemin* $[p_1, ..., p_c]$ *tel que tous les processus soient 2 à 2 distincts :* $i \neq j \Rightarrow p_i \neq p_j$.

Définition 11 *Un* chemin minimal de p_1 à p_c *est un chemin direct* $[p_1, ..., p_c]$ *tel que* $i + 1 < j \Rightarrow \neg(\exists d \mid p_i \in d \wedge p_j \in d)$.

Définition 12 *Un* cycle *est un chemin direct tel qu'il existe un domaine contenant la source et la destination du chemin, et tel qu'il n'existe pas de domaine contenant tous les processus du chemin.*

CHAÎNES

Définition 13 *Etant donné une trace, une* chaîne *est une suite non vide* $(m_1, ..., m_k)$ *de messages de cette trace, telle que chaque message est émis après réception du précédent :* $\forall i < k, \exists p \in \mathbb{P}, dst(m_i) = p \wedge src(m_{i+1}) = p \wedge m_i <_p m_{i+1}$.

Définition 14 *Soit* $C = (m_1, ..., m_k)$. *Alors, par définition, la* longueur *de* C *est égale à* k, *sa* source *est égale à* $src(m_1)$, *et sa* destination *est égale à* $dst(m_k)$.

Définition 15 *On définit le* chemin associé à une chaîne $(m_1, ..., m_k)$ *d'une trace* valide *par :*

$$[src(m_1), src(m_2), ..., src(m_k), dst(m_k)]$$

Le chemin ainsi défini est bien un chemin : deux processus consécutifs correspondent en effet à la source et à la destination d'un même message, qui appartiennent par conséquent à un même domaine (puisque la trace est supposée valide).

Définition 16 *Une* chaîne directe *est une chaîne dont le chemin associé est direct. Une* chaîne minimale *est chaîne dont le chemin associé est minimal.*

TRACES ABSTRAITES La répartition des processus en domaines n'est visible qu'au niveau système. Au niveau applicatif, les processus s'échangent des messages « abstraits » qui ne sont pas contraints par les domaines (un message « abstrait » peut aller d'un processus à un autre directement, même s'ils ne font pas partie d'un même domaine). Par contre, ces messages « abstraits » sont représentés, au niveau système, par des chaînes de messages « concrets ». Les définitions suivantes précisent les conditions dans lequelles une trace peut être vue comme une trace de messages abstraits correspondant à une trace concrète. Elles définissent en fait comment les messages abstraits sont implémentés par des messages concrets.

Définition 17 *Une* abstraction *d'une trace est un ensemble de chaînes* minimales $C = \{c_1, ..., c_k\}$ *tel que* $\forall (m_1, ..., m_k) \in C, \forall i < k, p = dst(m_i) \Rightarrow \neg(\exists m, m_i <_p m <_p m_{i+1})$.

Définition 18 *On dit qu'une trace T' est une trace abstraite associée à une trace T s'il existe une abstraction $C = \{c_1, ..., c_k\}$ de T telle que T' soit égale à la trace déduite*[2] *de T en considérant chaque chaîne $(m_1, ..., m_k)$ de C comme un message direct de $src(m_1)$ à $dst(m_k)$.*

Remarques :

- ces définitions signifient que les messages abstraits sont représentés par des chaînes de messages concrets « directes » (sans boucles, sans détours inutiles...) et qui ne se « croisent » pas. Elles excluent les situations du genre de celles de la figure 4.8 (où l'abstraction - invalide - est consituée des 2 chaînes entourées).
- toute trace peut être vue comme une trace abstraite associée à elle-même (en utilisant l'abstraction $C = \{(m_1), .., (m_k)\}$).

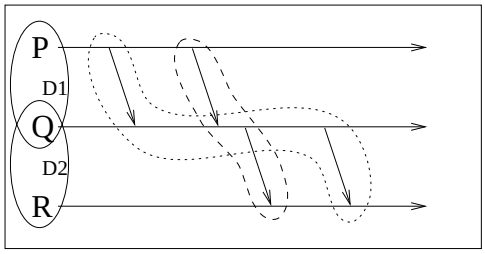

Figure 4.8 – Abstraction invalide d'une trace

4.2.4.2 Lemmes

Lemme 1 *Aucun domaine ne contient la source et la destination d'un chemin minimal de longueur supérieure ou égale à 3.*

Lemme 2 *Si $(m_1, ..., m_k)$ est une chaîne dont la source p et la destination q sont différentes, et si la trace est* correcte, *alors il existe une chaîne directe $(n_1, ..., n_l)$ ayant même source et même destination, et telle que $m_1 \leq_p n_1$ et $n_l \leq_q m_k$.*

Lemme 3 *Si m dépend causalement de n, alors soit il existe une chaîne $(n, ..., m)$, soit il existe une chaîne $(l, ..., m)$ où l est un message émis après n par le processus ayant émis n.*

Lemme 4 *Si une trace correcte ne respecte pas la causalité, alors il existe deux processus p et q, un message n de p vers q, et une chaîne $(m_1, ..., m_n)$ de p à q telle que $n <_p m_1$ et $m_n <_q n$.*

[2] La définition précise est trop pénible à détailler : (

4.2.4.3 Théorème

On souhaite montrer que si une trace correcte respecte la causalité dans chaque domaine, alors toute trace abstraite associée respecte la causalité de façon globale. En fait, c'est faux dans le cas général, comme le montre l'exemple de la figure 4.9 (en prenant comme trace abstraite la trace elle-même).

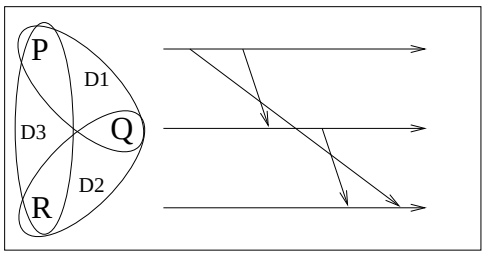

Figure 4.9 – Un contre-exemple

Enoncé

Théorème 1 *Les deux propositions suivantes sont équivalentes :*

P1 toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, respecte la causalité de façon globale.

P2 il n'existe pas de cycles.

Un corollaire immédiat, en utilisant le fait qu'une trace peut être vue comme une trace abstraite associée à elle-même, est que s'il n'existe pas de cycles alors toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale. D'autre part, la preuve de $P1 \Rightarrow P2$ ci-dessous montre aussi l'implication inverse : si toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale, alors il n'existe pas de cycles.

Les preuves des lemmes et du théorème sont données en annexe A.

4.2.5 Synthèse

Les sections précédentes ont présenté la notion de domaine de causalité. Nous avons prouvé que la causalité sur l'ensemble de domaines interconnectés est respectée s'il n'existe pas de cycle de domaines.

Cette notion soulève de nouveaux problèmes quant aux critères d'utilisation des domaines. Il est possible d'utiliser des critères physiques (machines, topologie réseau, architecture de l'intergiciel, etc.) ou des critères applicatifs (topologie de l'application, besoins de l'application). Cette séparation en domaines peut donc se réaliser à deux niveaux. Au niveau de l'architecture de l'application tout d'abord, dans ce cas les entités sont représentées par les composants de l'application. La gestion de l'ordonnancement intervient donc au niveau des échanges de messages entre ces clients, c'est ce qu'on appelle l'ordonnancement par topologie applicative. La deuxième approche se situe au niveau du système lui-même suivant l'architecture du réseau, dans ce cas les entités sont représentées par les sites de l'intergiciel. La gestion de l'ordonnancement intervient donc au niveau des échanges de messages entre les serveurs, c'est ce qu'on appelle l'ordonnancement par topologie réseau.

Dans tous les cas, les algorithmes de réduction de coûts (quantité de stockage limitée, taille des estampilles des messages réduite, etc.) impliquent une configuration de l'intergiciel en fonction des

besoins des applications et/ou des contraintes du système. La nécessité d'un intergiciel configurable et configuré devient donc capitale.

4.2.5.1 Domaines par topologie applicative

Nous avons présenté l'utilisation des domaines par topologie applicative dans [74]. L'idée vient d'une constatation simple : le coût d'une implémentation d'un ordonnancement causal des messages pourrait être nettement réduit si l'on pouvait connaître quels processus (ou composants applicatifs clients) d'une architecture distribuée avaient à communiquer ensemble plutôt que de supposer toutes les combinaisons possibles et donc d'utiliser une énorme matrice pour l'ensemble des processus.

Nous avons eu l'occasion d'étudier une application qui exprime clairement ces contraintes, l'application *NetWall*, un pare-feu développé par Bull et qui se sert de l'intergiciel asynchrone A3 pour gérer ses fichiers de log. Cette application a une topologie applicative très singulière (voir figure 4.10), tous les pare-feu sont reliés à un poste d'administration central unique. Chacun utilisant un site d'intergiciel pour communiquer. Les seules informations qui transitent se font entre le serveur central et les serveurs des pare-feu. Ce type de topologie possède des propriétés spécifiques qui agissent sur la forme de l'horloge matricielle, ainsi, seules la première ligne et la première colonne sont utilisées pour gérer l'ordonnancement causal[3], le reste de la matrice n'étant jamais mis à jour et reste à la valeur 0.

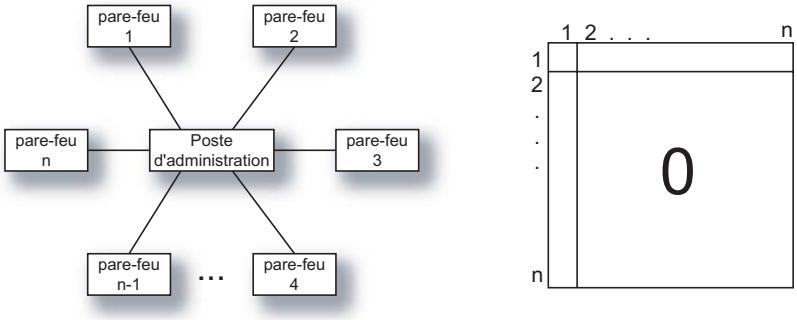

Figure 4.10 – La topologie applicative de NetWall et son horloge matricielle associée.

Avec cette propriété le coût[4] de la gestion d'une matrice complète paraît superflu. L'utilisation de langage de description d'architecture (ADL) pour *NetWall* nous permet de savoir exactement quels sont les canaux de communication et comment ils sont utilisés (voir figure 4.11).

Grâce à cette représentation on voit immédiatement les canaux de communication, il est donc aisé de savoir (dans ce cas) que les messages transitant toujours dans la même direction, la matrice aura certaines parties dites creuses, c'est-à-dire des cases qui ne seront jamais utilisées. On pourrait donc conserver la matrice sous la forme de vecteurs beaucoup moins coûteux et qui permettraient d'améliorer les performances de la gestion de l'ordonnancement et donc de l'intergiciel asynchrone dans sa globalité.

4.2.5.2 Domaines par topologie réseau

Le principe des domaines de causalité par topologie réseau consiste à séparer l'ensemble des sites de l'intergiciel en sous-domaines dans lesquels la causalité est respectée et de relier ces domaines par

[3] L'ordre pourrait être total et ne pas nécessiter d'horloge matricielle, si on prenait pour hypothèse l'utilisation de canaux FIFO.

[4] Place mémoire utilisée, sauvegarde sur disque de toute la matrice, test de réception sur l'ensemble de la matrice, etc.

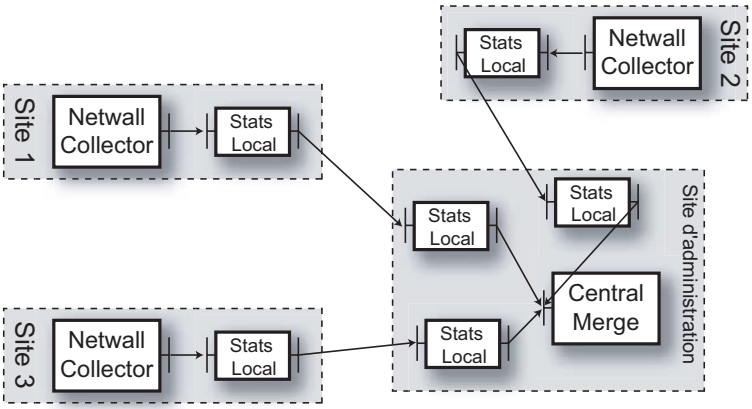

Figure 4.11 – Représentation de l'architecture de NetWall grâce à un ADL.

des sites spécifiques à cheval sur les domaines. Il est important de noter qu'on ne parle pas ici de la topologie réseau réelle de l'intergiciel mais bien de la topologie selon laquelle sera « découpée » la gestion de la causalité. C'est une vision logique du découpage de la causalité en sous-domaines[5]. L'utilisation des domaines permet d'utiliser n'importe quelle architecture de découpe[6]. Il est donc possible d'avoir une organisation en un graphe quelconque **acyclique** permettant d'être facilement transposé sur la topologie d'un LAN par exemple.

Dans la figure 4.12, l'intergiciel est représenté par un ensemble de huit sites (ou serveurs) interconnectés qui impose une matrice de taille 64 sur chaque serveur. Afin de réduire les coûts liés à l'horloge matricielle, cet ensemble est divisé en quatre domaines de causalité. le domaine A inclus $\{S1, S2, S3\}$, le domaine B inclus $\{S4, S5\}$, le domaines C inclus $\{S7, S8\}$ et les domaines D et E inclus respectivement $\{S3, S6\}$ et $\{S1, S4\}$.

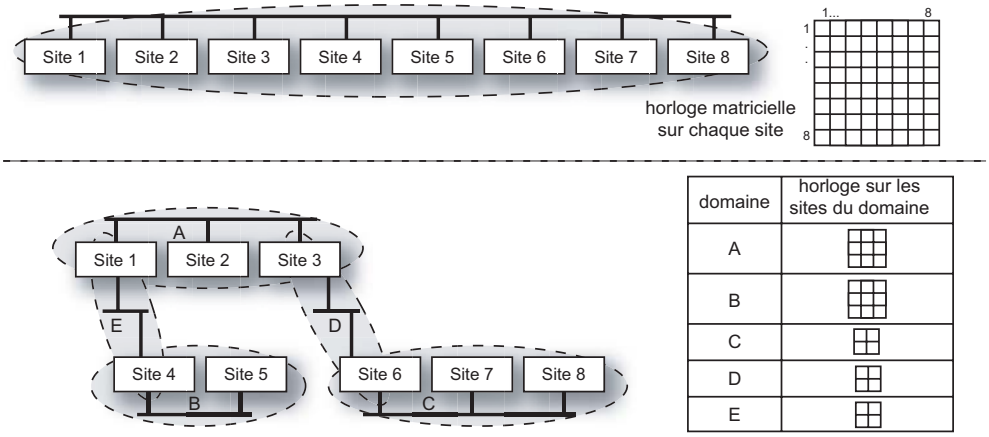

Figure 4.12 – Exemple de domaines de causalité par topologie réseau.

L'ordre causal est respecté dans chacun des domaines et un serveur appartenant à plus d'un domaine est appelé *serveur-routeur-causal*. Lorsqu'un client connecté au serveur 1 désire envoyer un message à un client connecté au serveur 8, le message sera routé en utilisant les chemins $S1 \rightarrow S3$

[5] Une idée assez proche avait été rapidement abordée dans [79] qui parlait de la notion de « voisins » où l'on ne mettait à jour que les éléments de la matrice qui correspondaient à un canal voisin (du même sous-réseau).

[6] A la différence de [10], qui impose une architecture en étoile et [2], qui impose une architecture hiérarchique.

(domaine A), $S3 \rightarrow S6$ (domaine D), puis $S6 \rightarrow S8$ (domaine C). Ce routage est assuré par le système et est complètement invisible aux clients qui n'ont pas la vision découpée de l'intergiciel. Le respect de la causalité uniquement dans chacun des sous-domaines permet de garantir son respect sur l'ensemble de l'intergiciel grâce à la propriété de *causalité par transitivité*.

La section suivante montre les améliorations de performance lors de l'utilisation des domaines de causalité par topologie réseau lors du déploiement d'un intergiciel à grande échelle.

4.2.6 Expérimentations

Nous avons effectué nos tests sur l'intergiciel asynchrone A3 que nous avons introduit dans le chapitre 1. Cet intergiciel étant le seul à proposer l'ordonnancement causal des messages, il permet donc de comparer plus facilement les résultats avec et sans gestion de la causalité par domaine.

La plate-forme A3 est composée d'un ensemble de serveurs dont le rôle est d'héberger les composants applicatifs (appelés agents) et de leur fournir les fonctions dont ils ont besoin : communication, persistance, ordonnancement, etc. L'ensemble des serveurs forment un bus logiciel où chaque serveur connaît (c'est-à-dire « sait accéder à ») tous les autres serveurs.

La gestion des domaines de causalité implique un découpage du bus en plusieurs « sous-bus » interconnectés dans lesquels la causalité est respectée. Cette transformation doit être transparente pour les clients applicatifs ; c'est-à-dire que le nom des agents et leur façon de communiquer doivent être inchangés. Deux problèmes doivent être résolus : la gestion de plusieurs matrices (dans le cas d'un serveur-routeur-causal appartenant à plusieurs domaines) et la gestion du routage des messages.

Modification des serveurs L'utilisation des domaines de causalité nous a obligés à revoir en profondeur le code de l'intergiciel et son aspect monolithique n'a pas simplifié la mise en place de l'architecture de cette expérimentation.

Pour résoudre le premier problème, nous avons donc créé sur chaque serveur un bus local de domaine (structure appelé *MessageConsumer*) pour chaque domaine dans lequel est inclus le serveur. Pour résoudre le problème du routage des messages nous avons suivi l'approche des protocoles réseau classiques en utilisant une simple table de routage. Ces nouvelles structures sont représentées par la figure 4.13.

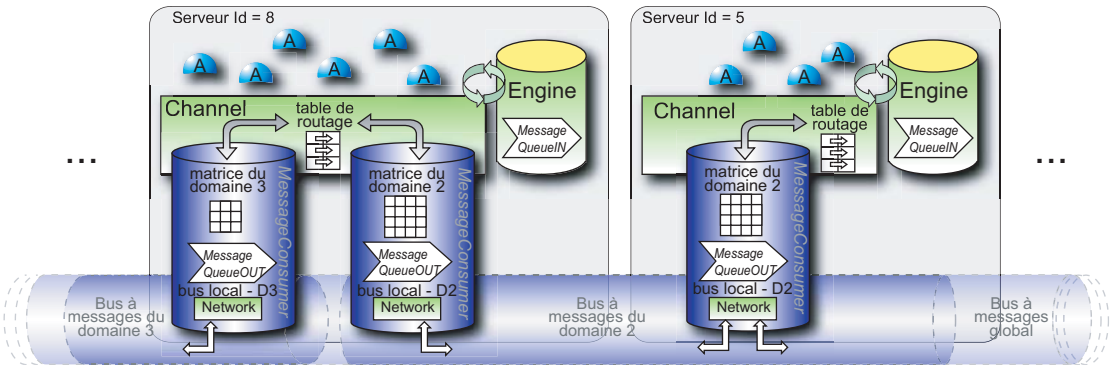

Figure 4.13 – Exemples de la nouvelle structure des serveurs.

Dans cette implémentation, un serveur d'agent a autant de *MessageConsumer* que de domaines auxquels il appartient. Chaque *MessageConsumer* possède une queue de messages et une horloge matricielle spécifique à son domaine. La table de routage donne pour chaque serveur destinataire

l'identifiant du serveur auquel envoyer réellement le message, à savoir : le serveur destinataire s'il s'agit d'un serveur dans le même domaine, ou le serveur-routeur sinon. Les tables de routage sont construites[7] statiquement au démarrage de l'intergiciel.

Le *Channel* assure la transmission fiable des messages et leur ordre causal. Il pose les messages dans le *MessageConsumer* en utilisant les informations de la table de routage, puis les estampille avec la matrice correspondante au domaine dans lequel ils sont envoyés. A la réception, le *Channel* vérifie l'estampille du message et le redirige soit vers la queue de message locale (si le message est destiné à un agent de ce serveur), soit au *MessageConsumer* correspondant au domaine destinataire. Les modifications faites au code du *Channel* sont données dans la figure4.14.

MessageConsumer Emetteur

```
evt = Get_Agent_Sent_Event()
// un agent envoi un événement
domainDestServer= RoutingTable[evt.dest]
mc = MessageConsumer(domainDestServer)
// choisit le MessageConsumer correspondant
// au domaine du serveur destination
stamp = mc.matrixclock(domainDestServer)
// récupère l'estampille pour le message
// = la matrice du domaine destinataire
msg = evt + stamp
mc.network.Send(msg) ⟶
// le message est sauvé dans
// MessageQueueOUT puis envoyé

Recv(ACK) ⟵
Remove(evt)
// supprime le message de MessageQueueOUT
```

MessageConsumer Recepteur

```
⟶ msg = mc.Recv
// réception d'un message par
// un MessageConsumer, le message
// est sauvé dans la queue locale
⟵ Send(ACK)
Check(mc.matrixclock)
mc.matrixclock.update(msg.stamp)
IF (evt.dest == this.server)
 Push(evt, QueueIN)
 // pose l'évènement dans la queue
 // QueueIN, l'Agent destinataire
 // pourra y réagir
ELSE
 systemDest= RoutingTable[evt.dest]
 mc = MessageConsumer(systemDest)
 // choisi le MC correspondant
 // au domaine du serveur destination
 stamp = mc.matrixclock(systemDest)
 // récupère l'estampille du message
 // = la matrice du domaine destinataire
 msg = evt + stamp
 mc.network.Send(msg)
 // le message est transferé dans
 // la MessageQueueOut et envoyé
 // au prochain serveur-routeur ou
 // directement au serveur
FI
```

Figure 4.14 – Modification du *Channel*.

Nous avons réalisé un ensemble de tests sur des systèmes centralisés et distribués à partir de cette implémentation. Le protocole de test utilisé ainsi que l'ensemble des résultats peuvent être trouvés dans [59] et [57]. Nous exposons les principaux résultats dans le paragraphe suivant.

[7] Les tables de routages sont construites grâce à un algorithme du « plus court chemin ».

Evaluation Les mesures réalisées initialement (sans les domaines de causalité) montrent clairement l'augmentation quadratique du coût de gestion de l'ordonnancement en fonction du nombre de serveurs et ceci en expérimentation centralisée ou distribuée. Les graphiques 4.15 et 4.16 montrent les résultats standards.

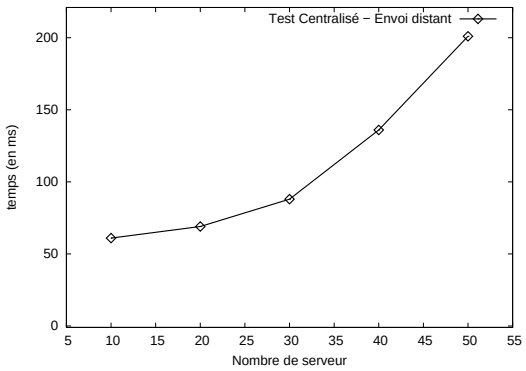

Figure 4.15 – Test centralisé sans *domaines de causalité*.

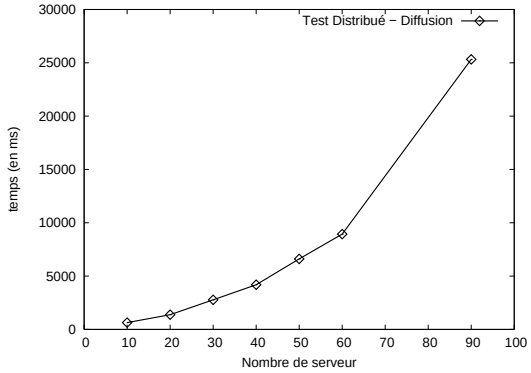

Figure 4.16 – Diffusion sans *domaines de causalité*.

Les expérimentations avec la gestion des domaines de causalité ont été réalisées avec une architecture de domaines en *bus*. D'autres tests utilisent des architectures en étoile ou en arbre (voir figure 4.17).

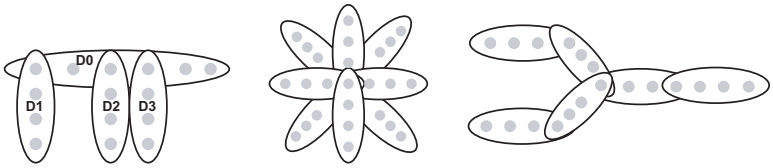

Figure 4.17 – Architectures de domaines en bus, en étoile et hiérarchique.

Le graphique 4.18 affiche une augmentation linéaire du temps de communication. Afin d'exprimer cette dépendance linéaire, considérons un arbre de d domaines, où chaque domaine possède exactement k sous-domaines et s serveurs ($2 \leq k \leq s-1$). Le nombre total de serveurs est donc $n = 1 + (s-1)(k(d+1)-1)/(k-1) \approx sk^d$ et le coût maximal d'envoi de message est de $C \approx (2d+1)s^2$ (le coût d'envoi d'un message dans un domaine de s serveurs est supposé être s^2).

Le coût linéaire observé résulte de notre découpage en $\sqrt{n}$ domaines de $\sqrt{n}$ serveurs avec une profondeur fixée à $d = 1$ (cas de l'architecture en bus), ce qui permet d'obtenir un coût $C \approx K \times n$. Dans le cas d'un arbre où s et k sont fixés (et $d > 1$), il serait même possible d'obtenir un coût logarithmique, car $C \approx 2ds^2 \approx 2s^2(\ln(n) - \ln(s))/\ln(k) \leq 2s^2 \ln(n)/\ln(k)$, donc $C \approx K' \times \ln(n)$. Cependant, $K' > K$ (en particulier si on prend en compte le coût du routage des messages, proportionnel à d), donc un arbre peut être moins efficace qu'un bus dans certains cas.

Le graphique 4.19 montre clairement le gain de performance apporté par l'utilisation des domaines de causalité.

Tous ces résultats dépendent relativement de notre plate-forme de test mais nous pensons qu'ils peuvent être considérés comme suffisamment généraux pour refléter les coûts d'utilisation d'une horloge matricielle dans un intergiciel asynchrone persistant.

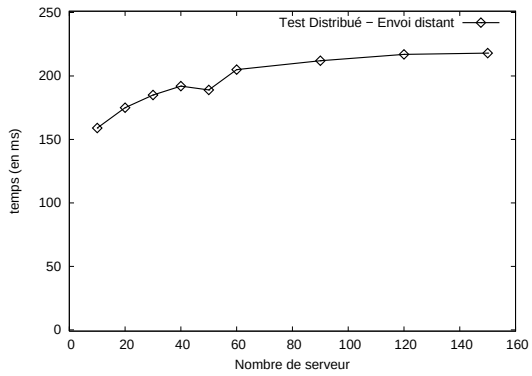

Figure 4.18 – Test centralisé avec *domaines de causalité*.

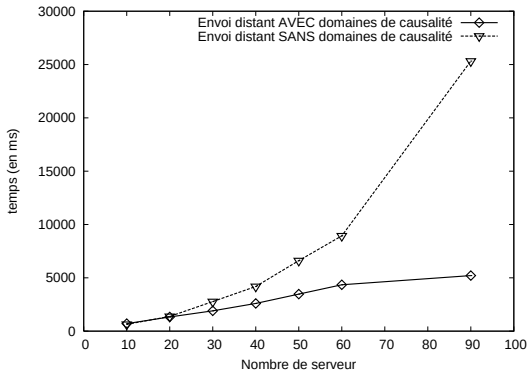

Figure 4.19 – Comparaison des coûts avec et sans gestion des *domaines de causalité*.

4.2.7 Bilan

Nos travaux sur les domaines de causalité ont été présentés dans [57, 58, 59, 74], ils ont permis de montrer que le découpage d'un intergiciel asynchrone en plusieurs groupes interconnectés permettait de réduire significativement les coûts associés à l'ordonnancement causal. Néanmoins, l'application des domaines de causalité aux intergiciels asynchrones ne va pas sans soulever de nombreux problèmes.

Les intergiciels asynchrones actuels sont assez monolithiques dans leur conception ; ils embarquent un ensemble de fonctionnalités non « débrayables » et ne permettent pas une spécialisation de l'architecture et du comportement. Dès lors, il devient difficile de mettre en place une propriété en fonction des besoins applicatifs spécifiques ou du système. Comme nous l'avons vu précédemment, il a été nécessaire de modifier en profondeur l'architecture de l'intergiciel A3 pour appliquer simplement la théorie des domaines par topologie réseau. De plus, cette expérimentation est limitée par son aspect statique. Il n'est pas possible de modifier les tables de routage en cours d'exécution, pire, il faut arrêter tous les sites et réinstaller la nouvelle architecture de découpage. Or, l'émergence des nouveaux périphériques omniprésents (PDA, cartes à puces, etc) crée de nouveaux besoins pour les intergiciels asynchrones. Ces tendances impliquent de nouveaux besoins en termes de configuration dynamique et d'adaptabilité des intergiciels. Par exemple, dans le cas des domaines de causalité, l'utilisation d'une architecture configurable statiquement et dynamiquement serait extrêmement avantageuse. Il deviendrait en effet possible de modifier le découpage des domaines en fonctionsde l'arrivée de nouveaux périphériques ou de leur mobilité géographique, etc. Et cela sans avoir à réécrire toute l'architecture de l'intergiciel et recommencer son installation sur l'ensemble des sites.

Il devient donc essentiel de fournir des mécanismes de contrôle pour aider à la configuration et au déploiement de l'intergiciel. Nous nous attarderons à proposer dans le chapitre suivant un modèle d'intergiciel asynchrone qui essaye de répondre à ces nouveaux besoins.

La section suivante revient sur nos travaux autour de la carte à puce et met en avant les difficultés d'adaptation d'un intergiciel asynchrone de type « boîte noire » sur une configuration système limitée.

4.3 Environnement à grande échelle : les cartes à puce

La section précédente a mis en avant les difficultés de configuration d'un intergiciel asynchrone lors de la modification d'une propriété non fonctionnelle. La volonté de changer le comportement de la propriété a demandé une refonte complète du code de l'intergiciel.

Dans cette section, nous présentons les difficultés liées au déploiement d'un intergiciel asynchrone sur un environnement à grand échelle : les cartes à puce.

Les cartes à puce font désormais partie intégrante de notre vie quotidienne. Cette technologie existant depuis plus de trente ans semble aujourd'hui parfaitement maîtrisée et pourtant de nouveaux problèmes apparaissent chaque jour. Les questions d'accès mobile à des services sont plus que jamais d'actualité dans un monde où les échanges se font désormais à l'échelle de la planète. Grâce aux progrès techniques importants réalisés ces dernières années, aussi bien au plan matériel que logiciel, les cartes à puce semblent désormais prêtes à répondre aux besoins de disponibilité et de sécurité des applications réparties modernes.

Une collaboration industrielle avec le laboratoire nous a permis d'expérimenter le portage d'un intergiciel asynchrone sur une plate-forme fortement contrainte qu'est la carte à puce.

4.3.1 Collaboration industrielle

Dans le cadre d'une collaboration entre l'INRIA et l'entreprise Schlumberger, nous avons mis au point un prototype d'intergiciel asynchrone sur carte à puce. Cette collaboration visait à développer des applications distribuées à base d'agents dans lesquelles la carte était un support d'exécution. Les objectifs de cette collaboration étaient donc multiples.

Tout d'abord Schlumberger s'oriente clairement vers des cartes à puce multiservice, c'est-à-dire des cartes à puces non plus dédiées à une seule application mais pouvant supporter plusieurs applications différentes, ceci afin d'éviter la multiplication des cartes dans le portefeuille de l'utilisateur. Il serait ainsi possible pour une même carte à puce d'être une carte de retrait bancaire, une carte de fidélité et une carte *Vitale* (carte d'assuré social). Dans ce cadre, un intergiciel asynchrone est vu comme un support d'installation et de mise à jour d'applications.

Ensuite, le développement d'un intergiciel à composants dans une carte à puce devait permettre une intégration de technologie à composants dans un système fortement contraint afin de prendre en compte le mode déconnecté et l'aspect mobilité de la carte. Les API de développement actuelles des cartes à puces étant limitées à un échange d'octets, l'apport de la sémantique des composants logiciels sur la carte devait permettre une amélioration du dialogue carte/monde extérieur. De plus, l'utilisation du formalisme de description de l'intergiciel (par un langage de description d'architecture) devait permettre de faciliter le développement d'applications réparties mettant en œuvre des cartes à puces. Dans ce cadre, un intergiciel asynchrone est vu comme un support de développement d'applications (réparties ou non) sous forme de services asynchrones.

De plus en plus, les cartes à puce sont vues comme des agents mobiles matériels qui, même si elles ne peuvent pas jouer ce rôle actuellement, deviennent des acteurs primordiaux de l'informatique omniprésente dans le cadre des fournisseurs de services mobiles.

Dans la section suivante nous détaillons le contexte du portage d'AAA sur carte à puce.

4.3.2 Utilisation de l'environnement JavaCard^{TM}

La technologie JavaCard est aujourd'hui utilisée par 90% des développeurs d'applications sur carte à puce. Il existe des alternatives à JavaCard, principalement des solutions propriétaires, comme les cartes Visual Basic de Microsoft par exemple. La portabilité des applications basées sur JavaCard est la raison principale du choix de cette technologie pour ce projet. Le succès actuel du langage Java peut aussi expliquer l'engouement pour la technologie JavaCard. Sur la carte à puce, les applications sont exécutées dans l'environnement d'exécution JavaCard (*JavaCard Runtime Environment*) [36]. Cet environnement contient la machine virtuelle JavaCard (JavaCard Virtual Machine), les classes de l'interface applicative JavaCard et les services associés (comme l'installateur d'applications par

exemple). Nous n'entrerons pas dans les détails de l'architecture et de l'API JavaCard car ce n'est pas le but de ce document. Néanmoins, nous précisons quelques notions sur l'environnement qui permettront de mieux comprendre les contraintes du contexte de travail.

Limitations de l'environnement JavaCard Les cartes à puce fonctionnent selon le modèle client/-serveur établi entre la station d'accueil (le client) et la carte (le serveur). C'est donc le code dans la station qui prend l'initiative d'appeler les méthodes dans la carte qui ne peuvent que répondre, et non l'inverse. Pour assurer la communication entre la station d'accueil et l'environnement d'exécution sur la carte, la norme `ISO78164` définit un protocole basé sur les unités de données applicatives (*Application Protocol Data Unit, APDU*) qui permet d'encapsuler les données dans un format commun aux deux parties. Une APDU peut être vue comme un paquet (au sens réseau) qui possède des champs de contrôle et un champ de données. Ce protocole de très bas niveau n'est pas pratique à utiliser, notamment à cause de la limitation en taille des messages envoyés (au maximum 261 octets pour une commande). Il est donc difficile d'envoyer des paramètres complexes (tableaux, objets, etc.) aux fonctions dans la carte, puisqu'il faut les découper en plusieurs APDU que l'application « encartée » devra reconstituer elle-même.

La technologie JavaCard fournit un mécanisme d'invocation de méthodes à distance qui permet aux méthodes situées sur la station d'accueil d'appeler les méthodes stockées dans la carte par le biais de la méthode *process*. Ce système facilite le travail du programmeur, bien qu'il soit obligé de manipuler des APDUs pour sélectionner la méthode qu'il désire invoquer et lui passer des paramètres.

La quantité de mémoire disponible sur les cartes à microprocesseur étant en général très limitée, ainsi que la puissance de calcul des processeurs utilisés, le langage utilisé dans l'environnement d'exécution JavaCard est un sous-ensemble du langage Java. Par exemple, on ne peut y trouver de types réels par exemple, ni de gestion des chaînes de caractères. De même, la machine virtuelle JavaCard est très réduite par rapport à la machine virtuelle Java classique. Elle ne supporte pas le chargement dynamique des classes et n'inclut pas de support pour les processus légers Java (*Java threads*), ni de ramasse-miettes (*garbage collector*). De plus tout objet alloué en mémoire est persistant et il est impossible de l'effacer (pas de méthode de désallocation).

Expérimentation Dans le cadre de la collaboration industrielle, nous avons réalisé le portage sur carte à puce de l'intergiciel asynchrone A3 introduit dans le chapitre 1. L'intergiciel A3 est composé de deux parties, le support de communication asynchrone fiable à base de queues de messages causalement ordonnés et le modèle à agent où les agents sont des objets réactifs autonomes clients de la couche de communication. Le but de l'expérimentation a été de fournir un environnement permettant d'avoir des agents sur la carte à puce avec les mêmes fonctionnalités que ceux du « monde A3 » et de les faire communiquer. Mais les restrictions de la JavaCard ne nous permettant pas de réaliser tel quel l'intergiciel A3 sur la carte à puce, nous avons recréé des conditions similaires à une JVM standard en perfectionnant l'environnement JavaCard.

Perfectionnement de l'environnement L'un des premiers travaux a donc été de lever les limitations afin de fournir une API minimale pour un développement plus propre et plus aisé. Nous avons donc développé une couche de communication entre la carte et la station d'accueil permettant l'envoi de flux de données sous forme de plusieurs APDU, la sérialisation d'objet (et de graphe d'objets) et enfin la gestion de type d'objets. L'architecture de cette API est protégée par un brevet français ([40]) et a fait l'objet d'un dépôt de brevet international en cours de validation. Nous ne rentrerons donc pas dans les détails de son implémentation.

La première partie de la couche de communication est responsable du transfert des octets de et

vers la carte. Elle est pilotée par les interfaces de lecture/écriture (*read/write*) de l'applet. Les octets sont envoyés (et reçus) via les APDU et stockés dans le (lu à partir du) composant de flux (*stream*), un tampon circulaire qui implémente les flux d'entrée/sortie standard de Java (voir figure4.20).

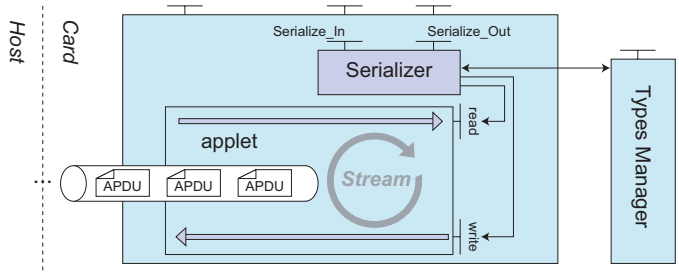

Figure 4.20 – La couche de communication JavaCard étendue.

La deuxième partie de la couche de communication est responsable de la conversion d'un flux d'octets non typé en un objet typé (et vice versa). Elle est pilotée par les interfaces de sérialisation en entrée et en sortie du composant de sérialisation (*Serializer*, *Serialize_In/Serialize_Out interfaces*). Pour envoyer des données hors de la carte, le composant de sérialisation transforme un objet typé (par le gestionnaire de types) en un flux d'octets qui est ensuite directement transformé sous forme d'APDU et transféré hors de la carte. De la même manière, lors d'une arrivée d'APDU, le composant de sérialisation vide le tampon de flux d'octets et le transforme en un objet géré par le gestionnaire de types. Le gestionnaire de types (*TypesManager*, voir figure4.21) rend le processus de sérialisation possible grâce aux méthodes d'encodage et de décodage (*encode/decode*).

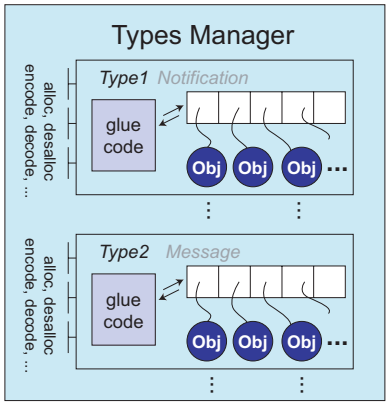

Figure 4.21 – Le composant de gestion des types.

Chaque type possède un code spécifique d'encodage sous forme d'octets et de décodage d'octets sous forme d'objet typé (*glue code*). Ce code est statiquement généré par le compilateur d'application carte qui embarque pour chaque type un code spécifique de sérialisation/désérialisation. De plus, le gestionnaire de types est responsable de l'allocation et de la désallocation des objets typés. Comme il est impossible dans l'environnement JavaCard de désallouer un objet de la mémoire, le gestionnaire de type garde un pointeur sur chaque objet alloué et pourra s'en resservir pour un autre objet du même type (c'est-à-dire réécrire le nouvel objet dans l'emplacement mémoire de l'objet inusité). Le gestionnaire de type fait donc office de ramasse-miette.

4.3.3 Architecture

L'architecture de A3 sur carte à puce est séparée en deux parties distinctes, le côté carte à puce et le coté station d'accueil (ou terminal). Le coté carte à puce inclut une implémentation spécifique de l'intergiciel A3 utilisant les améliorations faites à l'environnement JavaCard. Le côté terminal inclut un ensemble de serveurs A3 standards possédant des agents spécifiques interfacés avec l'API de communication d'APDU du terminal (connectée à la carte). L'architecture globale est décrite dans la figure 4.22.

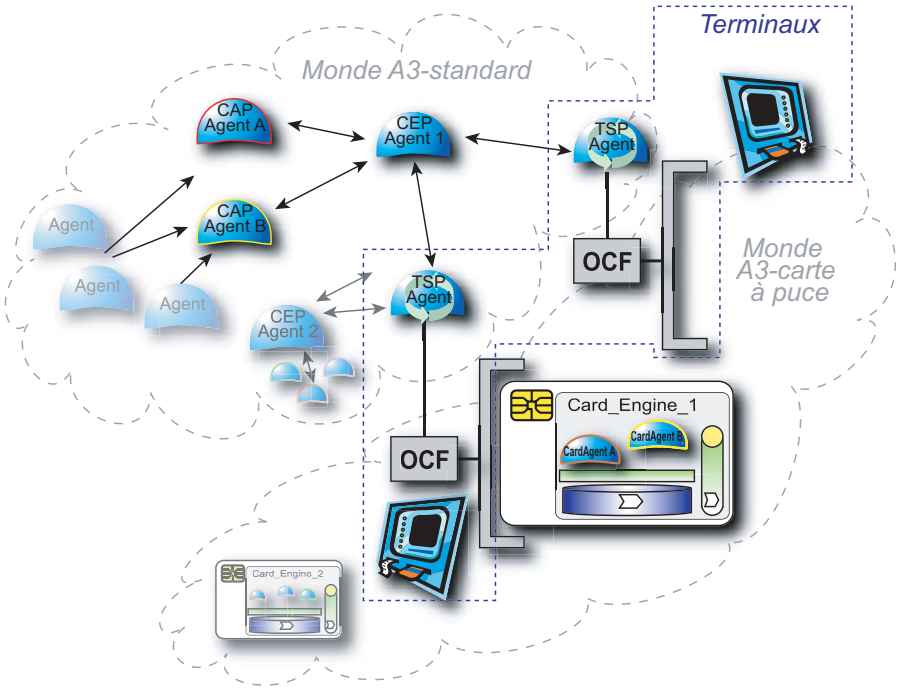

Figure 4.22 – Architecture A3/JavaCard.

4.3.3.1 Architecture sur les terminaux

L'architecture de l'intergiciel sur terminal permet de faire communiquer des agents entre les deux mondes (carte à puce et site d'exécution normal) et assure le respect des propriétés A3 (localisation, persistance, ordonnancement, ...) des agents sur carte.

La principale propriété à respecter est la localisation universelle des agents même en cas de mobilité de la carte de terminal en terminal. Ce problème a été contourné par l'utilisation d'un « agent proxy » appelé `TSPA` (*Terminal Side Proxy Agent*) présent sur chaque terminal où la carte peut se brancher. Cet agent est responsable de la liaison entre l'intergiciel présent sur la carte à puce et le « monde A3 » (≈ l'ensemble des autres sites) et permet de connaître le terminal sur lequel la carte est connectée. Les échanges de messages entre la carte et le monde A3 se font grâce à l'environnement JavaCard étendu fournissant la sérialisation/désérialisation. L'intergiciel sur carte à puce intègre donc un sérialiseur/désérialiseur de messages et son dual est présent sous la forme d'un agent dans le monde A3 appelé `CEPA` (*Card Engine Proxy Agent*). Le CEPA est donc le représentant dans le monde A3 de l'intergiciel sur la carte.

Tout message venant du monde A3 et destiné à un agent sur carte à puce est envoyé au CEPA qui le sérialise et envoie le résultat sous la forme d'un tableau d'octets au TSPA auquel la carte est branchée.

Le TSPA déclenche ensuite la carte (par les interface *Write* et *Read*) en envoyant le message à la couche intergiciel présente sur la carte qui fera réagir l'agent destinataire. Éventuellement, le TSPA récupère un ou des messages à renvoyer sous la forme d'un tableau d'octets et le transmet au CEPA qui les désérialisera et les enverra aux agents destinataires. Le CEPA joue donc le rôle d'un routeur de notifications de la carte vers les agents hors carte et vice versa. Afin de garantir la transparence de la localisation des agents, chaque agent de la carte possède un représentant dans le monde A3. Cet agent « miroir » (appelé `CAPA` pour *Card-Agent Proxy Agent*) reçoit les messages et les envoie au CEPA qui se chargera ensuite de les transmettre au TSPA.

Tous ces échanges de messages nécessitent une gestion de flux entre la couche intergiciel A3 présente sur la carte et le CEPA représentant de l'intergiciel. Cette gestion de flux (ainsi que le respect de l'ordonnancement des messages dont elle dépend) entre le monde carte et le monde A3 est décrite en détail dans le brevet [41], elle permet de gérer les déconnexions (impromptues) et de garantir la propriété de fiabilité des queues de messages.

4.3.3.2 Architecture sur la carte

L'intergiciel sur la carte à puce utilise l'API Javacard étendue, il s'agit d'une version entièrement modifiée de l'implémentation standard d'un serveur d'agent A3. Le serveur est piloté par les envois d'APDU du TSPA[8]. La réception de notifications par l'Engine fait réagir les agents destinataires qui posent à leur tour des notifications dans la queue de messages de sortie. Une fois les réactions terminées, le TSPA reçoit en retour les notifications à envoyer aux autres agents et les transmet au CEPA correspondant.

4.3.4 Bilan

L'apparition de l'API JavaCard a créé une révolution dans le monde des *SmartCards* et a transformé les cartes à puce en acteurs majeurs de l'informatique mobile. Néanmoins la limitation de l'environnement (pas de désallocation, pas de ramasse-miettes, ...) et le mode déconnecté quasi permanent ne facilitent pas leur intégration. C'est pourquoi nous avons utilisé une approche orientée sur l'utilisation d'un intergiciel asynchrone permettant l'amélioration de l'interopérabilité.

L'utilisation de l'intergiciel A3 a permis de réaliser le portage sur carte à puce tout en conservant les propriétés de cet intergiciel. Cependant, ce portage s'est fait au prix d'une architecture très compliquée mêlant les agents applicatifs pour le fonctionnement de l'intergiciel. Il a n'a pas été possible de prendre des briques existantes de l'intergiciel, alors même que la plupart des fonctionnalités du serveur sur carte sont identiques aux serveurs standards. La liaison entre l'intergiciel sur carte à puce et l'ensemble des autres sites doit se faire par l'intermédiaire d'un agent (un client applicatif) au lieu de passer par des couches de communication de l'intergiciel lui-même et cela à cause de la structure monolithique de celui-ci, ne permettant pas d'aller modifier son comportement et son architecture interne. De plus, chaque carte à puce nécessite un représentant sous la forme d'un agent rendant l'architecture très difficilement scalable.

Les besoins de configurabilité et d'adaptabilité exprimés par cette expérience confirment les constatations des sections précédentes et nous poussent à nous tourner vers la modélisation d'un intergiciel asynchrone adaptable.

[8] Tous les échanges passent par la couche OCF (*Open Card Framework*) qui implémente les « communications matérielles » entre le terminal et la carte.

4.4 Vers un intergiciel asynchrone adaptable

Nous avons vu dans les chapitres précédents que l'implémentation des intergiciels asynchrones actuels reste figée quels que soient les sites d'exécution et l'application (ou les applications) qui l'utilisent. C'est-à-dire que l'ensemble des propriétés non fonctionnelles fournies par l'intergiciel (la persistance des composants applicatifs ou des messages, l'ordonnancement des messages, etc.) sont inclues dans la couche intergicielle et sont donc fournies sur chaque nœud du système distribué. Toutefois, selon l'environnement d'exécution (système embarqué à ressources limitées par exemple) et du fait de contraintes de performances de l'application, il n'est pas toujours possible ni même souhaitable que l'intergiciel responsable de l'exécution et de la communication des composants assure systématiquement toutes les propriétés non fonctionnelles sur chaque nœud.

Nos expériences sur la configuration des intergiciels autour de propriétés comme l'ordonnancement causal ou le déploiement sur un environnement à grande échelle très spécialisé comme les cartes à puce, ont mis en avant les nombreux problèmes liés à l'aspect monolithique des intergiciels asynchrones.

Comme le dit Gul Agha dans [3], les intergiciels d'aujourd'hui doivent être suffisamment flexibles pour permettre leur adaptation aux changements, aussi bien du système sous-jacent que des besoins des applications. Les sections suivantes illustrent ce besoin en termes de configuration, de configurabilité et de mécanismes de contrôle.

4.4.1 Nécessité de configuration

Afin d'illustrer la nécessité de configuration d'un intergiciel, nous allons prendre l'exemple d'une application d'observation d'un système hétérogène distribué à grande échelle (*Grid Computing monitoring*).

Exemple De nombreux modèles d'application d'observation existent. Dans celui proposé par [62], l'observation est basée sur le découpage en quatre étapes (génération, traitement, dissémination, présentation). L'observation consiste donc à récupérer sur un site de maintenance (ou plusieurs sites de maintenance) les informations sur le fonctionnement (panne de machine, charge processeur, arrivée d'une nouvelle machine, ...) de plusieurs ensembles de machines sur des sites distants (voir figure 4.23). Ces informations brutes d'observation sont ensuite traitées (fusionnées, divisées, agrégées) sur des sites locaux puis acheminées à travers le réseau jusqu'au(x) site(s) de maintenance chargé(s) de présenter ces informations à l'administrateur humain.

Les ensembles de machines peuvent être aussi divers qu'une grappe d'une centaine de PC, un super calculateur, des terminaux de cartes à puce, etc. On souhaiterait, en outre, informer localement un technicien sur son assistant personnel d'une éventuelle panne afin qu'il prenne les décisions adéquates. Cette diversité des sites touche non seulement les machines à observer mais aussi les machines supportant les composants applicatifs de traitement (agrégation des données sur les sites locaux ou régionaux) ou de présentation des données (sites de maintenance, assistant personnel d'alerte, ...).

L'emploi d'un intergiciel responsable de la dissémination des informations est certainement la solution la plus apte à répondre aux besoins de passage à l'échelle, de flexibilité, et d'hétérogénéité d'une telle application d'observation. Cependant, les intergiciels synchrones ne sont pas adaptés à ces besoins. En effet, il serait irréaliste de gérer des millions de connexions client-serveur entre chaque machine d'un bout à l'autre du monde. L'utilisation d'un intergiciel asynchrone est actuellement reconnue comme étant une réponse à ces besoins croisés de passage à grande échelle, de flexibilité et d'extensibilité.

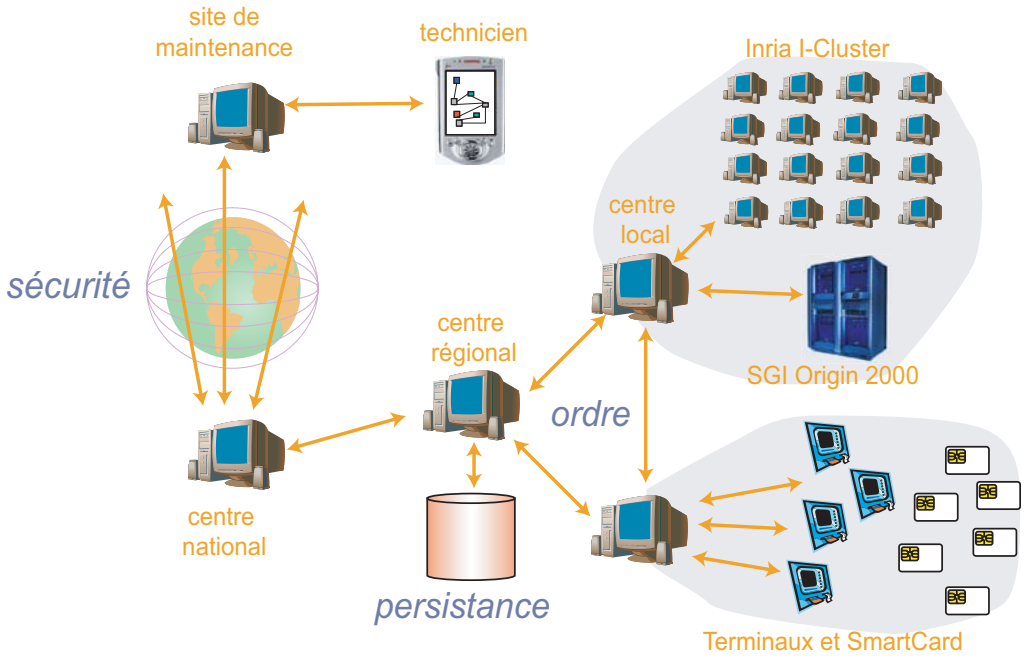

Figure 4.23 – Exemple d'application nécessitant une configuration de l'intergiciel.

Problèmes rencontrés Néanmoins la diversité des équipements et des infrastructures (avec des ressources et des fonctionnements hétérogènes) et les changements dynamiques du système à observer (déconnexion-reconnexion pour cause de mobilité, réduction de la bande passante sur des réseaux sans fil, etc.) posent de nombreux problèmes d'utilisation de l'intergiciel. Celui-ci doit être capable de s'exécuter de façon complètement indépendante des caractéristiques du site qui l'héberge. Cependant, il ne doit pas forcément fournir les mêmes propriétés sur l'ensemble des sites. Pourquoi, en effet, imposer des communications sécurisées entre les sites faisant partie d'un même réseau local (au risque de surcharger l'exécution) alors que cette sécurité n'est requise qu'entre deux sites hors du réseau local ? De la même manière pourquoi imposer un ordonnancement causal des messages à l'ensemble des sites alors qu'aucune cohérence n'est nécessaire au niveau applicatif ? La nécessité de configuration de l'intergiciel est donc d'autant plus forte que les sites d'exécution sont hétérogènes et a fortiori, lorsque les besoins applicatifs sont différents d'un site à l'autre.

4.4.2 Besoin de configurabilité

Comme nous l'avons vu, l'aspect monolithique des implémentations actuelles d'intergiciels asynchrones ne permet pas de répondre à ce besoin de configuration. La structure interne de chaque intergiciel reste figée et il n'est pas possible de la modifier à moins d'en réécrire le code ! L'utilisation d'un intergiciel modulaire devient donc nécessaire afin de pouvoir facilement configurer sa structure interne. Cette modularité peut se résumer en deux points :

- offrir la possibilité d'être exécutée de façon minimale (un intergiciel offrant un comportement fonctionnel de base sans aucune propriété non fonctionnelle) ;
- pouvoir accueillir (statiquement ou dynamiquement à l'exécution) de nouveaux comportements fonctionnels ou responsables de la gestion de propriétés non fonctionnelles.

Ces deux points rejoignent les architectures des ORB réflexifs comme DynamicTAO (voir section 2.2.2.2) qui possèdent un noyau minimal s'exécutant de façon continue permettant la mise à jour dynamique des composants.

Architecture à composants Nos besoins de modularité et de configurabilité nous mènent tout naturellement à nous tourner vers l'utilisation d'architecture à composants. Contrairement aux plates-formes actuelles, l'intergiciel que nous construirons n'aura pas nécessairement la même composition sur les différents sites physiques de l'application. On peut donc le qualifier d'hétérogène. La notion d'hétérogénéité est illustrée sur la figure 4.24. On voit que la composition de l'intergiciel n'est pas forcément la même sur les différents nœuds : les différents sites hébergeant l'application sont composés d'assemblages de composants différents. Chaque site fournit donc des propriétés spécifiques à son environnement d'exécution et à ces besoins applicatifs.

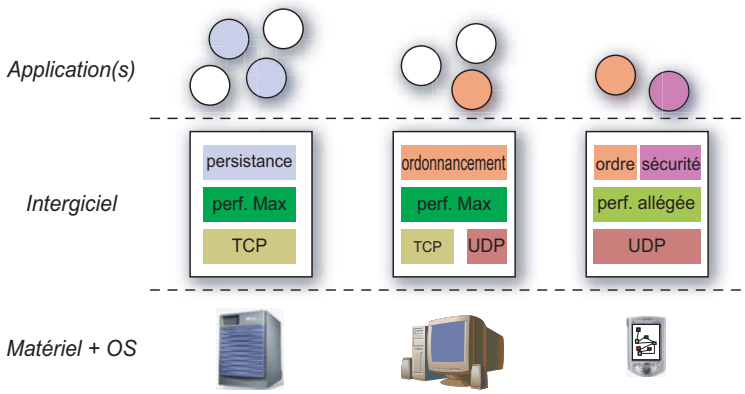

Figure 4.24 – Une composition d'intergiciel hétérogène.

De plus, les tendances émergentes de l'informatique omniprésente impliquent de nouveaux besoins en terme de configuration dynamique, d'adaptabilité et de déploiement des intergiciels.

4.4.3 Déploiement

Le déploiement d'une application est une opération qui consiste à installer les composants d'une application et à lancer son exécution. Les nombreux travaux menés autour du déploiement se sont trop souvent focalisés autour du déploiement d'applications réparties ([73]). Le déploiement de l'infrastructure d'exécution de l'application, en l'occurrence l'intergiciel, n'a que très rarement été abordé. Pourtant, il nous semble essentiel qu'un travail soit mené autour du processus de déploiement de l'intergiciel car il est trop réducteur de dire que l'application va pouvoir se déployer sur l'intergiciel si celui-ci n'a pas pu être correctement installé et démarré.

Revenons sur l'exemple d'application d'observation de système distribué hétérogène pour soulever le problème du déploiement de l'intergiciel. Dans ce cas il serait très intéressant de pouvoir déployer l'intergiciel de telle façon que l'application puisse démarrer même si une partie de l'infrastructure n'est pas disponible. En effet, une telle application ne requiert pas forcément que l'ensemble des sites soient actifs pour démarrer. Si une machine n'est pas disponible lors de l'installation de l'infrastructure, il faut pouvoir éviter une attente des autres sites. De plus, il serait nécessaire d'informer le concepteur de l'application d'un éventuel problème afin qu'il prenne les décisions appropriées (changement de la structure de l'intergiciel ou de l'application, déploiement de l'intergiciel sur un autre site,...).

Des primitives locales permettant une installation et une vérification de configuration sont donc nécessaires au sein même de l'intergiciel asynchrone. Ces primitives dites de « contrôle » doivent faire partie intégrante de l'intergiciel alors que le processus de déploiement est une tâche à part pouvant être intégrée par la suite à l'intergiciel.

4.4.4 Synthèse

Dans les sections précédentes nous avons vu plusieurs exemples de propriétés et d'infrastructures qui nous ont permis d'identifier trois besoins pour les intergiciels asynchrones. Ces besoins sont prônés par les propriétés applicatives (ordonnancement, persistance, routage, etc.), par les propriétés du système (topologie, protocole, matériel, etc.) et sur la combinaison des deux (déploiement dirigé par l'application et le système, personnalisation de l'intergiciel en fonction des applications, etc.). Ces trois besoins permettent aux intergiciels asynchrones de répondre aux contraintes d'utilisation dans les environnements distribués hétérogènes à grande échelle :

- Le besoin de configuration statique mais aussi dynamique permettant une adaptation de l'intergiciel à son environnement en cours d'exécution.
- Le besoin de configurabilité qui découle de la nécessité de configuration.
- Le besoin de mécanismes de contrôle facilitant la configuration et le déploiement de l'intergiciel sur les différents sites d'exécution.

De ces besoins résulte une exigence de minimalité de l'intergiciel. Les fonctionnalités offertes par l'intergiciel doivent être minimales dans le but évident de ne pas imposer de propriétés pénalisant les utilisations futures.

Un intergiciel asynchrone doit donc offrir un comportement fonctionnel de base sur lequel peuvent se greffer des extensions (fonctionnelles ou non fonctionnelles). De nombreux travaux autour des systèmes d'exploitation ont mené au paradigme des noyaux extensibles ([19, 83]). En reprenant cette notion, nous voulons que cette extension (ou configuration) puisse être réalisée de façon statique au démarrage de l'intergiciel ou dynamiquement. Par configuration dynamique nous entendons la faculté d'introduire des modifications dans l'intergiciel à l'exécution afin de l'adapter à son environnement. Les nombreux travaux autour de l'adaptabilité des ORB [22, 55] mettent en avant l'utilisation de la notion de réflexivité qui permet à un système de maintenir et d'utiliser une représentation de lui-même (voir chapitre 2 de l'état de l'art). Nous proposons donc un modèle d'intergiciel asynchrone utilisant des techniques de réflexivité associées à une architecture à composant qui sont les plus à même à répondre au besoin d'adaptation. De plus, nous proposons des mécanismes de configuration au niveau des composants de l'intergiciel permettant de contrôler systématiquement son déploiement.

Nous nous attarderons à proposer dans le chapitre suivant un modèle d'intergiciel asynchrone qui essaye de répondre à ces besoins d'adaptabilité.

Le chapitre suivant est consacré à la description de notre modèle d'intergiciel asynchrone adaptable : DREAM (***D**ynamic **RE**flective **A**synchronous **M**iddleware*).

Chapitre 5

DREAM : Un intergiciel asynchrone adaptable

5.1 Introduction

Les problèmes soulevés dans le chapitre précédent nous ont permis de dégager les besoins de configuration statique et dynamique, et de mécanismes de déploiement pour les intergiciels asynchrones de nouvelle génération.

Nous présentons dans ce chapitre, DREAM (***D**ynamic **RE**flective **A**synchronous **M**iddleware*) un intergiciel asynchrone adaptable. Cet intergiciel associe les nombreux travaux dans le domaine de l'asynchrone ([4, 78, 77, 70, 44],...) qui ont mené à la définition des modèles de communication asynchrones (*message queuing, publish/subscribe, ...*) et les réflexions menées dans les intergiciels synchrones autour des nouveaux besoins de configuration et d'adaptabilité ([22, 55, 53],...).

Notre architecture d'intergiciel asynchrone propose des mécanismes de configuration et de déploiement pouvant être interfacés avec des outils responsables de « l'intelligence » de la configuration et du déploiement. La suite de ce chapitre décrit en détail cette plate-forme asynchrone et ces mécanismes.

5.2 Architecture de DREAM

5.2.1 Modèle

L'état de l'art sur les intergiciels asynchrones dressé dans le chapitre 1 nous a permis d'identifier l'architecture d'un intergiciel asynchrone la plus adaptée aux besoins de passage à grande échelle, de flexibilité et d'extensibilité. Il se dégage clairement deux parties bien distinctes : une partie responsable de la délivrance des messages et une partie en charge du traitement des messages. La partie responsable de la délivrance fournit un *modèle de délivrance* définissant les principes fondamentaux du transfert des messages entre les différents sites. La partie en charge du traitement des messages définit la manière dont les messages vont être manipulés avant (ou après) avoir été transférés. Ce *modèle de traitement* des messages définit la sémantique de l'application, il spécialise le modèle de délivrance en fonction des applications. Il y a donc un modèle de délivrance commun à tous et un modèle de traitement spécifique à chaque application (voir figure 5.1).

Cette séparation est essentielle car elle propose d'utiliser un même modèle de délivrance quel que soit le traitement des messages. Il est important de noter qu'un même modèle ne signifie pas forcément les même propriétés sur les messages ou un même protocole d'envoi pour tous, ni même

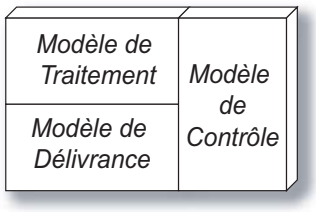

Figure 5.1 – Les modèles d'architecture de l'intergiciel asynchrone

une architecture de communication identique (nous verrons plus en détail ces aspects dans la section 5.3). De la même manière le modèle de traitement est spécifique à chaque application. L'utilisation d'un modèle de délivrance commun permet une meilleur interopérabilité des applications. Nous ne voulons pas, par cette architecture, recréer un standard de communication asynchrone déjà existant (l'API JMS notamment). Au contraire, le modèle proposé est plus général que les modèles actuels et surtout moins spécifique à tel ou tel type d'applications ou de plates-formes.

Afin de répondre aux besoins de contrôle exposés dans les chapitres précédents, nous proposons d'ajouter aux deux modèles précédents, un modèle transversal appelé *modèle de contrôle*. Ce modèle est commun au modèle de délivrance et au modèle de traitement. Il définit la manière de contrôler la délivrance et le traitement des messages et fournit des mécanismes permettant la configuration et de déploiement de l'intergiciel.

5.2.2 Concrétisation

Sur chaque site d'exécution, l'architecture de DREAM se décompose en trois entités : un composant MOM en charge de transférer des messages sur le réseau (*modèle de délivrance*), des composants services (*modèle de traitement*) qui produisent et consomment des messages à travers le MOM et un composant local d'administration et de déploiement (LA) responsable de la configuration, du déploiement et de l'administration (≈*modèle de contrôle*) (voir figure 5.2). L'intergiciel asynchrone DREAM est représenté par un composant composite pouvant être composé du MOM et de plusieurs services. Le LA est quant à lui, une entité externe qui peut configurer et déployer l'intergiciel DREAM (nous verrons dans la suite qu'il peut être utilisé indépendamment de DREAM).

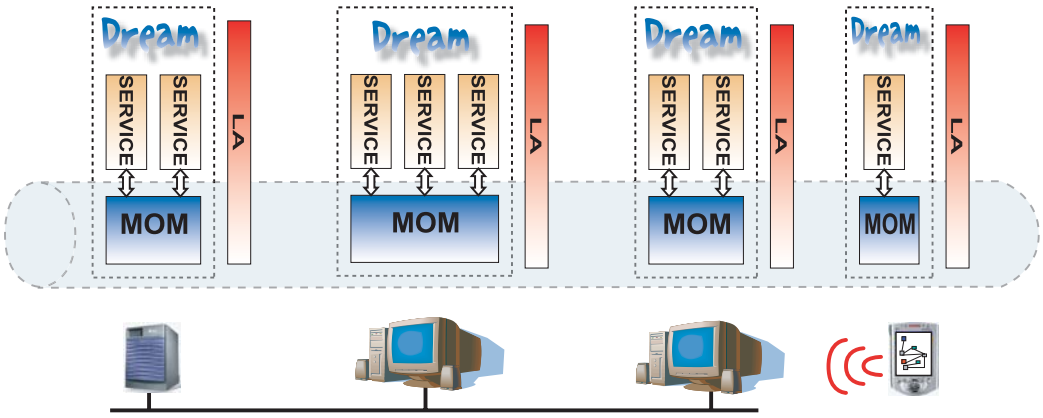

Figure 5.2 – L'architecture de DREAM.

La plate-forme DREAM possède à tout moment sur un site, au moins le composant d'administra-

tion et le composite englobant le MOM et les services. C'est ce que nous appelons l'invariant de la plate-forme. Le composite pouvant être composé du MOM et de services ou bien du MOM seul ou encore de service(s) seul(s). On peut en effet imaginer un service unique qui s'exécute de façon autonome (provisoirement de façon déconnectée). Dans ce cas il n'y aura pas de composant MOM sur la plate-forme mais seulement un service et le composant d'administration chargé de le configurer, déployer et administrer (voir figure 5.3).

Service Un service est un support de traitement de messages, une application s'exécute donc à travers un ou plusieurs services qui communiquent grâce au MOM. Chaque service requiert une interface d'envoi de messages et fournit une interface de réception de messages. C'est uniquement grâce à ces interfaces que les services peuvent envoyer des messages à travers le MOM. Néanmoins rien n'empêche à un service de proposer des interfaces externes pour communiquer vers l'extérieur pour des raisons fonctionnelles (voir section 5.4).

MOM Le MOM est responsable du transfert des messages entre les différentes plates-formes distribuées. Un service qui désire envoyer un message (vers un autre service), passe le message au MOM en désignant le service destinataire, puis le message est acheminé à travers le réseau au MOM lié au service destinataire. Le MOM est responsable de la transmission des messages envoyés par les services mais (par défaut) ne garantit aucune propriété sur ces messages. Néanmoins, comme nous le verrons dans la suite les possibilités de configuration de DREAM permettent de configurer le MOM afin de supporter diverses propriétés non-fonctionnelles.

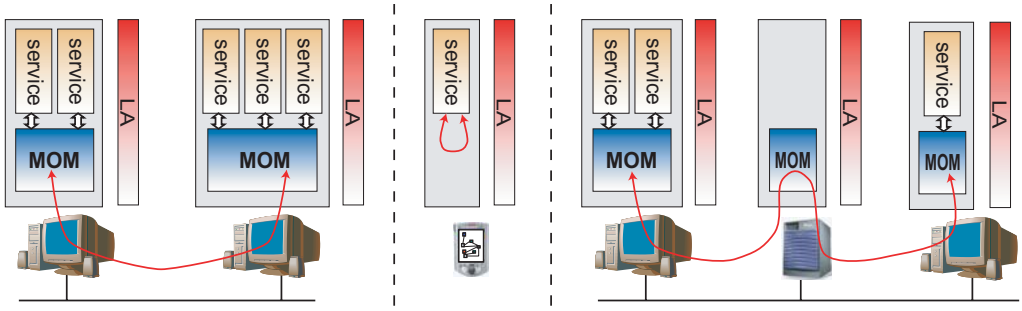

Figure 5.3 – Différentes architectures possibles de DREAM.

LA Le composant d'administration est une entité pouvant être utilisée indépendamment de DREAM. L'ensemble des LA est une application à part entière responsable du déploiement et de la configuration d'intergiciels pouvant être interfacés avec diverses plates-formes (voir section 5.5). Néanmoins, le composant local d'administration est présent quelle que soit la configuration de l'intergiciel. Il utilise les mécanismes de contrôles inclus dans le MOM et les services pour configurer l'intergiciel. Ces mécanismes de contrôles suivent tous le même *modèle de contrôle*.

5.2.3 Architecture à composants

Chaque entité de DREAM est modélisée par un composant Fractal ([27]). L'architecture de composition Fractal est bien adaptée aux besoins de DREAM car elle prend en compte, d'une part, l'aspect composition (qui permet la construction d'un intergiciel par un ensemble structuré de composants) et, d'autre part, l'aspect dynamique (qui facilite l'adaptation de l'intergiciel aux changements du système

et de l'application). De plus, les propriétés de réflexivité fournies par l'architecture Fractal (en particulier, l'accès à des meta-informations permettant de manipuler dynamiquement la structure et le comportement des composants) facilitent la reconfiguration dynamique des composants.

Pourquoi Fractal ? Les architectures à composants industrielles telles que les EJB de Sun [37], San Francisco d'IBM [6], le modèle à composants Corba de l'OMG (CCM) [68] et COM de Microsoft [67] permettent la gestion de composants industriels (business components) à travers des conteneurs qui fournissent automatiquement les services de propriétés non fonctionnelles telle que la persistance, la sécurité, les transactions, etc. Cette approche a l'avantage d'apporter une solution « tout-en-un » mais souffre d'un inconvénient majeur : le concept de configuration est embryonnaire voire simplement absent. De plus, leur modèle est dépourvu de notion de composition et toute structure est décrite à plat. En résumé, les architectures à composants industriels sont très utiles pour la gestion de composants industriels mais atteignent vite leur limite lors de leur utilisation pour la construction de systèmes complexes nécessitant des besoins de composition, de configuration et d'adaptation dynamique.

C'est pourquoi l'approche de Fractal est la plus apte à répondre à nos besoins car elle correspond exactement au modèle à composants nécessaire pour l'architecture de DREAM. Notre modèle de contrôle utilise donc celui de Fractal basé sur la réflexivité. Nous avons introduit, dans la section 2.2.1.4, Julia une implémentation de Fractal, utilisée pour le développement de DREAM. Nous présentons donc dans la section suivante le langage de description d'architecture que nous avons utilisé pour la description de DREAM.

5.2.4 MDL (*Middleware Description Language*)

Dans le chapitre 3, nous avons présenté différents ADL et nous avons vu les limites des approches actuelles. Nous proposons un ADL (étendu de celui de Julia) qui, associé au contrôleur de configuration, permet de lever les limites des ADL actuels.

L'implémentation actuelle de Julia définit un ADL relativement simple basé sur des descripteurs XML, et fournit un outil pour parcourir et analyser une configuration de composants décrite avec cet ADL. Cet outil crée automatiquement le patron de composants associé à la configuration décrite et permet par la suite de l'instancier. Nous avons modifié cet ADL afin d'obtenir plus de fonctionnalités. Les trois premiers éléments (spécification des type de composant, des patrons de composant et de la structure des composites) sont quasiment identiques à ceux fournis par l'ADL de Julia. Nous avons rajouté trois notions importantes : la définition de l'initialisation d'un composant (section 5.2.4.4), la définition du cycle de vie d'un composant (section 5.2.4.5) et surtout la définition d'une configuration d'intergiciel **distribué** (section 5.6.1.1).

Nous avons appelé notre langage de description de l'intergiciel un MDL (pour *Middleware Description Language*). l'utilisation d'un MDL est motivé par deux objectifs :

- Premièrement, simplifier la création de la configuration de l'intergiciel. Il est, en effet, tout à fait possible d'écrire la configuration de l'intergiciel « à la main » en écrivant en dur dans le code la description de chaque composant (leur type, leurs attributs, etc.). Néanmoins, cette description serait longue et fastidieuse à écrire, de plus elle serait peu réutilisable et surtout difficilement modifiable à l'exécution (dans le cas d'une reconfiguration). L'utilisation d'une description apporte une vision globale de la configuration de l'intergiciel et permet de spécifier son architecture à partir d'une bibliothèque de composants ;
- Deuxièmement, faciliter le déploiement de la configuration. Lors du déploiement d'une configuration, le MDL est transmis au LA qui se charge de déployer localement la description

donnée par le MDL. Pour cela le MDL définit plusieurs éléments qui permettent au LA de connaître la structure à instancier et les différentes actions à réaliser comme les liaisons entre composants locaux, l'initialisation des composants, etc. De plus le MDL permet de décrire la structure globale d'une configuration distribuée. Ces points sont décrits dans les sections5.5 et 5.6.

Le MDL est un langage de description d'architecture basé sur une description XML qui définit de manière précise les composants qui composent l'intergiciel. la description est réalisé grâce à un fichier permettant de spécifier en détail l'architecture de l'intergiciel et son comportement. Ce fichier est analysé par un analyseur spécifiquement implémenté pour DREAM mais qui utilise l'analyseur fournit par Julia pour la description des types de composants et des patrons de primitifs. L'analyseur remplit une structure complexe avec tous les éléments et les paramètres qui sera utilisée par la suite au moment de créer puis d'initialiser l'intergiciel ainsi défini. L'analyseur de Julia est donc basé sur deux éléments :

- Les types de composants qui définissent les interfaces fonctionnelles fournies et requises pour ce type ;
- Les patrons de primitifs qui définissent le nom de la classe Java et le type que cette classe implémente ;

L'analyseur d'un fichier en MDL de DREAM est basé sur trois éléments :

- La structure du composite qui définit un ensemble de sous composants, un ensemble de liaisons entre ces sous composants (pouvant être eux-aussi des composites), le type que cet assemblage implémente et la description des contrôleurs associés ;
- Les paramètres d'initialisation de l'architecture décrite qui définissent l'utilisation des méthodes d'initialisation ;
- La gestion du cycle de vie des composants et plus spécifiquement leur comportement d'arrêt.

L'intergiciel est défini par un composite englobant une configuration de composants. Sa description locale à un site est donc toujours de la forme suivante (nous verrons dans la suite la description globale) :

```
<composite name="..." type="...">
  <struct>
    <components>
    ...
    <bindings>
    ...
    <controller>
    ...
  </struct>
  <inits>
    ...
  </inits>
  <stop-policy type="..." />
</composite>
```

L'attribut `name` définit le nom du composant composite et l'attribut `type` donne le type de composant (décrit par un `<component-type>`) que ce composite implémente. Les autres éléments sont décrits plus en détail dans la suite.

5.2.4.1 Spécification des types de composant

Un type de composant est spécifié à l'aide de l'élément `<component-type>`. Il spécifie les interfaces fonctionnelles qui sont fournies et requises par les composants de ce type. Mais il ne spécifie pas les interfaces de contrôles des composants de ce type. Les interfaces étant spécifiques à chaque composant, deux composants d'un même type peuvent avoir des interfaces de contrôles différentes. Ces dernières sont donc associées à un patron de composant et décrites dans le fichier de configuration `julia.cfg` (voir section 2.2.1.4).

La forme générale d'un élément `<component-type>` est la suivante :

```
<component-type name="..." extends="...">
  <provides>
    <interface-type name="..." signature="..." contingency="..."
      cardinality="..."/>
    ...
  </provides>
  <requires>
    <interface-type name="..." signature="..." contingency="..."
      cardinality="..."/>
    ...
  </requires>
</component-type>
```

Les attributs `extends`, `signature`, `contingency` et `cardinality` sont optionnels, tout comme les sous éléments `<provides>` et `<requires>`. Lorsqu'une interface est fournie (`<provides>`) c'est une interface serveur (qui recevra des appels de méthode), alors qu'une interface requise (`<requires>`) est une interface cliente (qui réalisera des appels de méthode). L'attribut `extends` permet d'étendre un type de composant préalablement défini ; dans ce cas, il n'est nécessaire de définir que les interfaces (fournies ou requises) supplémentaires à celles du type étendu. L'attribut `signature` définit le nom de la classe Java implémentant cette interface ; il peut être omis uniquement lorsque le type interface est déjà défini[1]. L'attribut `contingency` définit l'utilisation de l'interface mais différemment selon qu'il s'agisse d'une interface client ou serveur. S'il s'agit d'une interface serveur et que la valeur de l'attribut est égale à `mandatory` (valeur par défaut) le composant doit implémenter cette interface, si il est égal à `optional` le composant n'est pas obligé d'implémenter cette interface. S'il s'agit d'une interface cliente et que la valeur de l'attribut est égale à `mandatory` (valeur par défaut) alors l'interface doit être liée à l'exécution, si elle est égale à `optional` l'interface n'a pas à être forcément liée à l'éxécution. La cardinalité de l'interface (c'est-à-dire son comportement de liaison) est exprimée par l'attribut `cardinality`. Si sa valeur est égale à `single` (valeur par défaut) alors l'interface de pourra être liée qu'à une seule autre, si elle est égale à `collection` elle pourra être liée à plusieurs interfaces.

Prenons par exemple la description du composant `DispatcherOut`[2], un composant du MOM, qui transmet les messages venant des services vers les différentes liaisons réseaux. Ce composant fournit une interface permettant aux services de lui passer les messages à envoyer (interface `inPush`) et requiert une interface (interface `outPush`) pour transmettre les messages aux différentes « pattes » réseaux. Plus précisément l'interface `inPush` est une interface serveur qui est obligatoirement implémentée par le composant (`contingency="mandatory"`) et qui n'est liée qu'a une seule interface cliente (`cardinality="single"`), elle est définie par la classe Java `fr.inrialpes.sardes.dream.common.Push`. L'interface `outPush` est une interface

[1] La sémantique des mécanismes d'héritage du MDL sont les mêmes que ceux de l'ADL Fractal. Ces derniers sont disponibles dans [27].

[2] Une description détaillée de ce composant du MOM est donnée dans la section 5.3.

cliente qui n'est pas forcément liée à l'exécution (`contingency="optional"`) mais qui peut être liée à plusieurs interfaces serveurs (`cardinality="collection"`).

```
<component-type name="fr.inrialpes.sardes.dream.mom.DispatcherOutType">
  <provides>
    <interface-type name="inPush"
      signature="fr.inrialpes.sardes.dream.common.Push"
      contingency="mandatory" cardinality="single" />
  </provides>
  <requires>
    <interface-type name="outPush"
      signature="fr.inrialpes.sardes.dream.common.Push"
      contingency="optional" cardinality="collection" />
  </requires>
</component-type>
```

5.2.4.2 Spécification des patrons de composants primitifs

La deuxième étape, après la spécification des types de composants, consiste à décrire les patrons des composants primitifs. Un patron de composant primitif est spécifié à l'aide de l'élément `<primitive-template>`. Il spécifie le nom de la classe Java du composant et le type de composant (décrit par un `<component-type>`) que ce composant implémente.

La forme générale d'un élément `<primitive-template>` est la suivante :

```
<primitive-template name="..." implements="..." extends="...">
  <primitive-content class="..."/>
  <controller>
    <attributes signature="...">
      <attribute name="..." value="..."/>
      ...
    </attributes>
    <template-controller desc="..."/>
    <component-controller desc="..."/>
  </controller>
</primitive-template>
```

L'attribut `name` définit le nom attribué à ce patron. lL'attribut `implements` donne le nom du type de composant que ce patron implémente. L'attribut `extends` permet d'étendre un patron de composant primitif préalablement défini ; dans ce cas, il n'est pas nécessaire de définir l'attribut `implements`. L'élément `<primitive-content>` et son attribut `class` permettent de donner le nom de la classe qui instancie le composant correspondant à ce patron. L'élément `<controller>` comporte trois sous-éléments : `<attributes>`, `<template-controller>` et `<component-controller>`. L'élément `<template-controller>` (resp. `<component-controller>`) peut être utilisé pour spécifier le descripteur de contrôleur du patron de composant (resp. du composant qu'il instancie). Un élément spécial permet de définir la valeur d'attribut du composant, c'est l'élément `<attributes>`. Il spécifie le contrôleur d'attribut (grâce à l'attribut `signature`) et les différents attributs du composant à initialiser à une certaine valeur lors de la création (grâce à l'élément `<attribute>` et ces attributs `<name>` et `<value>`).

Les attributs `implements`, `extends` et `signature` sont optionnels, tout comme les sous-éléments `<primitive-content>`, `<controller>`, `<attributes>`, `<template-controller>` et `<component-controller>`.

Prenons par exemple, la description du patron de composant `DispatcherInTmpl`[3], un composant du MOM, qui transmet les messages venant des différentes liaisons réseaux vers les services.

```
<primitive-template
    name="fr.inrialpes.sardes.dream.mom.DispatcherInTmpl"
    implements="fr.inrialpes.sardes.dream.mom.DispatcherInType">
  <primitive-content
    class="fr.inrialpes.sardes.dream.mom.DispatcherInImpl" />
  <controller>
    <component-controller desc="momPushCollectionPrimitive" />
  </controller>
</primitive-template>
```

Ce patron implémente le type de composant `DispatcherInType` et il instancie des composants ayant pour classe Java `DispatcherInImpl`. De plus Le composant instancié possède un ensemble de contrôleurs spécifiques spécifié par la description `momPushCollectionPrimitive` se trouvant dans le fichier `Julia.cfg`.

5.2.4.3 Spécification de la structure des composites

La troisième étape est la description même de l'intergiciel. Elle consiste à décrire le composant composite définissant la configuration de l'intergiciel. Le composite global peut être composé de sous-composants pouvant être eux-même des composites. On a donc une description récursive permettant de décrire n'importe quelle composition de composants.

Un composite est spécifié à l'aide de l'élément `<composite>`. Comme tout composant, un composite possède un nom (attribut `name`) et un type (attribut `type`). De plus, lorsqu'il s'agit du composite DREAM englobant l'ensemble d'une configuration de site, il possède un attribut `host` qui lui donne sa localisation (utilisé par le déploiement global détaillé plus loin).

L'élément `<composite>` utilise l'élément `<struct>` pour définir sa structure interne. Celui-ci comporte trois sous-éléments qui définissent :

- L'ensemble des sous-composants de ce composite (`<components>`);
- L'ensemble des liaisons entre ses sous-composants (`<bindings>`);
- La description des contrôleurs que ce composite implémente (`<controller>`).

La forme générale d'un élément `<struct>` est la suivante :

```
<struct>
    <components>
      ...
      <component ... />
      ...
      <composite name="..." type="..." >
         ...
      </composite>
      ...
    </components>
    <bindings>
      ...
      <binding ... />
      ...
    </bindings>
    <controller>
      <component-controller ... />
      <template-controller ... />
```

[3] Une description détaillée du composant associé (`DispatcherIn`) est donnée dans la section 5.3

```
    </controller>
</struct>
```

Elément <components> L'élément <components> spécifie l'ensemble des composants (sous-composants) que le composite englobe. Ces composants sont soit des composants primitifs, soit des composants composites. S'il s'agit de composants composites, la description est donnée par l'élément <composite> qui répète le même schéma que celui décrit dans cette section. S'il s'agit d'un composant primitif il utilise l'élément <component> pour donner sa description.

La forme générale d'un élément <component> est la suivante :

```
<component name="..." threaded="..." template="..." type="..." />
```

L'attribut name définit le nom du composant. L'attribut threaded peut prendre la valeur *true* ou *false*. Il permet de savoir si le composant est un composant gérant un processus léger (*thread*). Cette précision est très importante car un composant sans processus léger n'est pas manipulé de la même manière, notamment en ce qui concerne la gestion du cycle de vie comme nous le verrons dans la suite. L'attribut template définit le nom du patron de composant primitif que ce composant instancie. Ce nom doit avoir été préalablement défini par un élément <primitive-template>. L'attribut type donne le type que ce composant implémente. Il doit avoir été préalablement défini par un élément <component-type>.

Une fois tous les composants contenus dans le composite décrits, il faut définir les liaisons entre ces composants. C'est le rôle de l'élément <bindings>.

Elément <bindings> L'élément <bindings> spécifie l'ensemble des liaisons entre les composants du composite. Ces liaisons sont toujours réalisées d'une interface cliente vers une (ou plusieurs) interface serveur. Autrement dit, elles sont toujours réalisées d'une interface requise vers une (ou plusieurs) interface(s) fournie(s). Dans le cas d'une liaison d'une interface cliente vers plusieurs interfaces serveurs on parle de liaison vers des interfaces multiples ou de *collection d'interfaces*[4].

La forme générale d'un élément <bindings> est la suivante :

```
<bindings>
  ...
  <binding client="..." server="..." />
  ...
</bindings>
```

Un élément <bindings> comporte donc un ensemble de sous-éléments <binding> qui définissent chaque liaison entre les sous-composants primitifs et composites de ce composite. L'attribut client donne le nom de l'interface à lier à l'interface dont le nom est donné par l'attribut server. Chaque interface est spécifiée par le nom d'un sous-composant (ou le nom *this* pour désigner le composite lui-même), suivi d'un point et suivi du nom de l'interface de ce sous-composant.

Les liaisons décrites, il convient de définir la description des contrôleurs associés à ce composite, c'est le rôle de l'élément <controller>.

[4] Se reporter à la spécification de Fractal pour plus de détails : [27].

Elément `<controller>` L'élément `<controller>` donne la description des contrôleurs de ce composite. Il comporte un sous-élément `<component-controller>` qui a pour forme générale :

```
<component-controller desc="..." />
```

Cet élément donne la description des interfaces de contrôle de ce composant composite une fois instancié. La description, nommée par l'attribut `desc`, doit être définie dans le fichier de configuration de Julia. Cette description permet de définir de façon précise l'ensemble des interfaces de contrôle que le composant possède.

La section suivante explique la spécification des données d'initialisation des composants d'une configuration.

5.2.4.4 Spécification de l'initialisation de composants

L'initialisation d'un composant consiste à positionner la valeur de certains de ses attributs et à récupérer certaines de ces valeurs. On peut par exemple définir la taille d'une file de messages ou récupérer la valeur du port d'écoute TCP du composant réseau `NetworkIn` pour la donner à d'autres composants. L'initialisation est donc une étape essentielle de la configuration de l'intergiciel car elle permet de positionner les valeurs nécessaires au bon démarrage de l'intergiciel. Nous verrons dans la section 5.2.5 les détails de l'initialisation des composants à l'aide du contrôleur de configuration.

L'élément `<inits>` permet de spécifier, pour un composite donné, les ordres d'initialisation à effectuer pour ses sous-composants (primitifs ou composites). La forme générale d'un élément `<inits>` est la suivante :

```
<inits>
  <init-component component="...">
    <parameter name="..." value="..." />
    ...
  </init-component>
  ...
  <return name="..." value="..." />
</inits>
```

L'attribut `component` donne le nom du composant à initialiser, il s'agit du nom défini dans la spécification de la structure (voir plus haut). L'élément `<parameter>` est optionnel car il est tout à fait possible d'initialiser un composant (c'est-à-dire appeler la méthode `init()` sur ce composant) sans avoir à lui passer de paramètre. L'attribut `name` donne le nom du paramètre à initialiser (ce nom sera interprété dans la méthode `init()` du composant). L'attribut `value` donne la valeur du paramètre. Cette valeur n'est pas forcément fixe et peut être modifiée par le composant lui-même (si par exemple le port donné est déjà utilisé) ; nous verrons cela en détail dans la section suivante. L'attribut `value` peut être :

- Soit une valeur absolue, c'est-à-dire un nombre (le numéro du port d'écoute, la taille d'une file de messages, ...), une chaîne de caractères (le nom d'un service,...), etc ;
- Soit une valeur relative à un autre paramètre précédemment exprimé. Dans ce cas, la valeur est une chaîne de caractère donnant la référence du paramètre à prendre. La notation est du type *`NomComposant->nomParamètre`* où *NomComposant* définit un sous-composant du composite courant et *nomParamètre* définit le nom du paramètre à utiliser. Mais la valeur peut aussi faire référence à un paramètre du composite, dans ce cas la notation est du type `Composite->`*`nomParamètre`*. Un paramètre de composite est utile car il permet de définir

un paramètre dont la valeur pourra être partagée par plusieurs composants, ils n'auront qu'à utiliser la référence `Composite->`*`nomParamètre`* pour obtenir cette valeur. Dans le cadre d'une configuration globale, la valeur peut prendre une forme spécifique désignant « l'adresse d'un site » (voir section 5.6.1.1).

L'élément `<return>` permet au contrôleur de configuration de récupérer les valeurs de certains paramètres initialisés (ceux-ci pouvant être modifiés par le composant). Il définit le nom exporté (attribut `name`) et sa valeur (attribut `value`) qui est une référence vers un paramètre d'un sous-composant.

Il est important de noter que l'ordre dans lequel sont donnés ses éléments d'initialisation (les éléments `<init-component>`) est très important, car c'est l'ordre dans lequel les composants seront effectivement initialisés.

La section suivante décrit la spécification des données utilisées pour la gestion du cycle de vie des composants d'une configuration.

5.2.4.5 Spécification de la gestion du cycle de vie

La gestion du cycle de vie utilise deux informations, tout d'abord la nature du composant : composant passif ou actif (possédant un processus léger ou *thread*). Ce premier point est décrit dans la spécification de la structure du composant (voir plus haut). Ensuite l'ordre d'arrêt des composants spécifié par l'élément `<stop-policy>`. Nous verrons plus en détail l'utilité et l'utilisation de l'ordre d'arrêt des composants dans la section suivante, lors de la description du contrôleur de cycle de vie.

La forme générale d'un élément `<stop-policy>` est la suivante :

```
<stop-policy type="..." >
  <stop-component name="..." />
  ...
</stop-policy>
```

L'attribut `type` peut prendre trois valeurs : `automatic`, `custom` et `list`. Dans le premier cas, l'ordre d'arrêt des composants sera automatiquement donné par l'algorithme standard de création de l'ordre d'arrêt. Si la valeur est `custom`, l'ordre d'arrêt est laissé au contrôleur de cycle de vie du composite englobant. La valeur `list` permet de spécifier dans le MDL le nom des composants dans l'ordre où l'on souhaite qu'ils s'arrêtent. pour cela on utilise l'élément `<stop-component>` en donnant avec l'attribut `name` le nom du composant à arrêter.

5.2.4.6 Apports du MDL

Notre langage MDL est un outil qui répond au besoin de flexibilité de configuration et d'automatisation du déploiement. Son utilisation apporte une vision globale de la configuration de l'intergiciel et permet de spécifier son architecture à partir d'une bibliothèque de composants. La spécification du typage et des patrons (importée de l'ADL de Julia) permet une instanciation automatique des composants primitifs de l'intergiciel. L'architecture de composition des primitifs permet une meilleure structuration de l'intergiciel et simplifie le déploiement local de chaque composant. Ce déploiement local (c'est-à-dire la création, l'initialisation et le démarrage) est géré par le contrôleur de configuration décrit dans la section suivante. De plus l'association de ce contrôleur avec le MDL permet d'éviter

la reconfiguration « d'un bloc » de l'intergiciel, dont l'architecture peut évoluer par « bouts » (voir section 5.2.5.6).

Le déploiement global de l'intergiciel est grandement facilité par la définition des valeurs à récupérer (élément `return`). Ces valeurs pourront être ensuite traitées par le composant local d'administration (le LA) pour lier les différents « bouts » d'intergiciels. Nous verrons plus en détail le déploiement global dans la section 5.5.

Deux éléments spécifiques font leur apparition dans le MDL, il s'agit de l'élément d'initialisation et de l'élément de politique d'arrêt. Ils sont utilisés par le contrôleur de configuration des composants de DREAM décrit dans la section suivante. Ces deux éléments permettent de spécifier précisément la gestion du cycle de vie des composants. Ces deux éléments sont réellement un plus pour la gestion dynamique de l'architecture de l'intergiciel. C'est grâce à ces deux éléments qu'il est possible de manipuler dynamiquement l'architecture de l'intergiciel tout en garantissant une certaine intégrité. En cela, le MDL répond au besoin de mécanismes de contrôle (notamment des phases de démarrage et d'arrêt) pour le déploiement et la configuration dynamique de l'intergiciel.

Un exemple de MDL pour un service de type *Event-Based Service* (voir chapitre 6) est donné en annexe C

Les sections suivantes présentes les contrôleurs de DREAM, partie essentielle qui fournit les mécanismes pour la configuration de l'intergiciel.

5.2.5 Les contrôleurs de DREAM

Dans le chapitre 4, nous avons mis en avant le besoin de mécanismes de contrôle permettant de faciliter la configuration et le déploiement de l'intergiciel sur les différents sites d'exécution. Julia nous fournit des mécanismes de contrôle basés sur des contrôleurs et des intercepteurs présents dans la membrane du composant[5] (dans son conteneur). Un contrôleur est une entité (concrètement un objet) possédant du code capable de modifier le comportement ou la structure interne d'un composant. C'est grâce à ces mécanismes de contrôle, qui représentent le *modèle de contrôle*, que nous pouvons manipuler les différents composants de DREAM.

Les composants de DREAM utilisent plusieurs types de contrôleurs. Certains sont simplement les contrôleurs basiques fournis par Julia. C'est le cas du contrôleur de contenu (resp. du contrôleur de liaisons) que nous avons intégré tel quel dans DREAM car nous n'avions pas besoin de plus de fonctionnalités que celles fournies par l'interface de base *BasicContentController* (resp. *UserBindingController*). Pour les autres contrôleurs nous avons soit modifié l'interface de contrôle de base fournie par Julia (c'est le cas du contrôleur de nom, du contrôleur de cycle de vie et naturellement du contrôleur d'attributs[6]), soit créé un nouveau contrôleur pour nos besoins propres (c'est le cas du contrôleur de configuration). Nous décrivons en détails, dans les paragraphes suivants, l'ensemble de ces contrôleurs[7].

[5] Nous rappelons que le mot « composant » signifie aussi bien un composant primitif que composite.

[6] Tout contrôleur d'attribut étend l'interface *AttributController* afin de fournir les « getter » et « setter » associés aux attributs spécifiques du composant (`void setX(int X), int getX(), ...`).

[7] Nous ne parlons pas dans cette section de l'interface *ComponentIdentity* qui est une interface spécifique, commune à tous les composants permettant d'obtenir une référence sur un composant. Nous avons utilisé l'implémentaion de base fournie par Julia.

5.2.5.1 Contrôleur de nom

En règle générale, tout composant de DREAM implémente un contrôleur de nom. Le contrôleur de nom offre une interface pour accéder au nom logique d'un composant et pour le modifier. Dans DREAM, un nom est une information qui identifie un composant[8] (permettant de le distinguer des autres composants). Ce nom est une simple chaîne de caractères donnant le nom logique du composant mais aussi sa place dans l'architecture d'une configuration de DREAM. Autrement dit, le nom logique d'un composant est préfixé par le nom du composant composite auquel il appartient. Ce composite pouvant faire parti lui-même d'un autre composite, son nom est préfixé par son composite père, et ainsi de suite, les noms étant séparés par des points.

Par exemple, le composant `NetworkIn` est un sous-composant du composite `BasicNetwork` lui-même sous-composant du composite `Network` lui-même sous-composant du composite englobant `Dream`. Le nom de ce composant sera donc : « Dream.Network.BasicNetwork.NetworkIn ». Le nom complet d'un composant est donc déterminé à sa création par sa place dans l'architecture définie dans le MDL.

Listing 5.1 – Interface du contrôleur de nom

```
public interface NameController
{
  String getFcName();

  void setFcName(String name);
}
```

Le nom est essentiel car il permet de retrouver facilement la référence vers un composant à l'exécution (c'est-à-dire une référence vers son interface *ComponentIdentity*) à partir d'une simple chaîne de caractères. Il n'est pas nécessaire d'avoir le nom complet d'un composant pour obtenir une référence, le nom logique final (= le nom relatif) suffit à retrouver un composant à n'importe quel niveau d'encapsulation.

Le nom d'un composant n'est valable que pour une instance de DREAM donnée, c'est un nom local défini par le MDL de cette instance. Pour deux instances de Dream différentes on utilise souvent les mêmes composants (les même « pattes » réseaux au niveau du *Network* par exemple) qui ont donc le même nom logique. Pour les différencier on utilise un nom de niveau supérieur affecté au composite englobant : DREAM. Ainsi, un intergiciel DREAM dans son ensemble (c'est-à-dire plusieurs instances de DREAM sur des sites distants) est composé d'entités DREAM dont le nom global est affecté par un outil de déploiement (par exemple *dream1*, *dreamPDA1*, *dream2*, ...). La section 5.6.1.1 explique plus en détail ce nommage.

De la même manière un nom spécifique est donné aux services d'une instance de DREAM. C'est un nom global appelé `ServiceId` permettant de désigner un service dans un intergiciel DREAM constitué de plusieurs instances. Nous verrons plus en détail dans la section 5.4 comment sont affectés ces noms de service.

5.2.5.2 Contrôleur de cycle de vie

Un contrôleur de cycle de vie est utilisé pour gérer le cycle de vie d'un composant. Un cycle de vie est modelé par un automate à état. Dans notre cas, l'interface définit deux états (en relation avec

[8] A la différence de certains intergiciels (comme Jonathan par exemple) nous n'utilisons pas la notion de « contexte de nommage » (*naming context*) qui offre, certes, une plus grande liberté (et un nommage plus puissant) mais qui aurait compliqué notre système de nommage restreint à un environnement local.

les activités, ou processus légers, pouvant s'exécuter dans un composant) :

– L'état démarré (STARTED) dans lequel le composant peut émettre et accepter des appels de méthode ;
– L'état arrêté (STOPPED) dans lequel un composant ne peut pas émettre d'appels de méthode et peut recevoir des appels de méthode uniquement sur ses interfaces de contrôle (et donc pas sur ses interfaces fonctionnelles).

Suivant l'état dans lequel se trouve un composant, il sera possible ou pas d'accéder à certains de ces contrôleurs. Par exemple, il n'est pas possible de changer la structure interne d'un composant composite (c'est-à-dire d'ajouter ou de retirer des composants) si celui-ci n'est pas dans l'état arrêté.

L'interface de base fournie par Julia définit aussi trois méthodes `getFcState()`, `startFc()`, `stopFc()` qui, respectivement, retourne l'état du composant (démarré ou arrêté), démarre le composant, arrête le composant. Comme cela est suggéré ci-dessus, un appel de méthode (fonctionnelle) sur un composant arrêté sera bloqué jusqu'à ce que le composant soit (re)démarré (listing5.2).

Listing 5.2 – Interface standard du contrôleur de cycle de vie

```
public interface LifeCycleController {

  String STARTED = "STARTED";

  String STOPPED = "STOPPED";

  String getFcState ();

  void startFc ();

  void stopFc ();
}
```

Ce blocage des composants appelant peut mener à des problèmes de blocage du système dans le cas où un processus léger s'exécute dans l'appelant. Sur l'exemple de la figure 5.4, la méthode d'arrêt est appelée sur le composant 2 (sur son contrôleur de cycle de vie, *LifeCycleController*), puis le processus léger du composant 1 fait un appel sur la méthode fonctionnelle `printHello()`. Le composant 2 étant dans l'état arrêté, l'appel de méthode est bloqué et le processus est donc lui aussi bloqué en attente de redémarrage du composant 2. Si le système essaye ensuite d'arrêter le composant 1, celui-ci va attendre que le processus soit libéré de son appel sur le composant 2, et cet appel sur la méthode `stopFc()` (en étape 3) sera donc lui aussi bloqué. L'appel 3) ne rend donc pas la main et le système est bloqué, ne pouvant rien faire.

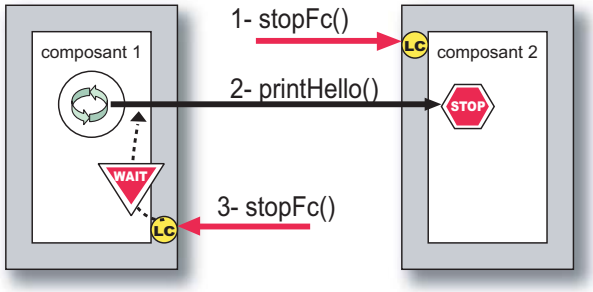

Figure 5.4 – Exemple de blocage à l'arrêt d'un composant.

Une solution pour éviter ce blocage est de toujours arrêter les composants comportant un processus léger avant d'arrêter les autres composants. Cela permet d'éviter qu'un appel de méthode (généré par un thread) s'intercale entre l'arrêt de tous les composants. En arrêtant le composant 1 avant le composant 2, dans notre exemple, on permet de garantir que le thread aura fini tous ses appels de méthode avant d'être arrêté. Il ne reste ensuite plus qu'à arrêter tous les autres composants. Evidemment, dans le cas où il y a deux composants avec des threads qui s'appellent mutuellement, il n'est pas possible de choisir quel composant arrêter en premier. De la même manière, il faut pouvoir arrêter un composant sans pour autant bloquer tous les composants qui sont liés à lui (par exemple arrêter le MOM de DREAM sans bloquer l'exécution des services).

Afin de ne pas créer de blocage, nous avons donc décidé de conserver l'information sur l'ordre d'arrêt des composants évitant un blocage. Cette information est conservée par le contrôleur de configuration du composite. Ainsi, l'arrêt d'un ensemble de composants se fait par l'appel de la méthode `stopFc()` sur le contrôleur de cycle de vie du composite englobant cette configuration. Le contrôleur récupère auprès du contrôleur de configuration l'ordre dans lequel arrêter ses sous-composants et propage cet appel au contrôleur de cycle de vie des sous-composants (voir figure5.5).

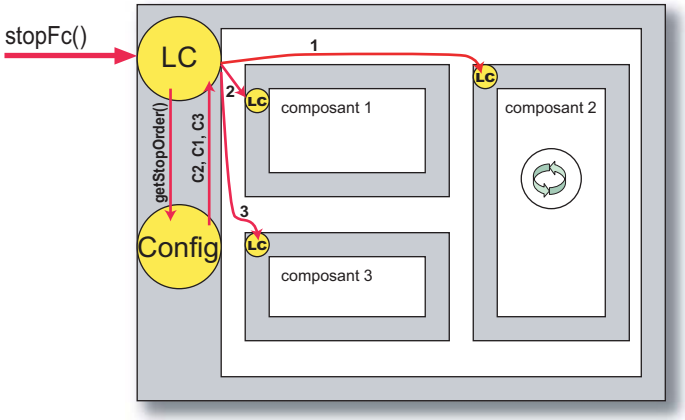

Figure 5.5 – Mécanisme d'arrêt des composants.

Nous avons définit plusieurs politiques d'arrêt dans le cas où un composant comportant un thread est impliqué dans l'architecture à arrêter. Ces politiques sont définies dans la description de la configuration de l'intergiciel : le MDL. L'élément `<stop-policy>` définit trois politiques possibles d'ordre d'arrêt :

- Automatique (`type="automatic"`), dans ce cas l'ordre d'arrêt est défini automatiquement par un algorithme lors de la création des composants.
- Par une liste explicite (`type="list"`), dans ce cas l'ordre d'arrêt est défini dans la suite du MDL en donnant explicitement le nom des composants dans l'ordre où le développeur désire qu'ils soient stoppés.
- Spécifique (`type="custom"`), dans ce cas l'ordre d'arrêt n'est pas conservé dans le contrôleur de configuration mais il est spécifiquement implémenté par un contrôleur de cycle de vie écrit par le développeur pour ce composite (c'est le cas du composant MOM, par exemple, qui possède une architecture et un contrôleur de cycle de vie spécifique).

La définition d'un contrôleur de cycle de vie spécifique permet de répondre au besoin de certains composants qui nécessitent une stratégie d'arrêt en rapport avec leur fonctionnement particulier. Ces contrôleurs n'utilisent pas l'implémentation par défaut mais une version qui répond à ses besoins. C'est donc au développeur de définir ce contrôleur.

Dans Dream, les composants services et le composant MOM sont les deux types de composants à posséder un contrôleur de cycle de vie spécifique couplé à une architecture spécifique elle-aussi. Nous verrons plus en détail dans les sections 5.3 et 5.4 ces spécificités.

5.2.5.3 Contrôleur de contenu

Un contrôleur de contenu est utilisé pour la gestion du contenu des composants composites. Seuls les composants composites possèdent donc ce contrôleur. Nous rappelons que le contenu d'un composant (composite) est un ensemble de composants non ordonnés. Le modèle permet de créer une hiérarchie de composants jusqu'à un niveau arbitraire. Ce contrôleur permet donc de contrôler cette structure hiérarchique de composants. L'interface spécifie des méthodes pour récupérer l'ensemble des sous-composants (l'intégralité de la composition), en ajouter de nouveaux ou en retirer. La sémantique des opérations d'ajout et de retrait est assez générale : Soit $C1$ le contenu d'un composant qui contient c et soit $C2$ le contenu d'un composant après une opération d'ajout (resp. de retrait), $C2$ est un nouveau contenu qui contient c (resp. ne contient pas c). La sémantique ne précise rien sur la relation entre $C1$ et $C2$, notamment elle ne garantit pas que $C2 = C1 \cup \{c\}$ (resp. $C1 = C2 \cup \{c\}$).

L'interface du contrôleur de contenu spécifie aussi une opération pour récupérer l'ensemble des interfaces internes du composant. Celles-ci sont utilisées pour être lier aux sous-composants.

Le contrôleur de contenu fournit une méthode de vérification des liaisons du composant. Plus précisément, cette méthode permet de vérifier que toutes les liaisons des sous-composants sont locales dans le contexte de ce composant.

La description de l'interface du contrôleur de contenu est donnée dans le listing5.3.

Listing 5.3 – Interface standard du contrôleur de contenu.

```
public interface ContentController {

  InterfaceReference[] getFcInternalInterfaces();

  InterfaceReference getFcInternalInterface(String interfaceName);

  ComponentIdentity[] getFcSubComponents();

  void addFcSubComponent(ComponentIdentity subComponent);

  void removeFcSubComponent(ComponentIdentity subComponent);

  void checkFc();
}
```

L'exécution de certaines opérations nécessite la vérification de contraintes sur l'état du composant. Il est, en effet, impossible d'ajouter ou de retirer des sous-composants si le composant composite est dans un état démarré. Ainsi, le contrôleur de contenu est relié au contrôleur de cycle de vie afin de vérifier son état avant de réaliser les opérations d'ajout et de retrait. S'il se trouve dans l'état démarré, l'appel sur l'une de ces deux opérations générera une exception propagée chez l'appelant. Par contre, s'il se trouve dans l'état arrêté, l'opération aboutira normalement.

5.2.5.4 Contrôleur de liaisons

Le contrôleur de liaisons permet de gérer les différentes liaisons pour un composant donné. Toute liaison entre deux composants **doit** se faire par l'intermédiaire du contrôleur de liaisons. Le modèle Fractal interdit à un composant d'appeler une méthode directement sur un autre composant

(en possédant sa référence Java par exemple). L'interface du contrôleur de liaison est donnée dans le listings 5.4.

Listing 5.4 – Interface standard du contrôleur de liaison.

```
public interface UserBindingController {

  Object getFcBindings (String clientItfName);

  void addFcBinding (String clientItfName, Object serverItf);

  void removeFcBinding (String clientItfName, Object serverItf);
}
```

Le contrôleur permet de récupérer l'interface (fonctionnelle) cliente d'un composant pour la lier à une interface (fonctionnelle) serveur d'un autre composant. Le contrôleur ne prend en charge que les liaisons locales à un composant :

- Les interfaces clientes et serveurs à lier doivent se trouver dans le même espace d'adressage (la même JVM)[9] : la référence d'une interface serveur est la référence Java vers l'objet qui implémente cette interface.
- Les composants auxquels appartiennent les interfaces clientes et serveurs à lier doivent se situer dans le même composant englobant : pour chaque liaison l_i entre un composant $c1_i$ et un composant $c2_i$, il doit exister un composant c qui contient directement ou indirectement $c1_i$ **et** $c2_i$.

C'est donc le contrôleur de liaisons du client qui contient les références vers les interfaces serveurs auxquelles il est lié. Le contrôleur de liaison d'un composant « serveur » ne possède pas de référence vers les clients auxquels il est lié.

Une liaison entre deux composants ne peux s'effectuer que si les deux composants sont dans un état arrêté (voir section 5.2.5.2). Lorsqu'une demande de liaison est réalisée (`bindFC(...,...)`), le contrôleur de liaison vérifie auprès du contrôleur de cycle de vie que le composant est arrêté.

5.2.5.5 Contrôleur d'attributs

Un contrôleur d'attribut est utilisé pour configurer les attributs primitifs d'un composant, c'est-à-dire les attributs qui sont de type primitif[10]. Le modèle spécifie que les développeurs doivent introduire des interfaces qui étendent l'interface `AttributeController` afin de contrôler, par des méthodes d'accès (*setter* et *getter*), les valeurs des attributs d'un composant.

Par exemple, supposons qu'un composant possède trois attributs `a1`, `a2` et `a3` de type entier (type Java `int`). Si l'on désire contrôler les attributs `a1` et `a2`, il faut fournir une interface qui étend `AttributeController` et qui fournit les méthodes : `int getA1()`, `void setA1(int a1)`, `int getA2()` et `void setA2(int a2)` (voir listing 5.5).

[9] Si les interfaces à lier ne se trouvent pas dans le même espace d'adressage, il est nécessaire de créer un composant de liaison spécifique possédant ses propres interfaces clientes et serveurs.

[10] Les types primitifs incluent les types primitifs Java (booléens, entiers, ...) et les chaînes de caractères.

Listing 5.5 – Exemple d'interface d'un contrôleur d'attributs

```
public interface MyAttController extends AttributeController{

  int getA1()

  void setA1(int a1)

  int getA2()

  void setA2(int a2)

}
```

5.2.5.6 Contrôleur de configuration

Afin de répondre à nos besoins de déploiement et de configuration dynamique, nous avons introduit un contrôleur spécifique associé à chaque composant de DREAM : le *contrôleur de configuration*. Le contrôleur est décrit dans le listing 5.6 ; il comporte trois fonctions de bases nécessaires à la configuration et au déploiement de l'intergiciel, les fonctions de création, d'initialisation et de configuration.

Création La fonction de création permet de créer un composant ou une hiérarchie de composants. Cette fonction n'est implémentée que par les composants composites. Elle reçoit des données (un objet `creationData`) qui contiennent, entre autre, la description du ou des composants à créer. Ces données ne sont en fait qu'une instanciation Java des données XML contenues dans le MDL (éléments `<struct>` et `<bindings>`). Ainsi, à la création, un composite reçoit les données de son architecture et instancie un à un ses sous-composants en fonction de l'architecture donnée (élément `<struct>`) puis lie les composants en fonction des ordres de liaison (élément `<bindings>`).

Initialisation La fonction d'initialisation consiste à positionner la valeur de certains attributs du composant (ou des sous-composants s'il s'agit d'un composite) et à récupérer certaines de ces valeurs. Cette fonction essentielle utilise le MDL, et plus précisément l'élément `<inits>` qui lui donne les paramètres à initialiser (voir section 5.2.4.4). La méthode d'initialisation d'un composant doit être appelée avant qu'il ne soit démarré (appel de la méthode `start()`) afin de garantir que tous ses paramètres soient positionnés à l'exécution.

Si cette méthode est appelée sur un composite, elle va récursivement appeler la méthode `init()` sur tous ses sous-composants qui nécessitent l'initialisation d'un paramètre (en résolvant les valeurs relatives des paramètres) puis elle retourne tous les paramètres à renvoyer (définis par l'élément `<return>`). Si cette méthode est appelée sur un composant primitif, celui-ci exécute sa méthode `init()` spécifique et peut retourner des informations qui seront récupérées par le composite. Les sous-composants sont initialisés dans l'ordre déclaré dans le MDL. Cet ordre, défini par le développeur a une très grande importance car il permet de garantir la validité des paramètres à valeur relative.

Configuration La fonction de configuration permet de modifier l'architecture à l'exécution. Cette fonction est pour l'instant laissée à la charge du développeur d'application[11].

Listing 5.6 – Interface standard du contrôleur de configuration.

```
public interface ConfigurationController {

    Object create(String componentName, CreationData creationData)
        throws CreationException;

    Object init(String componentName, Object initialisationData)
        throws InitializationException;

    Object configure(String componentName, Object configurationData)
        throws ConfigurationException;
}
```

5.3 Le composant MOM

Les sections précédentes ont décrit en détail le modèle de contrôle de DREAM qui fournit les mécanismes permettant la configuration et le déploiement de l'intergiciel. Nous allons maintenant présenter le modèle de délivrance de DREAM, le MOM.

Le composant MOM représente la partie communication de DREAM ; il offre aux services des interfaces simplifiées de communication asynchrone. Le MOM fournit une interface serveur *Push* et une interface cliente *Push*. Le mode *Push* (voir chapitre 1) est le mode de communication asynchrone par excellence. Il consiste à réaliser un envoi/reception de manière implicite. Lorsqu'un service désire envoyer un message il le « donne » au MOM qui se charge de le transférer à destination. De même, lorsque le MOM reçoit un message pour un service il le « pose » dans le tampon du service.

Architecture externe Vu de l'extérieur, le MOM est donc un simple composant avec une interface Push en entrée et en sortie. Néanmoins, cette architecture n'est pas satisfaisante à l'exécution. En effet, un des buts de l'architecture de DREAM est de permettre la reconfiguration à l'exécution. l'architecture simpliste dans laquelle les services sont directement liés au MOM impose des restrictions lors de l'arrêt du MOM pour configuration. Une fois le MOM dans l'état arrêté, tout service qui fera un appel sur l'interface *Push* sera bloqué en attente de redémarrage du composant (voir section 5.2.5.2). Ce blocage peut être long en fonction du temps que prendra la reconfiguration (si elle nécessite une validation dépendante d'autres sites), et le blocage d'un service peut être très gênant dans le cadre de certaines applications (il peut avoir besoin de faire des calculs, ...).

La solution que nous avons utilisée pour éviter ce blocage est d'insérer un composant de substitution tampon entre le MOM et les services et ceci de façon transparente pour les services. Dans cette architecture (voir figure 5.6), la partie fonctionnelle du MOM est englobée dans un composant appelé `Administrable_MOM` qui possède un contrôleur de cycle de vie spécifique. Lorsque le MOM doit être arrêté, le contrôleur donne l'ordre au tampon (`MOM_in_Buffer`) de ne plus transmettre les messages qui lui parviennent des services au MOM mais de les garder dans une file d'attente FIFO. Puis le MOM est arrêté. Ainsi, tous les messages venant des services sont stockés dans le tampon en attente de la fin de la reconfiguration. Une fois que tout est terminé, le contrôleur peut redémarrer le MOM puis « ouvrir la vanne » des messages sur le tampon.

[11] Une implémentation basique est proposée permettant uniquement l'ajout de nouveaux composants au sein d'une configuration (création, liaison et activation).

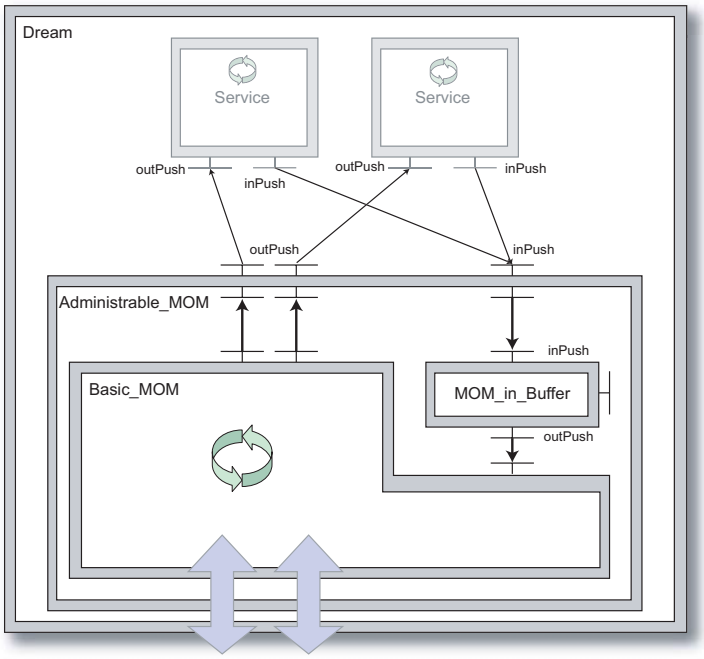

Figure 5.6 – L'architecture externe du MOM.

Listing 5.7 – Interface du contrôleur du tampon du MOM.

```
public interface PushPushBufferController {

    void openOut();

    void closeOut();
}
```

Architecture interne La particularité de DREAM est de fournir une architecture minimale assurant les fonctions de base sans propriétés non-fonctionnelles. C'est dans cette optique que l'architecture interne du MOM a été conçue (voir figure 5.7). Le MOM est constitué essentiellement de composites appelés `Network`, en charge d'envoyer et de recevoir les messages sur le réseau. Chaque composant `Network` possède deux composants *threadés*, un composant qui reçoit les messages en provenance d'autres entités DREAM (`NetworkIn`) et un composant qui envoie les messages vers les MOM d'autres entités `Dream` (`NetworkOut`). Il peut y avoir plusieurs composites `Network`, chacun pouvant être en charge de respecter une ou plusieurs propriétés non-fonctionelles requises par les services. Cette possibilité de choix de « portes » différentes est essentielle pour la configuration de l'intergiciel sur un réseau à grande échelle comme nous l'avons vu dans le chapitre 4 à propos des domaines de causalité.

Afin de sélectionner la porte à utiliser pour tel ou tel message, un répartiteur (le `DispatcherOut`) passe les messages des services à la « bonne » porte de sortie. Cette connaissance du routage des messages est connue soit statiquement au déploiement soit dynamiquement après reconfiguration. Toutes les données de routage des messages sont définies en fonction des besoins applicatifs mais aussi des contraintes du système (architecture du réseau, ...). De la même manière, un

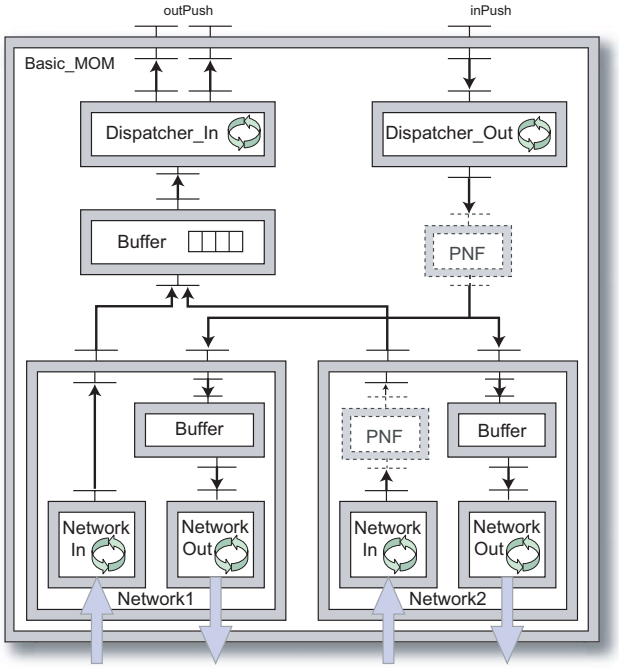

Figure 5.7 – L'architecture interne du MOM.

répartiteur de messages en entrée (le `DispatcherIn`) se charge de donner les messages au bon service en fonction du `ServiceId` contenu dans le message (voir section 5.4).

Cette architecture du MOM se veut suffisamment simplifiée pour pouvoir reproduire n'importe quel protocole asynchrone existant. Par exemple il serait simple de créer un `Network` qui respecterait le protocole d'ordonnancement causal des messages afin de permettre aux services de communiquer en conservant une cohérence d'exécution. C'est donc la modularité qui donne cette force à cette architecture de MOM.

5.4 Les services

La majorité des intergiciels asynchrones actuels ne proposent qu'un seul modèle de traitement et la possibilité de déployer plusieurs applications différentes sur une même instance de l'intergiciel en est donc réduite. Dans DREAM, chaque service représente un modèle de traitement de messages spécifique. Un service est un support d'exécution asynchrone qui utilise le composant MOM pour envoyer et recevoir des messages. Une application distribuée est composée d'un ensemble de services répartis sur différents sites. Chaque service étant indépendant des autres, une entité DREAM d'un site peut supporter plusieurs services faisant partie d'applications différentes[12]. Cette indépendance entre service permet de les configurer en fonction des besoins spécifiques de chaque application sans gêner l'exécution des autres services et donc des autres applications qui utilisent l'intergiciel.

Architecture L'architecture d'un service vu de l'extérieur est un simple composant qui possède une interface d'envoi de messages et une interface de réception de messages. Néanmoins, l'architecture doit répondre aux mêmes contraintes de disponibilité, lors d'une configuration, que ceux exposés dans

[12] Un service peut être vu comme une « personnalité » de l'intergiciel au sens Jonathan [88] (plusieurs personnalités pouvant coexister en même temps).

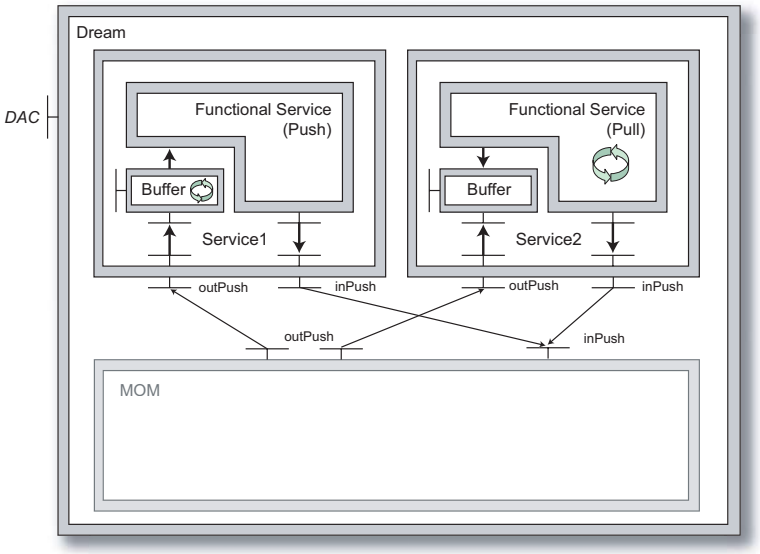

Figure 5.8 – L'architecture des services.

la section précédente pour l'architecture externe du MOM. Nous avons donc retenu le même type de solution, à savoir l'utilisation d'un tampon de messages en entrée (voir figure5.8). Cette architecture est déclinée en deux types possibles de services asynchrones, le service en mode *Push* et le service en mode *Pull*. Dans le premier cas, le tampon est *threadé* et lorsqu'il reçoit un message du MOM il va appeler la méthode de réaction sur la partie fonctionnelle du service. Dans le mode *Pull*, le tampon accumule les messages et c'est donc au service de venir chercher ces messages lorsqu'il le désire. Le contrôle du tampon est calqué sur celui du MOM.

L'architecture de DREAM permet donc la création de deux types de services, le service actif (mode *Pull*) et le service passif (mode *Push*). L'implémentation de la partie fonctionnelle du service est laissée à la charge du développeur.

Désignation Chaque service de DREAM doit pouvoir être accessible par les autres services ; chaque service possède donc un numéro unique en fonction du site sur lequel il se trouve. Le MOM étant responsable du routage entre les différents sites, il est donc désigné par un numéro unique défini au déploiement par son MDL. L'identifiant d'un service (ou son nom logique global) est donc composé du numéro de la plate-forme sur laquelle il s'exécute (le numéro du MOM) et d'une estampille locale unique du service. Cette identifiant global est appelé `ServiceId`

Contrôle Les services représentent l'application ; ils nécessitent donc une gestion spécifique du contrôle. Pour cela le composant englobant les services (le composant `Dream`) possède un contrôleur appelé `DreamAdministrationController` ou `DAC` (voir listing 5.8).

Listing 5.8 – Interface du contrôleur DAC.

```
public interface DreamAdministrationController {

  ServiceId addService(FunctionnalService desc);

  ServiceId getServiceId(String name);

  void activateService(ServiceId sid, Message msg);

  void startService(ServiceId sid);

  void stopService(ServiceId sid);

  void startMOM();

  void stopMOM();
}
```

Ce contrôleur permet d'ajouter, d'activer et de démarrer des services, y compris le MOM[13]. Pour l'ajout d'un service le contrôleur se comporte comme le contrôleur de configuration (section5.2.5.6). Il récupère un objet représentant l'architecture du service à créer puis l'instancie.

L'activation, souvent confondue avec le démarrage du composant, permet d'allumer la mèche d'exécution d'un service, c'est un démarrage applicatif[14]. En effet, dans le cadre d'une application asynchrone passve, chaque service est en attente de réception de message (modèle *Push*), il est donc nécessaire de lancer un premier message qui enclenchera les réactions en chaîne. Les données d'activation, reçues sous la forme d'un message, viennent du composant LA, une fois l'architecture globale de l'application mise en place et les services démarrés (au sens composant).

5.5 Le composant de déploiement et d'administration locale

Le composant d'administration est une entité pouvant être utilisée indépendamment de DREAM. L'ensemble des LA représente une application à part entière responsable du déploiement et de la configuration d'intergiciels pouvant être interfacée avec diverses plates-formes (c'est-à-dire d'autres intergiciels que DREAM). Chaque LA possède une partie communication asynchrone qui permet d'échanger les informations avec les autres LA et une partie protocole de configuration et déploiement. La partie communication est assurée par un composant MOM spécifique qui garantit la propriété de transfert fiable des messages. La partie protocole est spécifique à chaque intergiciel (voir figure 5.9). Actuellement, seule l'implémentation d'un protocole de déploiement et de configuration pour DREAM existe mais il suffirait d'implémenter la partie protocole d'administration pour réutiliser des LA comme application de déploiement et de configuration générique.

Le composant local d'administration est présent quelle que soit la configuration de l'intergiciel. Il utilise les mécanismes de contrôle décrits dans la section5.2.5 pour configurer l'intergiciel. Le LA peut être vu comme un composant de *bootstrap* de l'intergiciel. Il doit être présent sur chaque site pour pouvoir instancier l'intergiciel ou le déployer (à distance).

[13] Le MOM peut être vu comme un service en ce sens qu'il possède les même interfaces.
[14] Par opposition au `start()` qui est un démarrage au niveau composant.

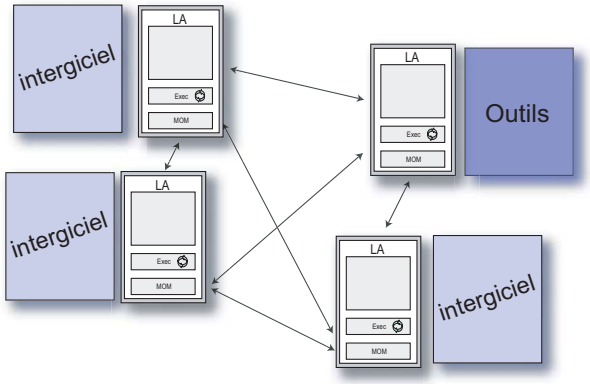

Figure 5.9 – Les LA sont présents sur chaque site.

Architecture du LA pour DREAM Le LA reçoit les ordres de configuration et de déploiement par son propre MOM. La version actuelle de DREAM permet d'envoyer un fichier MDL donnant la configuration de l'intergiciel à réaliser. Le noyau exécutif du LA récupère ce fichier et le donne au composant responsable de la vérification du fichier et de chargement des classes Java. Puis le composant suivant se charge de transformer ce fichier en un objet[15] de données compréhensibles par le contrôleur de configuration de DREAM (voir section 5.2.5.6). Le LA reçoit en retour des éléments pouvant servir au déploiement (port d'écoute du MOM, ...) de l'ensemble de la configuration. Il utilise pour cela l'interface DAC permettant de créer, d'activer et de démarrer les services et le MOM de DREAM.

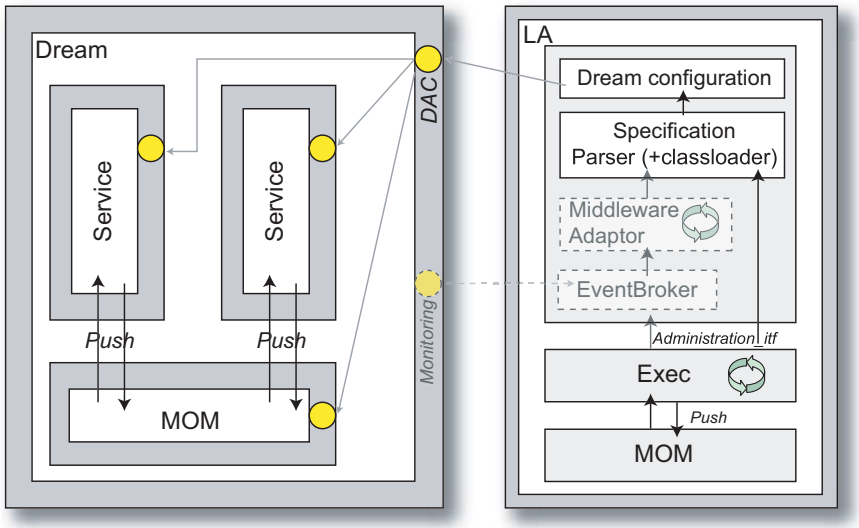

Figure 5.10 – Architecture du LA.

Le fichier de configuration de l'intergiciel peut être soit donné de façon ad hoc, c'est-à-dire « à la main » lors du lancement de l'intergiciel localement, soit créé par un outil de configuration et de déploiement.

[15] `CreationData` s'il s'agit d'une configuration initiale ou `ConfigurationData` s'il s'agit d'une configuration à l'exécution.

La notion de LA rejoint l'idée du modèle de déploiement Java d'*Application Helper* (AH) décrite dans [64]. L'auteur propose un modèle de déploiement dans lequel le AH est un programme présent localement sur chaque site client et qui peut exécuter des actions de téléchargement de classes, d'installation de nouvelles applications ou parties d'application. Mais cette idée est réduite à un environnement point à point où le AH fait le lien entre le serveur de déploiement et le site client et n'est pas adaptée à l'utilisation à grande échelle.

C'est pourquoi nous proposons dans la section 5.6 un modèle de configuration et de déploiement distribué à grande échelle utilisant une hiérarchie de LA pour configurer et déployer l'intergiciel.

Extensions Le LA a été conçu pour recevoir ultérieurement des composants d'observation de la plate-forme. Ces composants d'observation (de *monitoring*) reçoivent les informations sur l'intergiciel par l'intermédiaire d'un contrôleur d'observation. Ces informations sont ensuite traitées par un composant possédant une « intelligence » d'adaptation. Ce composant peut prendre la décision de reconfigurer l'intergiciel en créant un nouveau fichier de configuration. Les informations d'observation peuvent aussi être agglomérées par une unité de configuration globale (ou des unités hiérarchiques) prenant des décisions d'adaptation en fonction de l'ensemble des sous-sites. Ces notions de configuration et de déploiement à grande échelle sont traitées dans la section suivante.

5.6 Passage à grande échelle

Les sections précédentes ont présenté notre méthode de configuration locale sur une instance d'intergiciel donnée. Cette proposition permet désormais de résoudre les problèmes de la configuration et du déploiement local en déléguant au contrôleur de configuration la charge de mettre en place la configuration de l'intergiciel (donnée sous la forme d'un fichier MDL). La description MDL *est donnée au LA* qui se charge d'appeler les méthodes sur le contrôleur de configuration.

Cette dernière phrase parait naturelle mais elle met en jeu le problème extrêmement complexe du déploiement et de la configuration distribuée à **grande échelle**. En effet, si l'action de « donner une description MDL au LA » paraît simple localement (il suffit que le LA lise un fichier MDL sur disque par exemple) elle devient beaucoup plus compliquée lorsqu'un ensemble de LA doivent installer une configuration globale de DREAM sur un ensemble de sites distribués. Cette complexité tient essentiellement aux aspects de grand nombre de participants et de contraintes de distribution. L'augmentation du nombre de participants entraîne une augmentation des coûts de gestion du déploiement, du nombre d'erreurs possibles à traiter, de la coordination des différents acteurs, etc. L'aspect distribué met en jeu les problèmes de contraintes physiques dus à la latence du réseau, à la déconnexion temporaire de sites, aux pannes des machines, au blocage de sites en attente, etc.

Description hiérarchique Dans cette section, nous essayons de répondre au problème de la scalabilité du processus de déploiement et de configuration en nous inspirant des travaux réalisés par la société Scalagent (présentés dans la section 2.3.2.1) qui utilisent une hiérarchisation du contrôle de déploiement testée avec succès.

Nous proposons d'utiliser une description globale de l'intergiciel, sous la forme d'un MDL donnant les différentes configurations de sites et leurs interconnexions, et de calquer le processus de déploiement sur la structure hiérarchique de cette description. Cet aspect hiérarchique, associé à un environnement d'exécution asynchrone et fiable, permet de déployer et de configurer l'intergiciel sur des sites répartis à très grande échelle tout en facilitant la gestion des pannes et des sites temporairement déconnectés.

Dans notre proposition, nous ne dissocions pas le processus de configuration du processus de déploiement. La notion de configuration englobe deux activités. Premièrement, *choisir* les éléments constitutifs de l'intergiciel, c'est-à-dire définir l'architecture globale de l'intergiciel, ses interconnexions, mais aussi les propriétés non fonctionnelles des sites, les différents services, etc. Deuxièmement, *placer* cette description sur les sites distribués afin de fournir le support d'exécution aux applications. La première activité est une tâche extrêmement complexe car elle nécessite de prendre en considération les besoins des applications et les contraintes du système. Ce sont des travaux tels que ceux réalisés par le projet Aster (voir 2.2.4). Nos travaux se concentrent sur le processus de placement et donc sur les mécanismes qui permettent la configuration de l'intergiciel.

Le début de ce chapitre s'est concentré sur les mécanismes locaux, les sections suivantes présentent donc les mécanismes distribués.

5.6.1 Diviser pour mieux... contrôler !

Déployer et configurer un intergiciel consiste à installer les différentes entités coopérantes, qui forment l'intergiciel, sur leurs sites respectifs et à leur donner le moyen de coopérer. « Donner les moyens de coopérer » signifie que l'on procède à la liaison de ces différentes entités, c'est-à-dire que l'on échange leur(s) référence(s) dans le système (la référence peut prendre des formes très différentes, dans le cas de DREAM il s'agit d'un port TCP comme nous le verrons dans la suite).

Alors que dans le déploiement applicatif, le processus se base sur un intergiciel (et donc l'ensemble de ses propriétés) pour s'exécuter, le déploiement d'intergiciel ne peux pas prétendre au support d'un autre intergiciel complet pour être déployé. Cela reviendrait à reporter le problème.

Nous pensons que le déploiement d'un intergiciel peut être vu de la même manière qu'un déploiement applicatif mais utilisant un support un peu plus allégé. Ainsi, notre processus se base sur l'hypothèse[16] qu'il existe un LA sur chaque site où l'intergiciel sera potentiellement déployé. L'ensemble des LA répartis va permettre de distribuer le contrôle du déploiement et de la configuration entre plusieurs entités ayant chacune la responsabilité d'une partie du processus global. La parallélisation des différentes activités de chaque LA va permettre d'augmenter les performances lors du déploiement à grande échelle. De plus cette distribution du contrôle a l'avantage de fournir une continuité du service. En cas de panne de certains LA, les autres peuvent continuer à fonctionner et cela grâce à l'utilisation de communications asynchrones.

La propriété de distribution doit être associée à celle de fiabilité pour garantir que le processus résiste aux erreurs. Cette propriété est assurée par le processus global qui permet de résister aux erreurs levées par le déploiement local et aux blocages occasionnés par les pannes des sites. Ces erreurs sont remontées au niveau de l'initiateur afin qu'il prenne les décisions adéquates.

Hiérarchisation du traitement La parallélisation des activités de déploiement-configuration est réalisée selon un découpage hiérarchique des LA. Initialement, tous les LA se connaissent et peuvent dialoguer entre eux (figure 5.9). Puis, un LA initiateur (commandé par un outil par exemple) reçoit une configuration globale d'intergiciel à mettre en place. Dans cette configuration, sous forme d'une description MDL (voir section suivante), les LA participant à la configuration sont découpés en une hiérarchie de groupes de LA, appelés domaines, sous la forme d'un arbre. Chaque domaine possède un LA responsable des ordres de déploiement vers les LA de son domaine et de ses sous-domaines. Ce LA, appelé LA-maître, contrôle le déploiement et la configuration de l'ensemble des LA présents dans le domaine et est l'initiateur des ordres vers les LA maîtres de ses sous domaines (voir figure 5.11).

[16] Cette hypothèse peut être restrictive car elle sous-entend que les LA ont dû être installés **avant**. Néanmoins ce problème récurrent du *bootstrap* pourrait être contourné par l'intégration, comme service système, des LA.

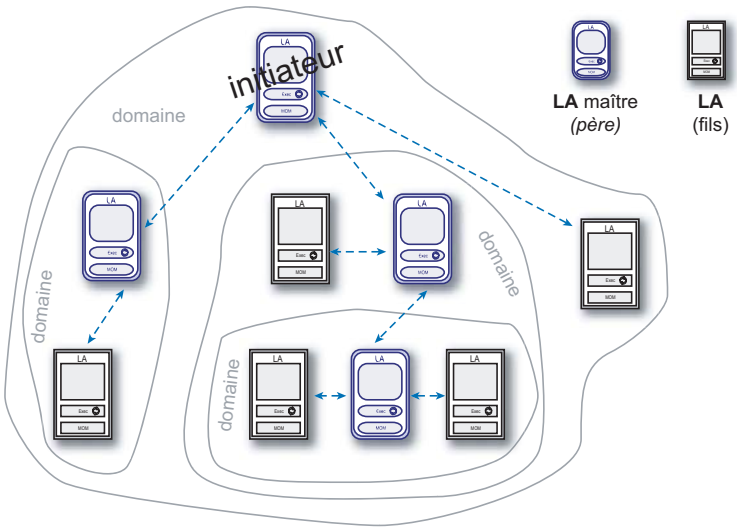

Figure 5.11 – Hiérarchisation des LA.

Actuellement, le LA-maître est choisi de façon arbitraire, il s'agit du premier LA rencontré dans la description globale. Mais le choix du LA-maître pourrait tout à fait être dicté par des choix topologiques (architecture du réseau par ex.), physiques (puissance de la machine par ex.) ou logiques (en fonction des applications par ex.) et définis par l'initiateur. De la même manière le découpage en domaines est donné par l'initiateur, c'est donc lui qui décide comment vont « se parler » les LA. La façon de découper les sites est actuellement plus ou moins arbitraire : le découpage est réalisé en fonction de la communication des sites (voir section suivante). Mais il serait beaucoup plus judicieux de réaliser ce découpage à l'aide d'un algorithme automatisé qui prendrait en paramètre la répartition géographique des sites ou les propriétés des sites par exemple. Ces travaux font partie des perspectives de notre travail.

A chaque LA est donc rattachée une configuration d'intergiciel à mettre en place. Cette configuration est décrite à l'aide du MDL global.

5.6.1.1 MDL et architecture globale

Alors qu'une description MDL locale permet de décrire la configuration d'un site, une description MDL globale décrit l'ensemble des configurations de chaque site et leurs interconnexions. Dans le cadre de DREAM, les différents sites sont reliés par leur composant `Network`. Ainsi, un service qui désire communiquer avec un service d'un autre site passe le message à son composant `Network` qui enverra le message au `Network` du site destinataire[17].

La description globale est découpée en groupes d'intergiciels appelés domaines. Un domaine est un ensemble d'intergiciels dont le déploiement et la configuration sont « pilotés » par un LA spécifique appelé LA-maître qui dialogue avec les LA de chaque site pour mettre en place la configuration globale. La figure 5.12 illustre une configuration globale découpée en domaines de LA.

Comme nous l'avons souligné précédemment, le découpage des domaines de LA n'est pas forcément corrélé à la configuration des différents sites et de leurs interconnexions. Actuellement la hiérarchie des LA doit être réalisée par un intervenant extérieur[18] afin de répartir au mieux leurs

[17] Actuellement, le protocole utilisé entre les `Network` est restreint à TCP/IP. Il faudrait étendre le MDL pour définir, dans la description, le protocole à utiliser.

[18] L'intervenant extérieur peut être un administrateur humain, un outil intelligent automatisé, etc.

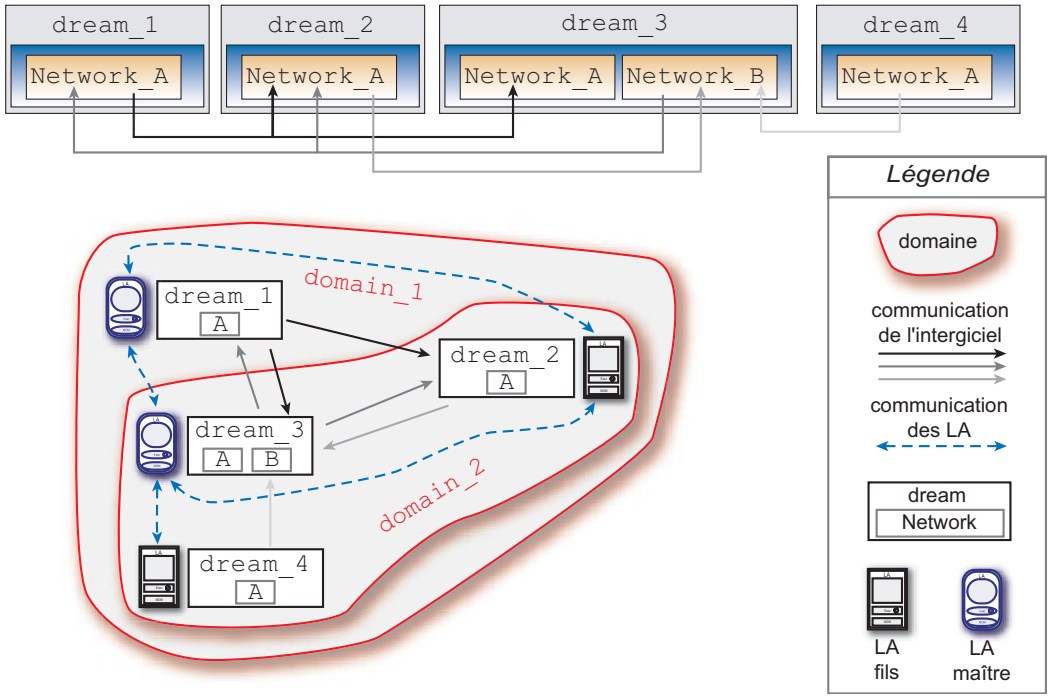

Figure 5.12 – Architecture distribuée et domaine de configuration.

échanges d'informations. Nous partons donc du principe que la description est donnée par une entité supérieure au LA maître se trouvant au sommet de la hiérarchie.

Dans le MDL global apparaît un nouvel élément, il s'agit de l'élément `domain`. La forme générale d'un élément `domain` est la suivante :

```
<domain name="domainName">
  ...
  <composite name="dreamName" type="dreamType" host="...">
  ...
  </composite>
  <bindings>
    <binding client="domainName.dreamName.NetworkName"
      server="domainName.dreamName.NetworkName">
    ...
  </bindings>
  ...
</domain>
```

l'attribut `name` donne le nom unique du domaine. Ce nom est très important car il permet de situer un site dans la hiérarchie globale. Lors d'une liaison entre deux sites (exprimée par l'élément `binding`) la référence donnée pour la liaison est le nom *complet* du site à atteindre. Le nom complet est représenté par la succession des noms des différents domaines « traversés » pour atteindre le site. Cette hiérarchisation du nom permet au LA de retrouver le site dans la hiérarchie des domaines (par exemple `domain1.domain3.dream8.NetworkB`). Si le site à lier se trouve dans le domaine courant, il est possible d'utiliser le mot `this` pour désigner le domaine.

Le listing ci-dessous donne la description MDL pour la configuration globale définie dans la figure 5.12.

```
<domain name="domain_1">
  <composite name="dream_1" type="dreamType" host="194.199.25.18">
    ...
    <composite name="Network_A" type="compositeNetwork">
    ...
  </composite>
  <domain name="domain_2">
    <composite name="dream_2" type="dreamType" host="194.199.25.30">
      ...
      <composite name="Network_A" type="compositeNetwork">
      ...
    </composite>
    <composite name="dream_3" type="dreamType" host="194.199.25.42">
      ...
      <composite name="Network_A" type="compositeNetwork">
      ...
      <composite name="Network_B" type="compositeNetwork">
      ...
    </composite>
    <composite name="dream_4" type="dreamType" host="194.199.25.50">
      ...
      <composite name="Network_A" type="compositeNetwork">
      ...
    </composite>
    <bindings>
      <binding client="this.dream_2.Network_A" server="this.dream_3.Network_B">
      <binding client="this.dream_3.Network_B" server="this.dream_2.Network_A">
      <binding client="this.dream_4.Network_A" server="this.dream_3.Network_B">
    </bindings>
  </domain>
  <bindings>
    <binding client="this.dream_1.Network_A" server="domain_2.dream_2.Network_A">
    <binding client="this.dream_1.Network_A" server="domain_2.dream_3.Network_A">
    <binding client="domain_2.dream_3.Network_B" server="this.dream_1.Network_A">
  </bindings>
</domain>
```

Les liaisons internes (locales) à un domaine sont décrites dans l'élément du dit domaine. Tandis que toutes les liaisons externes au domaine (c'est-à-dire qui ne font pas partie du domaine ou qui sont entre ce domaine et un autre domaine de niveau supérieur) sont exprimées dans l'élément du domaine père (domaine de niveau supérieur).

Comme on peut le voir sur ce listing, les liaisons sont définies comme un lien entre deux composants `Network`. Dans notre cas très spécifique de liaison TCP/IP entre les `Network`, le fait d'établir une liaison correspond à donner au `Network` client le numéro du port TCP d'écoute du `Network` serveur. Le port d'écoute peut être changé à l'initialisation du `Network` serveur même s'il a été défini dans la description MDL[19]. C'est donc le LA local qui se charge de récupérer la nouvelle valeur et de la transmettre au LA-maître. Cette activité fait partie du processus global décrit dans la suite.

5.6.2 LA fils

Le LA fils possède un rôle de simple exécutant. Il permet au processus de déploiement de maîtriser l'activité d'une instance d'intergiciel locale et de configurer ses liens avec d'autres sites. Il reçoit une configuration locale sous la forme d'un fichier MDL et a la charge de l'instancier puis de renvoyer les valeurs de retour et enfin de lier ses `Network` locaux. Le fonctionnement d'un LA est décrit en termes de messages car le processus global est fondé sur un mode de communication asynchrone.

[19] On peut imaginer que le port défini dans le MDL est utilisé au moment de la mise en place sur le site. Le composant `Network` peut alors définir une nouvelle valeur.

Le rôle du LA fils est donc découpé en plusieurs étapes, avec tout d'abord deux activités séquentielles :

1. *Instanciation* : Cette activité est réalisée lorsque le LA reçoit un fichier de configuration locale, c'est-à-dire la description MDL de la configuration à mettre en place (comme celle de l'annexe C.1 par exemple). Le LA instancie le composite `Dream` puis appelle la méthode *create()* sur le contrôleur de configuration en passant en paramètre la configuration donnée dans la description MDL.
2. *Initialisation* : La phase d'initialisation permet de fixer les valeurs de certains paramètres sur les composants. Pour cela le LA appelle la méthode *init()* du contrôleur de configuration sur les composants spécifiés par l'élément `<init...>` du fichier MDL. Il est important de se rappeler que les valeurs données par la description MDL peuvent être changées par le composant lors de l'initialisation, ce serait typiquement le cas d'un port TCP déjà utilisé.

   Dans le cas du composant `Network` le LA doit initialiser des paramètres très spécifiques qui portent toujours le même nom : `sendingPort_X` et `listeningPort`. Le premier représente les différents liens vers les ports d'écoute distants auxquels ce `Network` peut vouloir envoyer des messages. Il peut y avoir plusieurs ports d'envois identifiés par un numéro spécifique (lettre `X`). Le deuxième est unique et donne le port d'écoute de ce `Network`. Cette valeur est initialisée par le LA et peut être modifiée lors de l'initialisation par le composant `NetworkIn` (qui possède toujours un paramètre `listeningPort`).

La figure 5.13 illustre l'activité d'initialisation puis l'utilisation du port lors de la liaison distante.

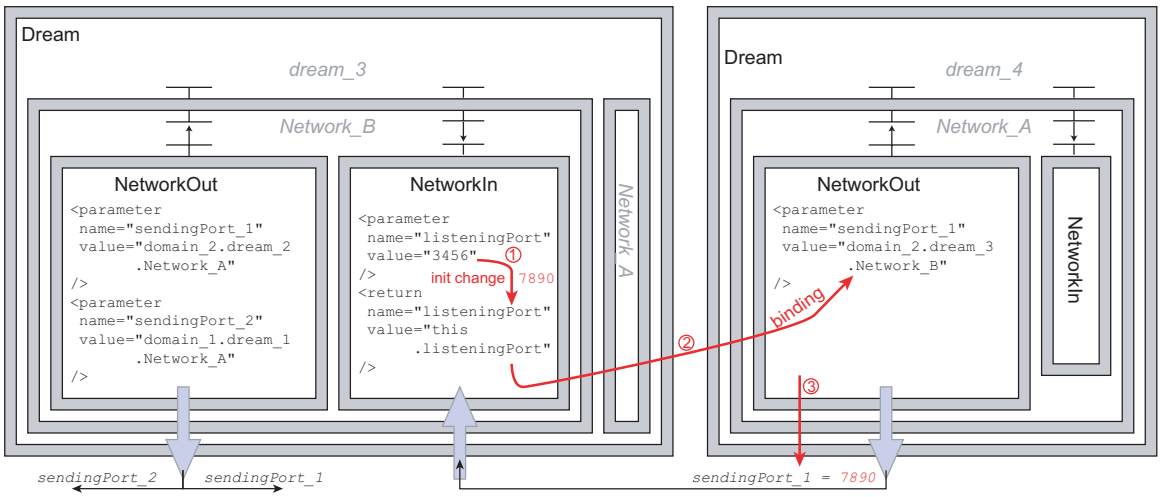

Figure 5.13 – Initialisation et liaisons distribuées.

Après les deux étapes présentées ci-dessus viennent deux activités pouvant être réalisées dans un ordre arbitraire. Tout d'abord l'exportation qui renvoie le numéro du port d'écoute TCP du `Network` serveur au LA-maître. Puis la liaison qui permet de relier effectivement deux `Network`.

Une fois que ces activités ont été réalisées vient l'opération d'activation qui permet de faire effectivement démarrer l'intergiciel. Il est possible de faire démarrer l'intergiciel même si toutes les liaisons n'ont pu être faites (≈ même si tous les `Network` voulant être liés n'ont pu l'être). Cela permet de ne pas bloquer complètement l'intergiciel et de démarrer au moins une partie des sites.

3. *Exportation* : La phase d'exportation permet de récupérer les valeurs données dans l'élément `<return...>` de la description MDL, la valeur la plus importante étant celle du paramètre

`listeningPort` des composants `Network` qui donne le port d'écoute effectif du `Network` (valeur qui a pu être modifiée à l'initialisation). Ces valeurs sont ensuite envoyées au LA-maître qui aura la charge de les retransmettre aux `Network` clients.

4. *Liaison* : L'activité de liaison est initiée par le LA-maître du domaine. Elle consiste à envoyer un message au LA fils du `Network` client en lui donnant la valeur du port d'écoute du `Network` serveur auquel il désire se lier. Pour cela, à chaque réception d'une autorisation de liaison, le LA appelle la méthode *init()* sur le `NetworkOut` désigné en donnant les données d'initialisation restreintes à la valeur du paramètre `sendingPort` désigné. Ainsi, dans notre exemple, lorsque le LA du site `dream_4` reçoit la liaison `<server="this.dream_3.Network_B" host="194.199.25.42" port="7890">`, il voit dans son référentiel que cette valeur est rattachée à l'attribut `sendingPort_1` de son `Network_A` et met à jour cette valeur.
5. *Activation* : L'activation est initiée par le LA-maître. Elle consiste à appeler la méthode *start()* sur le contrôleur de cycle de vie du composite DREAM.

La figure 5.14 illustre les transitions et les changements d'état local. Dans cette figure, seule la transition d'exportation est initiée localement par le LA après l'initialisation, les autres sont initiées par le LA-maître du domaine. Chacune de ces opérations génère des événements de retour au LA-maître, lui permettant de maintenir une représentation de l'état de chaque site du domaine.

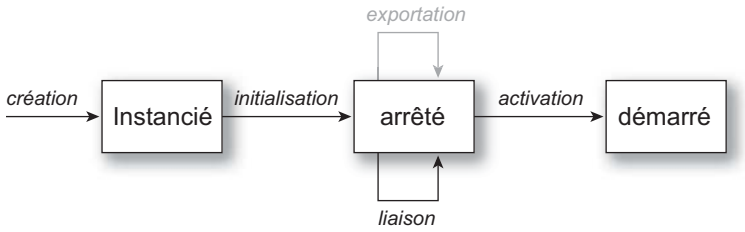

Figure 5.14 – Graphe d'état de l'intergiciel local.

La section suivante explique le fonctionnement des LA-maîtres, et comment est géré le processus global.

5.6.3 LA-maître et processus global

Le fonctionnement d'un LA-maître est toujours initié par le LA-maître de niveau supérieur (son pére), sauf dans le cas du LA initiateur. Une communication asynchrone est créé entre le père et le fils. Le LA père envoie les demandes d'opérations et le LA-maître de niveau inférieur exécute et renvoie les événements de contrôle (les exportations, les acquittements, etc.). Les résultats des opérations effectuées permettent de mettre à jour un référentiel d'architecture du domaine conservé par le LA.

Le LA-maître a donc un double rôle, celui de LA fils pour son intergiciel local, celui d'un initiateur pour tous les LA fils de son domaine et tous les LA-maîtres de ses sous domaines. Il ouvre des sessions de communication avec ses LA de niveau inférieur en envoyant les ordres de configuration/déploiement.

Organisation globale Lors de l'ouverture d'une session (au démarrage du déploiement), le LA-maître crée les différentes activités en accord avec la description MDL du domaine auquel il est associé. Il y a une activité de création et d'activation par site qu'il doit créer et une activité de liaison pour chaque liaison à laquelle il est directement rattaché (c'est-à-dire les liaisons internes à son domaine ou entre son domaine et un sous domaine). Ces activités peuvent de dérouler de façon indépendante

mais toujours en lien avec le référentiel conservé. Elles s'exécutent parallèlement ou séquentiellement suivant leur type. Les activités de création sont toutes parallèles alors que la liaison ne peut s'effectuer qu'après la création et la réception des éléments de retour nécessaires (port TCP serveur). Il est donc possible d'envoyer un ordre d'activation même si toutes les liaisons n'ont pas été réalisées, permettant une disponibilité rapide de l'intergiciel. Cela peut d'ailleurs créer des problèmes de cohérence de l'intergiciel qui doivent être gérés par l'initiateur de plus haut niveau (voir section6.6).

1. *Création* : Pour la création, le LA-maître reçoit la configuration à mettre en place sous la forme d'un MDL global qui décrit toute la configuration de son domaine et de ses sous domaines. Le LA découpe cette description hiérarchique et envoi à chaque LA maître des sous domaines la description qui les concerne (c'est-à-dire le sous ensemble du MDL qui correspond à ce domaine). Ensuite, le LA attend le retour des exportations des La fils (ou LA maître) qui doivent être liés dans le domaine.

   La création se diffuse sur l'arbre des domaines selon les liens entre les LA.

2. *Liaison* : Une liaison est réalisée lorsque le LA reçoit un événement d'exportation d'un LA fils (s'il s'agit d'une liaison locale au domaine) ou d'un LA-maître (s'il s'agit d'une liaison inter-domaine). Lorsque les LA fils envoient leurs événements d'exportation, le LA-maître utilise l'élément `<binding...>` pour savoir quelles exportations il attend en retour.

   Ainsi, dans l'exemple de la figure 5.13, le LA maître du domaine 2 reçoit du site `dream_3` la valeur `domain_2.dream_3.Network_B.listeningPort=7890`. Il regarde dans son référentiel toutes les liaisons qui ont besoin de cette valeur et renvoie l'information au(x) client(s) concerné(s). Dans notre exemple il renvoie l'information : `<server="this.dream_3.Network_B" host="194.199.25.42" port="7890">` à `domain_2.dream_4.Network_A`.

3. *Activation* : L'opération d'activation est envoyée par l'initiateur, elle peut être réalisée même si toutes les liaisons décrites dans le fichier MDL ne sont pas encore faites. L'activation comme la création se diffusent sur l'arbre des domaines.

Afin de connaître l'état d'avancement du déploiement, les LA-maîtres envoient régulièrement des informations à leur père. Ces informations sont remontées jusqu'à l'initiateur de plus haut niveau. Ces informations peuvent par exemple être utilisées par un outil graphique pour afficher une barre de progression. Mais surtout ces informations sont utilisées pour la gestion des pannes.

Gestion des pannes Le processus de déploiement à grande échelle est une tâche complexe qui est sujette à des pannes, à des sites déconnectés, des blocages, etc. Il est donc essentiel de pouvoir détecter les pannes et de pouvoir y réagir pour éviter un blocage total et complet de l'ensemble des sites. Nous pensons que la détection de panne doit agir comme un contrôle de *workflow* qui traite les éventuelles erreurs durant le processus de déploiement. Ces erreurs doivent pouvoir être traitées localement ou notifiées à l'initiateur du déploiement si une décision locale ne peut être prise. La section6.6 essaye de répondre à ces problèmes en proposant une solution basée sur une stratégie *optimiste* qui consiste à agréger les événements les plus importants vers l'initiateur. Le traitement des pannes est décidé par l'initiateur qui peut relancer un déploiement complet ou bien partiel.

5.6.4 Synthèse sur le passage à grande échelle

Cette section a présenté le processus de déploiement distribué pour notre intergiciel asynchrone. Comme nous l'avions vu dans le chapitre 2, le déploiement des intergiciels est encore un problème peu abordé. De plus, les travaux dans le déploiement applicatif sont très souvent gérés de manière centralisée et ne prennent pas en compte les contraintes du système.

Nous avons proposé un processus de déploiement et de configuration distribué hiérarchique. Notre processus permet de gérer le déploiement en découpant l'ensemble des sites en plusieurs sous ensembles appelés domaines. Chacun des domaines est « piloté » par un LA-maître qui a la charge de mettre en place la configuration pour son domaine et de déléguer la configuration de ses sous domaines aux LA-maître de niveau inférieur.

La hiérarchisation du déploiement permet d'éviter la création de goulots d'étranglement et facilite le passage à grande échelle. De plus l'exécution parallèle et asynchrone des différentes activités garantit une disponibilité rapide de l'intergiciel.

Néanmoins de nombreux travaux restent à fournir sur ce processus, notamment au niveau de la gestion des erreurs et de l'automatisation du traitement. Nous réalisons une étude qualitative de notre proposition dans la section 6.6.

5.7 Conclusion

Ce chapitre a présenté DREAM, un intergiciel asynchrone adaptable. Son architecture est dictée par un découpage en trois modèles : un *modèle de délivrance*, un *modèle de traitement* et un *modèle de contrôle*. Le modèle de délivrance est représenté par le composant MOM qui a la charge de transférer les messages d'un site à l'autre. Il permet aux services de s'abstraire des problèmes inhérents à la distribution (protocoles réseaux différents, routage,...). Le modèle de traitement est représenté par les services qui fournissent la sémantique de l'application et peuvent s'exécuter en parallèle (fournissant donc de multiples sémantiques à l'intergiciel). Le modèle de contrôle de DREAM est basé sur celui de Fractal et ajoute l'utilisation d'un composant spécifique (le LA) qui a en charge l'intégrité de la configuration et reconfiguration de l'intergiciel local et global.

Parallèle Cette architecture peut d'ailleurs être mise en parallèle avec l'architecture en couches pour intergiciel proposée par Douglas C. Schmidt dans [80], de plus en plus reconnue par la communauté. La correspondance entre DREAM et ce modèle d'architecture est présentée sur la figure5.15.

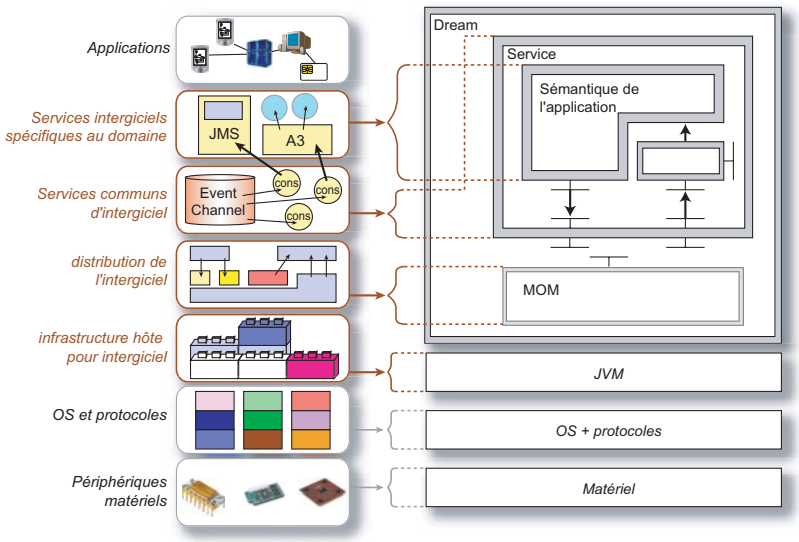

Figure 5.15 – Architecture de DREAM selon le modèle de D. C. Schmidt.

La partie infrastructure hôte, qui doit fournir les mécanismes génériques de communication et de concurrence est, dans notre cas, représentée par l'utilisation de la JVM. La partie distribution

de l'intergiciel est clairement représentée par le MOM de DREAM car il permet aux services de s'abstraire de la distribution sur les différents sites. La notion de services standard d'intergiciel est donnée par les services *Push* et *Pull* de DREAM ; Ils permettent de créer un modèle de traitement commun à tous les services spécifiques d'applications. Ce que D.C. Schmidt appelle les services de l'intergiciel spécifiques au domaine est symbolisé par l'implémentation de la partie fonctionnelle des services de DREAM qui donnent la sémantique à l'application.

Bilan A la fin du chapitre 4 nous avons axès nos motivations sur 3 points : les besoins de configurabilité, de configuration et de mécanismes de contrôle. Ces motivations étaient guidées par les défauts des implémentations actuelles des intergiciels asynchrones qui se concentrent essentiellement sur l'optimisation des modèles de traitement (gestion d'abonnement sur différents critères par exemple) ou de délivrance (routage efficace des messages par exemple) mais ne prennent pas en compte les nouveaux besoins de l'informatique omniprésente de demain.

DREAM propose un modèle suffisamment général permettant l'adaptation de l'intergiciel à n'importe quel modèle de traitement ou de délivrance. L'ajout du modèle de contrôle (comprenant FCF modifié et les LA) permet de configurer l'intergiciel en fonction des besoins applicatifs ou des contraintes du système. Le modèle de contrôle proposé permet de gérer la configuration statique ou dynamique sans avoir à arrêter l'intergiciel, à reprendre tout le code, à recompiler et à réinstaller la nouvelle instance. Il suffit d'envoyer la nouvelle configuration (sous forme d'une description MDL) au LA du site qui se chargera de transmettre les ordres au contrôleur de configuration. La fonction de configuration de ce dernier devant être implémentée par le développeur de l'intergiciel (une implémentation de base est présente et offre déjà certains mécanismes). Le modèle de contrôle facilite le déploiement local grâce aux notions d'initialisation et de gestion de cycle de vie.

La souplesse d'utilisation est donc le principal avantage de DREAM. Néanmoins quelques points restent à approfondir comme le déploiement à grande échelle. Notre proposition basée sur l'utilisation d'entités externes à l'intergiciel (les LA) responsables notamment du déploiement hiérarchique est prometteuse mais des travaux supplémentaires seraient nécessaires sur la notion de traitement des pannes et à l'observation des LA (*monitoring*) pourtant essentielle à l'adaptabilité des intergiciels.

De plus, malgré les extensions apportées à l'ADL de Julia pour augmenter la flexibilité de configuration et de déploiement, notre MDL n'est pas encore tout à fait abouti. Le principal inconvénient du MDL est son aspect statique. Il ne permet pas de décrire la sémantique des composants applicatifs à l'exécution comme l'ADL Wright par exemple. Cela pourrait pourtant aider grandement un processus d'automatisation de configuration de l'intergiciel.

De plus, le MDL ne permet pas encore de gérer le partage de composants qui est une des fonctionnalités mise en avant par Fractal. Le partage est important car il permettrait de mettre en commun des propriétés communes à des parties différentes de l'intergiciel (par exemple la gestion de la persistance de composants). Néanmoins, cette fonctionnalité étant présente dans Julia, nous pensons que son intégration dans le MDL devrait pouvoir se faire aisément.

La flexibilité de conception, les propriétés fournies par les modèles utilisés et l'architecture générale de DREAM en font un intergiciel asynchrone atypique. Les intergiciels asynchrones existants se concentrent essentiellement sur un problème bien particulier sans pouvoir prendre en compte les changements dynamiques nécessaires à cause de leur structure monolithique. DREAM répond au besoin de « boite blanche » permettant la manipulation et l'adaptation de l'intergiciel tout au long de sa vie sur différents aspects de la plate-forme qui n'auraient pas été anticipés lors de sa création (ajout de nouvelles propriétés, ajout de nouveaux services, ajout changement dans la dissémination,...).

Nous illustrons dans le chapitre suivant les possibilités de configuration de DREAM en fournissant l'API à agent A3. nous proposons une implémentation allégée qui n'embarque pas l'ensemble

des propriétés non fonctionnelles. Cette implémentation « minimale » permet donc l'utilisation d'application A3 tout en adaptant l'intergiciel aux besoins applicatifs.

Chapitre 6

Expérimentation

Dans le chapitre précédent nous avons présenté DREAM, un modèle d'intergiciel asynchrone adaptable. DREAM est basé sur une infrastructure à composants réflexifs lui conférant des propriétés de configurabilité, de configuration et d'adaptabilité. Un des buts de DREAM est de fournir un environnement pour le développement de services asynchrones adaptables. Ce chapitre propose une implémentation d'un service à événements à base d'agents.

6.1 Introduction

Le but de ce chapitre est de montrer la faisabilité d'une implémentation d'intergiciel asynchrone utilisant les propriétés d'adaptabilité de DREAM.

Cette expérimentation propose l'implémentation d'un service à événements pouvant accueillir des applications à base d'agents de type A3 (voir section 1.3.1.4). Ce service appelé `Event-Based Service` (ou EBS) fournit un support d'exécution pour agents A3 et permet d'accueillir n'importe quelle application qui respecte le modèle à agent A3. Pour développer ce service nous avons réalisé une réingénierie de l'intergiciel A3 afin de ne conserver que les éléments importants permettant la réalisation d'une architecture simplifiée. Cette réalisation ne propose donc aucune propriété spécifique telle que la persistance, l'ordonnancement ou encore l'atomicité. L'EBS garantit le respect de l'API Agent et son implémentation à travers DREAM lui offre la souplesse de l'adaptabilité. Ainsi nous avons pu, grâce aux interfaces de contrôle, reconfigurer l'architecture interne du service pour y ajouter la propriété non fonctionnelle d'atomicité des réactions d'agents. Mécanisme qui aurait été impossible à réaliser dans l'implémentation monolithique de l'intergiciel A3.

Note : Les listings présentés dans ce chapitre sont des versions partielles du code, parfois tronqué pour plus de clarté. L'implémentation des composants principaux est donnée en annexe B.

6.2 L'Event-Based Service

L'architecture décrite dans la figure 6.1 représente la partie fonctionnelle du service. Dans le modèle A3, une application est constituée d'agents. Chaque agent est un objet réactif (passif), autonome, qui est manipulé par une entité spécifique, l'`Engine`, lui garantissant plusieurs propriétés. Ce composant, qui est le seul à possèder un processus léger (*thread*), récupère les messages qui lui sont destinés dans le tampon extérieur (en passant par une couche d'abstraction : l'`EBSMessageTranslator` qui transforme un message « DREAM » en message « A3 ») puis cherche l'agent destinataire dans un catalogue (l'`AgentsRegistry`) qui appelle le composant

`AgentReactor` responsable de faire réagir l'agent (c'est-à-dire appeler la méthode `react()` sur l'agent concerné).

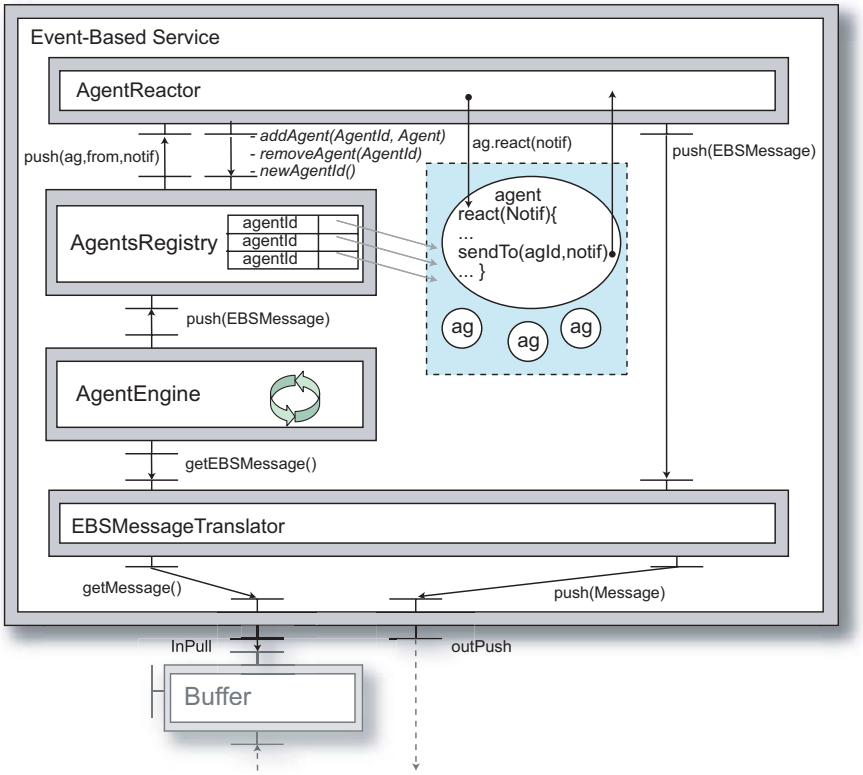

Figure 6.1 – Architecture du service `EventBasedService`.

Durant sa réaction (exécutée par l'Engine), l'agent peut envoyer des notifications à d'autres agents par l'intermédiaire de la méthode `sendTo(AgentId agId, Notification notif)`. Cette méthode transmet la notification au composant `AgentReactor`. Les notifications sont ensuite transformées en messages compréhensibles par le MOM (avec un identifiant de destinataire sous forme de `ServiceId`) grâce au composant `EBSMessageTranslator`, puis les messages sont acheminés par le MOM jusqu'au service `EventBased` concerné.

Chaque `EventBasedService` est identifié par un identifiant DREAM unique. Dans l'implémentation actuelle, les identifiants des services sont de simples entiers. Afin de conserver la compatibilité avec les applications A3 existantes, le nommage doit rester le même, à savoir que chaque service (représentant un serveur d'agent) possède un `ServerId`. Cet attribut `ServerId` est en fait un `EBServiceId` sous la forme d'un entier. Cet identifiant est utilisé par les `AgentId`[1].

Les agents utilisent des notifications pour communiquer. La transmission de ces notifications entre services EBS se fait par l'encapsulation dans un `EBSMessage` qui contient l'`AgentId` destinataire, l'`AgentId` émetteur et donc la notification (voir listing 6.1).

[1] Un `AgentId` étant constitué de 3 champs : un champ *from* site (service) de création de l'agent, un champ *to* site d'exécution de l'agent et un champ *stamp* une estampille locale unique.

Listing 6.1 – Implémentation des messages A3 `EbsMessage`.

```
public class EbsMessage implements Serializable{

  /** AgentId of sender. */
  private AgentId from;

  /** AgentId of destination agent. */
  private AgentId to;

  /** Notification The notification. */
  private Notification notif;
}
```

Les `EBSMessage` sont encapsulés dans des `Message` compréhensibles par le MOM et permettant de les router jusqu'au service destinataire. Le composant `EBSMessageTranslator` se charge d'encapsuler les messages `EBSMessage` en partance pour d'autres EBServices dans des `Message` et vice versa (voir section 6.2.4).

La description initiale MDL de ce service est donnée en annexe C.

6.2.1 Le composant `AgentEngine`

Le composant `AgentEngine` est le cœur du service EBS. C'est le seul composant qui possède un processus léger, il pilote donc l'ensemble des autres composants à la manière de l'architecture A3. L'*engine* développé pour l'`EBService` est différent de celui standard fourni par l'intergiciel A3. Dans ce dernier, l'*engine* donne plusieurs propriétés aux agents (persistance, atomicité, etc.) qui ralentissent l'exécution globale du serveur même si elle ne sont pas nécessaires à l'application. Le composant `AgentEngine` propose simplement la sémantique d'exécution selon le modèle à Agent, à savoir une exécution itérative en trois étapes :

1. Récupération dans la queue du `Message` destiné à un agent local.
2. Faire réagir l'agent destinataire.
3. Supprimer le `Message` de la queue.

L'*AgentEngine* n'inclut donc pas de propriété non fonctionnelle « en dur » dans le code mais respecte uniquement la sémantique à Agent A3. Cela permet de configurer dynamiquement l'intergiciel en lui ajoutant de nouvelles propriétés uniquement si l'application le nécessite.

Comme le montre le listing B.1, dans notre implémentation, l'`AgentEngine` ne réalise pas l'étape 2 par lui-même. Les agents étant des objets Java standards, il est nécessaire de conserver une référence de cet objet suivant son identifiant `AgentId`. Ce rôle est tenu par le composant `AgentRegistry` (voir section 6.2.3). Afin de décomposer l'exécution permettant plus de modularité, la réaction d'un agent est réalisée par un autre composant appelé `AgentReactor` (voir section 6.2.2).

L'`AgentEngine` pilote donc cette chaîne d'exécution grâce à son thread qui exécute la méthode `runningMethod()`. Il étend donc la classe `DreamThreadComponent`, qui intègre un contrôleur de cycle de vie pour composant threadé comme décrit dans la section5.2.5.2.

6.2.2 Le composant `AgentReactor`

Le composant `AgentReactor` fournit l'API Agent. D'une certaine façon, l'`AgentReactor` peut être vu comme une couche d'abstraction entre le « monde composant » de DREAM et le « monde objet » des agents.

Il est responsable de la réaction des agents. Il reçoit de l'`AgentRegistry` la référence de l'agent à faire réagir, la notification et l'identifiant de l'émetteur par l'intermédiaire de l'interface `AgentPush` (voir listing 6.2). Grâce à cela il appelle, sur l'agent, la méthode `react` avec l'identifiant de l'agent émetteur et la notification.

Chaque agent étend la classe Java `Agent`. C'est par cet intermédiaire que l'`AgentReactor` intercepte les envois de messages des agents en cours de réaction. Chaque fois qu'un agent `X` désire (au cours de sa réaction) envoyer des messages, il utilise la méthode `sendTo`. Cette méthode est appelée sur la classe `Agent` puis sur l'`AgentReactor` qui crée un message `EBSMessage` qui comporte l'identifiant de l'agent émetteur (celui qui est en train de réagir), l'identifiant de l'agent destinataire et la notification associée.

Listing 6.2 – Interface du composant `AgentReactor`.

```
public interface AgentPush {
  /**
   * push the message to agentReactor
   * @param agent is the Agent object wich receive the notification
   * @param from is the AgentId of the sender Agent.
   * @param not is the notification
   */
  public void push(Agent agent, AgentId from,
    Notification not) throws Exception;
}
```

Ce message est ensuite transmis au composant `EBSMessageTranslator` par l'interface `EBSPush` qui se charge de l'encapsuler dans un message standard DREAM en donnant l'identifiant du service destinataire (voir section 6.2.4).

L'ensemble du mécanisme est décrit dans la figure 6.2.

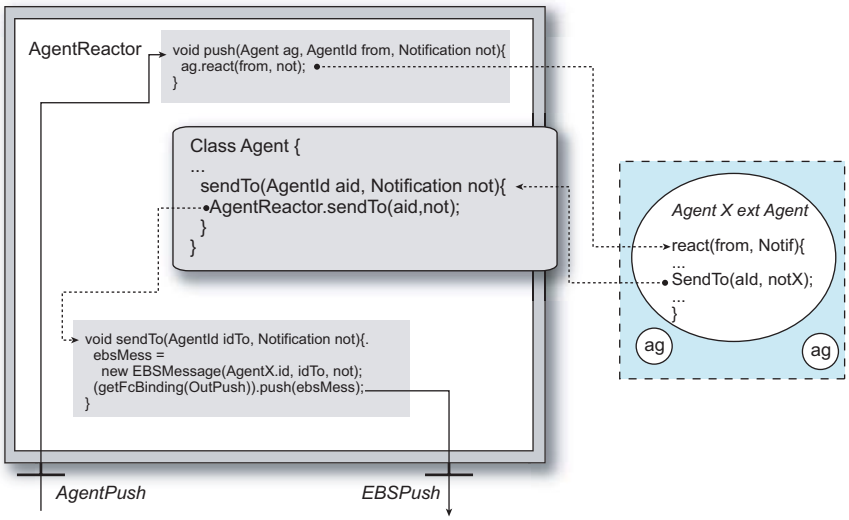

Figure 6.2 – Principe de réaction des agents par l'intermédiaire de l'`AgentReactor`.

6.2.3 Le composant `AgentRegistry`

L'`AgentRegistry` représente la base de donnée des agents locaux. Il possède une table de hashage qui fait la correspondance entre un `AgentId` et la référence sur l'objet Java Agent. L'`AgentEngine` passe les messages, destinés aux agents, à l'`AgentRegistry` par l'intermédiaire de l'interface `EBSMessagePush` (voir listing 6.3).

Listing 6.3 – Interface Push du composant `AgentReactor`.

```
public interface EBSMessagePush {
  /**
   * push the message to agentReactor
   * @param ebsMess the EBSmessage wich contain the local AgentId
   */
  public void push(EBSMessage ebsMess) throws Exception;
}
```

Mais l'`AgentRegistry` propose aussi une autre interface serveur destinée à l'administration des agents (création, suppression, etc.). Dans le modèle à Agent, chaque agent est créé par un agent spécifique appelé `AgentFactory`. Ce dernier est un agent comme les autres qui reçoit des messages plus spécifiques à sa fonction, à savoir des messages de création d'agent (donc création d'un nouvel `AgentId`) et de suppression d'agent. Il existe toujours une instance de l'`AgentFactory` dans un EBService, il possède, à ce titre, un identifiant spécifique (`AgentId` = *#serveurId.#serveurId.1*). L'interface `A3Admin` propose donc un ensemble de méthodes permettant à l'`AgentRegistry` de gérer ces manipulations d'agents (voir listing 6.4).

Listing 6.4 – Interface d'administration des agents de l'`AgentRegistry`.

```
public interface A3Admin {
  /**
   * addagent adds a new agent into the Event-based service
   * @param agentId the AgentId of the new Agent
   * @param agent the object Agent to add into the Agent's table
   */
  public void addAgent(AgentId agentId, Agent agent);

  /**
   * removeAgent removes (deletes) a agent of the Event-based service
   * @param agentId the AgentId of the Agent to remove (delete)
   */
  public void removeAgent(AgentId agentId) throws UnknownAgentException;

  /**
   * newAgentId creates a new agentId (UNIQUE for all Event-based services)
   * @param eventBasedServiceId the EBS Id on which the Agent is deployed
   */
  public AgentId newAgentId(EBServiceId eventBasedServiceId);
}
```

Lorsqu'un agent désire créer un autre agent il créer l'objet Agent normalement (`new Agent(...)`) et récupère l'`AgentId` de l'agent ainsi créé. Pour cela la méthode de construction de la classe `Agent` appelle (par l'intermédiaire de l'`AgentReactor`) la méthode de création d'un nouvel `AgentId` à l'`AgentRegistry` qui est le seul habilité à gérer les `AgentId`. Par la suite, lorsque le nouvel agent doit être déployé (c'est-à-dire mis en place sur son service d'exécution puis démarré), cela créé un notification de création pour l'`AgentFactory` destinataire. La notification

est constituée de l'objet Agent sérialisé. Ce processus[2] est illustré sur la figure 6.3.

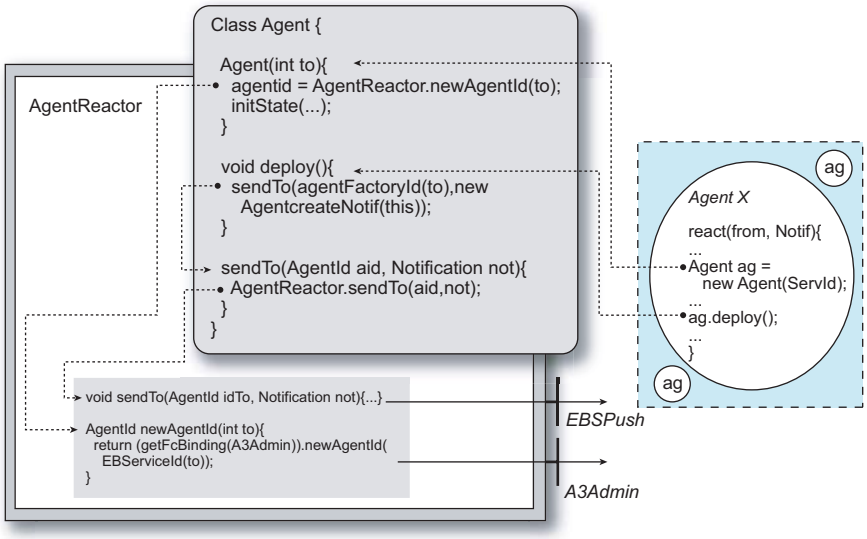

Figure 6.3 – Processus de création d'agent (étape 1).

Lorsque la notification arrive à l'`AgentFactory`, celui-ci lance la création de l'agent contenu dans la notification par l'intermédiaire de deux méthodes de l'`AgentReactor`. La méthode de création (resp. suppression) de l'`AgentReactor` est relayée à l'interface A3Admin de l'`AgentRegistry`. Celui-ci enregistre (resp. supprime) l'entrée de l'agent dans sa table locale. Ce processus est illustré sur la figure 6.4.

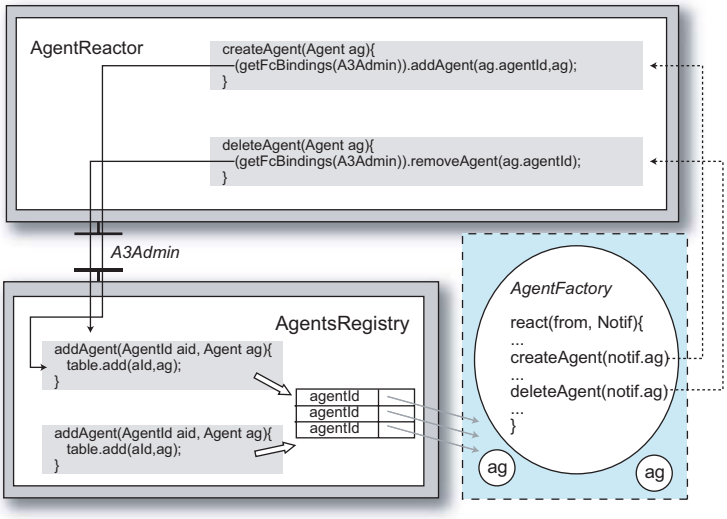

Figure 6.4 – Processus de création d'agent : l'`AgentFactory` (étape 2).

[2] Ce processus suit exactement le fonctionnement de l'API Agent et le principe de création d'agent par l'`AgentFactory`.

6.2.4 Le composant `EBSMessageTranslator`

Le composant `EBSMessageTranslator` (`EBSMT`) a pour but de faire l'interface entre les messages de « type DREAM » reconnus par le MOM et pouvant circuler entre les différents sites et les messages de type EBS c'est-à-dire les messages comportant des destinataires sous forme d'`AgentId`.

Lorsque l'`AgentEngine` récupère un message (étape 1), il appelle la méthode *getMessage()* sur l'`EBSMessageTranslator` qui va lui-même le chercher dans le tampon externe du service EBS (service DREAM de type *Pull*). L'`EBSMT` récupère donc un message DREAM qui encapsule un message EBS (à la manière des encapsulations de paquets dans les protocoles réseaux). Il retourne donc un `EBSMessage` à l'`AgentEngine` qui continue son exécution.

Le rôle le plus important de `EBSMT` se déroule lors de l'opération inverse. Lorsque l'`AgentReactor` passe à l'`EBSMT` un message EBS à envoyer, celui-ci doit l'encapsuler dans un message DREAM standard comportant l'identifiant du service destinataire en fonction de l'agent destinataire. Il doit donc retrouver l'identifiant du service DREAM de type EBS qui possède l'`AgentId` destinataire. Dans l'implémentation actuelle, les services DREAM sont identifiés par des nombres entiers et l'identifiant d'un service EBS est donc un nombre entier. Ceci simplifie l'interaction avec les `AgentId` qui utilisent des nombres entiers pour identifier leur site d'exécution (champ *to*). Ainsi, pour connaître l'identifiant du service destinataire, il suffit de regarder le champ *to* de l'`AgentId` destinataire.

6.3 Le MOM

L'implémentation du MOM utilisé avec le service EBS est une implémentation standard comme décrite dans le chapitre précédent (section 5.3). Nous avons utilisé un seul composant `Network` sans propriété non fonctionnelle spécifique.

Le MOM permet de transmettre des `Message` ayant un format standard : un champ *from* donnant l'identifiant du service émetteur, un champ *to* donnant l'identifiant du service destinataire et les données relatives au service destinataire. Dans notre cas restreint d'utilisation de services de type unique, les données seront un simple `EBSMessage`. Comme décrit dans la section précédente, les identifiants de services sont actuellement de simples nombres entiers.

Cette implémentation des identifiants de service limite grandement les possibilités de routage plus complexe par le MOM. Il pourrait être plus intéressant d'utiliser un nommage hiérarchique dans `Dream` comme celle proposée par le processus de déploiement en domaines de configuration. L'utilisation de `EBSMessageTranslator` prendrait alors tout son sens car il permettrait de dissocier le nommage applicatif du nommage au niveau intergiciel du MOM.

Ainsi, le composant `Network` possède la correspondance entre les `ServiceId` et leur couple @IP/numéro de port. Cette connaissance est inscrite de façon statique dans le composant mais il est tout à fait possible de mettre à jours ces données via l'utilisation de le contrôleur de configuration (mise à jour de attribut *table de routage* grâce au contrôleur d'attribut).

6.4 Reconfiguration dynamique

L'architecture du service EBS de DREAM a été réalisée dans le but de permettre l'ajout dynamique de propriétés non fonctionnelles. Afin d'illustrer une reconfiguration de l'architecture nous avons ajouté au service EBS un composant responsable de l'atomicité des agents. Pour cela, nous

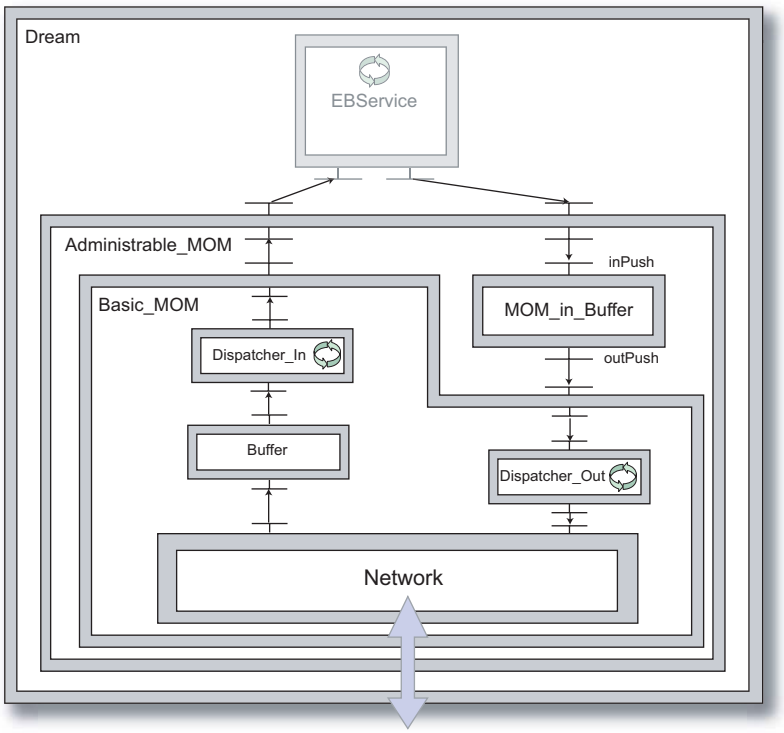

Figure 6.5 – Le MOM implémenté pour cette expérimentation.

avons utilisé l'implémentation de base du contrôleur de configuration à qui nous avons avons passé la nouvelle architecture du service décrite à l'aide du MDL (description donnée dans l'annexeC.2). Le contrôleur de configuration arrête donc le service en suivant l'ordre d'arrêt décrit dans le MDL initial puis met à jour la nouvelle configuration en ajoutant le nouveau composant[3].

6.4.1 Ajout de la propriété d'atomicité des agents

Notre exemple ajoute la propriété d'atomicité des réactions d'agent. Le composant, appelé `AgentsTransaction`[4], garantit que si une erreur se produit durant l'exécution de la réaction d'un agent celle-ci sera défaite, l'agent retournera dans son état précédent et toutes les notifications envoyées au cours de cette réaction seront supprimées.

Le composant est donc tout simplement inséré entre l'`AgentRegistry` et l'`AgentReactor` d'où il pourra intercepter les notifications. Une fois ajouté au composite `EBService`, les interfaces `AgentPush` et `EBSPush` sont déliées puis l'`AgentsTransaction` est lié à chacune de ces interfaces. La figure 6.6 illustre la nouvelle architecture.

Ainsi lorsque l'`AgentEngine` appelle la méthode *push()* sur `AgentRegistry` puis que celui-ci appelle la méthode sur son interface `AgentPush`, c'est la méthode *push()* de `AgentsTransaction` qui est donc appelée. Dans cette méthode l'`AgentsTransaction` réalise l'appel sur l'`AgentReactor` (= l'appel sur son interface cliente `AgentPush`) à travers un bloc de traitement d'exception[5]. Durant cette réaction l'agent peut envoyer des messages par l'intermédiaire

[3] Nous supposons que la classe du nouveau composant est disponible localement mais rien n'empêcherait l'utilisation d'un chargeur de classe à distance.

[4] L'implémentation de l'`AgentsTransaction` est donnée dans l'annexe B.4.

[5] Bloc « *try*{} *catch()*{} » en Java.

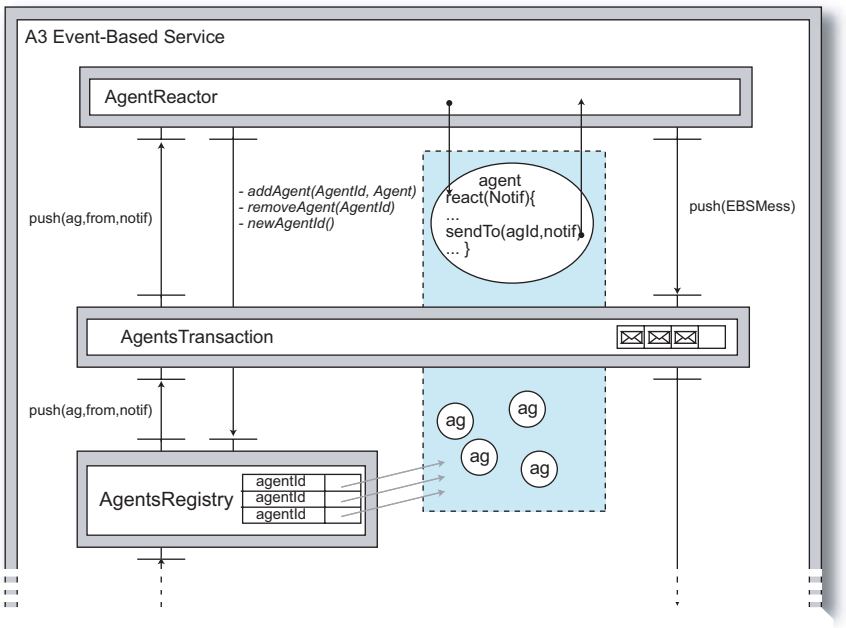

Figure 6.6 – Ajout du composant `AgentsTransaction` à l'EBS.

de l'`AgentReactor` via l'interface `EBSPush`. L'`AgentsTransaction` étant relié à cette interface, il intercepte tous les messages EBS envoyés par l'agent. Ainsi, si la réaction se réalise sans erreur (sans exception), l'agent est sauvegardé sur disque puis tous les messages provisoirement stockés sont envoyés (appel de la méthode *push(EBSMessage)* sur l'`EBSMessageTranslator` via l'interface `EBSPush` à laquelle il est lié). Si une erreur se produit durant la réaction, l'agent est rechargé depuis son image sur disque, les messages stockés sont effacés puis une notification d'erreur est envoyée à l'agent émetteur de la notification.

La reconfiguration par l'ajout de ce composant a été testée avec succès. L'implémentation du service EBS à l'aide de composant DREAM permet donc d'ajouter « à la volée » des composants responsable de propriétés non fonctionnelles et cela sans perturber l'exécution des autres services ou des sites distants. Ce composant peut, de la même manière, être enlevé sans que cela impose un arrêt complet de l'intergiciel.

6.5 Synthèse sur l'expérimentation

Cette expérimentation à permis de montrer la faisabilité d'une implémentation d'intergiciel asynchrone à l'aide de DREAM. Nous avons expérimenté l'utilisation de l'API A3 fournie par l'EBS sur de petites applications à base d'agents. Dans le modèle A3 il existe une implémentation spécifique d'agent appelée `AgentProxy` qui possède un processus léger et peut communiquer avec l'extérieur. Nous n'avons pas développé ces agents qui sont pourtant utilisés dans de nombreuses applications A3. Nous pensons qu'il serait très intéressant de transformer ces agents en service DREAM à part entière. Cela permettrait d'augmenter le parallélisme d'exécution et donc les performances et d'attribuer des propriétés uniquement à ces agents très spécifiques.

Nous n'avons pas utilisé totalement le potentiel de configurabilité de DREAM pour cette implémentation par manque de temps. Beaucoup de choses auraient pu être réalisées de façon plus

poussée comme le nommage ou l'utilisation de multiples personnalités en parallèle. La solution du nommage (utilisation de nombre entier) n'est pas satisfaisante car elle limite la dynamicité et la scalabilité de l'intergiciel (difficulté de rajouter de nouveaux sites automatiquement, routage des messages statique, etc.). Une solution basée sur des identifiants hiérarchiques serait sans doute plus adaptée à notre intergiciel adaptable[6].

Néanmoins, le service EBS a été implémenté et a permis démontrer la faisabilité d'une personnalité à Agent sur DREAM. Notre implémentation allégée du modèle A3 n'embarque pas l'ensemble des propriétés non fonctionnelles standards de l'intergiciel A3. Il est possible d'ajouter les propriétés de façon dynamique et cela sans perturber l'exécution de l'ensemble de l'intergiciel. Cette implémentation « minimale » permet donc l'utilisation d'applications A3 tout en adaptant l'intergiciel aux besoins applicatifs. De la même manière, la flexibilité de configuration du MOM permet de rajouter (sur certains sites) un ordonnancement causal des messages. Une combinaison avec nos travaux sur la causalité par transitivité pourrait permettre d'adapter l'intergiciel aux contraintes de certains environnements tout en réutilisant les mêmes composants.

6.6 Analyse et perspectives du passage à grande échelle

Dans les sections précédentes, nous avons présenté une implémentation de DREAM qui fournit la possibilité d'exécuter des applications respectant l'API Agent A3. Nous avons démontré la faisabilité d'un intergiciel asynchrone adaptable avec notre modèle. Mais nous n'avons pas abordé la notion de déploiement et de configuration distribuée à grande échelle. En effet, le processus proposé dans la section 5.6 n'a pas pu être mis en oeuvre par manque de temps. Et bien que les concepts proposés soient très prometteurs, ils nous semblent incomplets et mériteraient d'être plus approfondis.

Cette section tente donc de dresser une étude qualitative de notre proposition de processus. Nous abordons les points qui mériteraient d'être plus aboutis et nous proposons des pistes pour le futur.

6.6.1 Initialisation (*bootstrap*)

Dans notre proposition, nous partons d'une hypothèse simple qui est l'omniprésence des LA. Ainsi, il est nécessaire que tous les sites qui veulent recevoir un « bout » d'intergiciel possèdent un LA.

Minimalité Cette hypothèse peut paraître très restrictive car elle impose la présence d'un « bout de code » avant tout déploiement. Cela pose deux problèmes :

1. Comment le LA est-t-il lui-même mis en place sur ce site ?
2. La complexité du processus rend le LA relativement coûteux en termes de place mémoire et de temps processeur.

Le premier point est clairement un problème récurrent qui n'a pas de solution évidente et idéale. On pourrait, par exemple, imaginer qu'il y a un processus léger présent comme service système qui permettrait de charger en mémoire des LA. Mais se pose alors la question du déploiement des LA qui nécessiterait un processus de déploiement, etc. ! Il est évident que l'on peut toujours aller plus loin dans la minimalité des mécanismes présents mais cela ne fait que reporter le problème. C'est pourquoi nous pensons que notre solution à base de LA sur chaque site est un bon compromis entre minimalité de service de déploiement (c'est-à-dire un service permettant, au moins, d'instancier un

[6] Des travaux sur la hiérarchisation du nommage, en corrélation avec les domaines de configuration, sont en cours d'élaboration mais ne sont pas assez aboutis pour être exposés ici.

intergiciel localement) et support scalable de déploiement (c'est-à-dire un service qui peut répondre aux problèmes de grande échelle, de gestion de pannes, etc.).

Le deuxième point soulevé est le coût d'un tel processus sur les performances du site de déploiement. La réponse la plus simple à cette question est qu'il n'est pas possible de fournir une solution qui soit à la fois fiable, performante et « légère ». Il est donc nécessaire de faire un choix sur l'importance d'un processus de déploiement de l'intergiciel et ce choix doit être fait par l'administrateur du système.

LA, domaines et connaissance globale L'initialisation des LA est réalisée avec l'hypothèse qu'ils se connaissent tous, c'est-à-dire que chaque LA est capable d'envoyer un message à n'importe quel LA. Il est évident que cette hypothèse nuit grandement à la dynamicité du processus de déploiement. Il est en effet impossible de rajouter un site, dans l'ensemble des sites déjà présents, qui n'aurait pas été préalablement déclarés.

La solution pourrait être d'utiliser les domaines comme groupes de sites dynamiques. Au lieu de découper arbitrairement les domaines comme c'est le cas actuellement, il serait plus pertinent d'utiliser la topologie du réseau en déléguant la connaissance à une tête de pont par exemple. Il serait intéressant de réaliser ce découpage à l'aide d'un algorithme automatisé qui prendrait en paramètre la répartition géographique des sites ou les propriétés des sites par exemple. De la même manière, le LA-maître est choisi de façon arbitraire, il s'agit du premier LA rencontré dans la description globale. Mais le choix du LA-maître pourrait tout à fait être dicté par des choix topologiques (architecture du réseau par exemple), physiques (puissance de la machine par exemple) ou logiques (en fonction des applications par exemple) et définis par l'initiateur.

6.6.2 Liaisons

Dans notre processus, les liaisons sont décrites dans un fichier MDL global défini (et donné) par l'initiateur. Cette description permet au LA maître de chaque domaine de connaître précisément quel site communique avec quel autre. L'information permet un meilleur traitement des pannes car il est possible de savoir quelles sont les liaisons manquantes en cas de problème et laisse la décision à l'initiateur de prendre une décision (activation avec des liaisons partielles, activation locale de certains domaines, etc.). De plus, il est possible d'envoyer un ordre d'activation même si toutes les liaisons n'ont pas été réalisées, permettant une disponibilité rapide de l'intergiciel. Mais cette fonctionnalité peut créer des problèmes de cohérence de l'intergiciel, si une liaison est manquante alors que l'intergiciel en avait absolument besoin pour fonctionner correctement.

La description de configuration pourrait donc être étoffée d'une notion de liaisons *requises*, c'est-à-dire des liaisons qui sont indispensables au bon fonctionnement de l'intergiciel. Cette notion permet d'ajouter une sorte de connaissance sémantique de l'intergiciel en imposant des liaisons qui sont sémantiquement indispensables à l'exécution globale.

Liaisons complexes Une restriction qui serait intéressante de lever est la liaison entre les différents intergiciels. Actuellement, notre processus se limite à l'utilisation du protocole TCP pour lier deux sites. Cela a l'avantage de simplifier la gestion des liaisons car il suffit de donner le port de connexion TCP au `Network` client pour qu'il se lie au `Network` serveur.

L'utilisation de liaisons élaborées comme celles fournies par le canevas logiciel Jonathan [8] permettrait de décrire non seulement les liens entre les participants mais aussi le type de protocole de communication utilisé. Par exemple, on pourrait signifier que la liaison entre deux sites est sécurisée par tel ou tel protocole. Cela est réalisé dans les descriptions d'applications comme nous l'avons

vu dans le chapitre 3 et mis en oeuvre dans le déploiement applicatif de la société Scalagent (voir section 2.3.2.1).

6.6.3 Gestion des pannes

Le processus de déploiement à grande échelle est une tâche complexe qui est sujette à des pannes, à des sites déconnectés, des blocages, etc. Il est donc essentiel de pouvoir détecter les pannes et de pouvoir y réagir pour éviter un blocage total et complet de l'ensemble des sites. Dans cette section nous présentons des débuts de réponses à la gestion des pannes en nous basant sur les travaux réalisés par la société Scalagent.

Détection Dans le processus proposé par Scalagent, la détection des pannes agit comme un contrôle de *workflow* qui traite les éventuelles erreurs durant le processus de déploiement. Ces erreurs sont alors notifiées à l'initiateur du déploiement.

Nous considérons qu'il existe deux pannes possibles. Soit une activité est bloquée comme par exemple lorsqu'une création ne se réalise pas ou lorsqu'une liaison ne parvient pas à son terme. Soit un LA tombe en panne, parce que son site d'exécution a subi une panne fatale par exemple.

Il existe deux possibilités de détection de pannes. La première consiste à borner le temps d'exécution d'une activité et à déclencher des erreurs lorsque le temps s'est écoulé. Il est évident que cette solution n'est pas viable dans le cadre de déploiement à très grande échelle. Le nombre d'activités à surveiller, les temps de latence des réseaux, la cohérence du bornage, etc. autant de paramètres qui rendent cette solution non applicable à grande échelle.

Une autre solution consiste à se baser sur une stratégie *optimiste*. Elle consiste à ne pas remonter de messages d'erreur mais à agréger et à remonter les événements d'observation permettant d'avoir, au niveau de l'initiateur, une connaissance détaillée de l'état du déploiement. Cette connaissance peut être utilisée par l'initiateur pour traiter ces pannes.

Traitement des pannes Le traitement des pannes par l'initiateur consiste à réaliser des *reconfigurations* de la configuration d'intergiciels déployée. L'initiateur peut prendre la décision d'arrêter certains groupes de sites afin de leur envoyer une nouvelle configuration. De plus, les possibilités de reconfiguration partielle de DREAM permettraient de mettre à jour le MOM sans pour autant empêcher les services de s'exécuter. La possibilité d'activation des sites avant configuration complète favorise la disponibilité des services.

Perspectives Nous pensons que la configuration et le déploiement à grande échelle doit se tourner vers une hiérarchisation, non seulement des mécanismes comme c'est le cas dans notre solution, mais aussi de « l'intelligence ». Ainsi, il serait plus rapide et plus efficace que les LA-maîtres de domaine prennent des décisions de reconfiguration sans pour autant faire nécessairement remonter des informations et attendre une décision « d'en haut ». Néanmoins, de nombreux problèmes se posent quant à la reconfiguration des sites, car un changement sur un site peut avoir des conséquences tout un ensemble de sites et les décisions locales peuvent se révéler catastrophiques pour l'ensemble des sites. Cette hiérarchisation de l'intelligence de traitement nous semble donc une des grandes voies à suivre dans la continuité de ce travail.

Conclusion

Objectifs de la thèse

Cette thèse porte sur la configuration et le déploiement des intergiciels asynchrones sur système hétérogène à grande échelle. Notre objectif a été de fournir une infrastructure d'intergiciel asynchrone adaptable permettant de répondre aux nouveaux défis de l'informatique.

Notre motivation principale pour travailler sur ce problème a été la diversification des plates-formes matérielles et logicielles, amenée par les avancées importantes des technologies et des réseaux. Notre travail se situe notamment dans le cadre d'une croissance de plus en plus grande de l'*informatique omniprésente* dans laquelle pléthores de périphériques hétérogènes et mobiles communiquent entre eux au sein de multiples applications.

En effet, un intergiciel asynchrone configurable permet d'offrir un comportement fonctionnel de base sur lequel peuvent se greffer des extensions et donc répondre aux contraintes spécifiques de son environnement. La configuration doit être réalisée de façon statique au démarrage de l'intergiciel ou dynamiquement en cours d'exécution afin de répondre aux changements inhérents du système (arrivée de nouveaux clients, mobilité du périphérique, etc.).

Approche

Pour fournir un intergiciel asynchrone adaptable, nous nous sommes basés sur les principes de configurabilité et de configuration des intergiciels synchrones et plus précisément des ORB réflexifs.

Nous avons proposé un modèle d'intergiciel asynchrone utilisant des techniques de réflexivité associées à une architecture à composants qui est la plus à même de répondre aux besoins d'adaptation. De plus, nous ajoutons aux mécanismes de contrôle réflexifs au niveau des composants (*notion de contrôleur*), un contrôle à plus gros grain (*architecture des LA*) permettant de gérer systématiquement le déploiement de l'intergiciel local et global.

Les intergiciels asynchrones actuels se concentrent souvent sur un modèle de traitement de messages ou sur l'optimisation de la dissémination et de la délivrance des messages. L'originalité de notre approche est de proposer un intergiciel asynchrone qui permet au développeur de réaliser n'importe quel modèle de délivrance et de traitement tout en se servant d'un modèle de contrôle (extensible) qui facilite la configuration.

Réalisation

Nous avons implémenté un prototype en Java, nommé DREAM (***D**ynamic **RE**flective **A**synchronous **M**iddleware*) qui met en œuvre les principes proposés. Notre architecture se décompose en 3 entités :

– Le composant MOM représente la partie communication de DREAM. Il offre des interfaces de communication asynchrones simplifiées aux services et prend en charge le routage des messages. L'architecture interne du MOM permet la configuration de plusieurs modèles de délivrance en même temps (par l'utilisation de multiples composants `Network`).
– Les services sont des supports d'exécution asynchrones qui utilisent le composant MOM pour envoyer et recevoir des messages. Une application distribuée est donc composée d'un ensemble de services répartis sur différents sites. L'utilisation de multiples services indépendants les uns des autres permet une configuration « au plus près » des besoins des applications les utilisant.
– Le composant local d'administration (le LA) est présent quelle que soit la configuration de l'intergiciel. Il utilise les mécanismes de contrôle pour configurer l'intergiciel. Le LA peut être vu comme un composant de *bootstrap* de l'intergiciel. Il doit être présent sur chaque site pour pouvoir déployer et configurer l'intergiciel.

Chacune de ces entités utilise notre modèle de contrôle qui se sert du modèle à composants réflexifs Fractal (à travers l'implémentation Julia). Le modèle de contrôle est basé sur la combinaison de plusieurs contrôleurs permettant la configuration de l'intergiciel. Le *contrôleur de cycle de vie* et le *contrôleur de configuration* sont les deux principaux contrôleurs.

Le premier gère le démarrage et l'arrêt des composants pour la configuration. L'implémentation de base de ce contrôleur propose une solution pour arrêter et démarrer les composants en évitant les interblocages et donc garantit la disponibilité du reste de l'intergiciel.

Le contrôleur de configuration propose trois fonctions nécessaires à la configuration et au déploiement (création, initialisation et configuration). Chacune de ces fonctions est couplée à un langage de description d'architecture pour l'intergiciel appelé MDL (***M**iddleware **D**escription **L**anguage*).

Notre langage MDL est un outil qui répond au besoin de flexibilité de configuration et d'automatisation du déploiement. Son utilisation apporte une vision globale de la configuration de l'intergiciel et permet de spécifier son architecture à partir d'une bibliothèque de composants. De plus, l'association du contrôleur avec le MDL permet d'éviter la configuration « d'un bloc » de l'intergiciel, dont l'architecture peut évoluer par « bouts » (possibilité de changer uniquement un composant sans pour autant arrêter puis modifier l'ensemble de la configuration).

Évaluation

Nous avons expérimenté DREAM en implémentant un service à événements pouvant accueillir des applications à base d'agents de type A3. Cette expérimentation à permis de montrer la faisabilité d'une implémentation d'intergiciel asynchrone à l'aide de DREAM.

Différents problèmes ont été soulevés dans ce document. Par cette expérimentation, nous avons tenté de répondre aux besoins de *configurabilité*, *configuration* et de *déploiement*.

Configurabilité DREAM utilise un modèle à composants autorisant la composition qui permet la construction d'un intergiciel asynchrone comme un ensemble structuré de composants. La possibilité de créer l'intergiciel comme une interconnexion d'autres composants offre un haut degré de configurabilité. De plus l'utilisation du MDL apporte une vision globale de la configuration et permet de spécifier l'architecture à partir d'une bibliothèque de composants.

Configuration Le modèle de contrôle de DREAM permet de configurer l'intergiciel en fonction des besoins applicatifs ou des contraintes du système. Le modèle proposé permet de gérer la configuration

statique ou dynamique sans avoir à arrêter l'intergiciel, reprendre tout le code, recompiler et réinstaller la nouvelle instance. Il suffit d'envoyer la nouvelle configuration (sous forme d'un MDL) au LA du site qui se chargera de transmettre les ordres au contrôleur de configuration. De plus, la gestion de cycle de vie proposée par la combinaison entre la description du MDL et le contrôleur de cycle de vie garantit la disponibilité de l'intergiciel.

La réingénierie de l'intergiciel A3 a montré qu'il était possible d'exécuter le modèle à agent à l'aide d'un intergiciel asynchrone minimal, c'est à dire qui fournit uniquement l'API de base sans embarquer pléthore de propriétés non fonctionnelles inutiles à l'application. Il a été aussi possible de rajouter dynamiquement un composant responsable de la gestion de l'atomicité des agents. Cette tâche a été rendue possible par l'utilisation des fonctionnalités de configuration de DREAM.

Déploiement L'architecture hiérarchique des composants permet une meilleure structuration de l'intergiciel et simplifie le déploiement local de chaque composant. Ce déploiement local est grandement simplifié grâce au contrôleur de configuration à qui il suffit de donner une configuration d'intergiciel sous forme d'un MDL dans lequel les ordres d'initialisation sont inscrits.

De plus, le MDL global associé à la hiérarchisation en domaines des LA permet d'éviter la création de goulot d'étranglement et facilite le passage à grande échelle. Notre proposition de processus permet de gérer le déploiement en découpant l'ensemble des sites en plusieurs sous ensembles appelés domaines. Chacun des domaines est « piloté » par un LA-maître qui a la charge de mettre en place la configuration pour son domaine et de déléguer la configuration de ses sous domaines aux LA-maîtres de niveau inférieur. L'exécution parallèle et asynchrone des différentes activités garantit une disponibilité rapide de l'intergiciel.

DREAM apporte donc du sang neuf dans le domaine des intergiciels asynchrones. Son architecture adaptable et l'implémentation proposée se démarque des autres projets existants et le prépare à l'émergence de l'informatique ubiquitaire.

Perspectives

Ce travail est un des premiers efforts sur la création d'un intergiciel asynchrone adaptable et de ce fait reste encore très incomplet. En effet, dans la réalisation de DREAM, nous avons exploré de nombreuses voies mais nous n'avons pas pu les approfondir. Deux voies nous semblent désormais essentielles à creuser.

La première est la configuration *automatisée* en fonction des besoins applicatifs et des contraintes du système. L'idée serait d'utiliser la description de l'architecture de l'application dans laquelle sont, en outre, spécifiéesr les propriétés non fonctionnelles requises. L'outil déduirait la configuration optimale de l'intergiciel asynchrone sous la forme d'un MDL. Les mécanismes de configuration fournis par Dream seraient ensuite mis à contribution pour la mise en place de l'intergiciel.

Le déploiement de l'intergiciel est la deuxième grande voie d'exploration dont les intergiciels asynchrones pourraient bénéficier. Nos travaux sur le déploiement n'en sont qu'au balbutiement. Mais notre proposition de hiérarchisation en domaines d'intergiciels nous semble très prometteuse car elle permet de déléguer les mécanismes et donc de rendre le processus scalable. Nous pensons que l'une des grandes voies à suivre dans la continuité de ce travail est la hiérarchisation du contrôle à grande échelle. Ainsi, déléguer les prises de décision, comme sont délégués les traitements, favoriserait le passage à grande échelle.

Globalement, nous pensons que c'est vers l'aide au développement que les intergiciels doivent se tourner. Les mécanismes de configuration sont de plus en plus aboutis et DREAM a permis d'ouvrir

la brèche pour les intergiciels asynchrones. Ces mécanismes permettront de concentrer les efforts sur la mise en place d'outils intelligents hiérarchiques qui les utiliseront.

Annexes

Annexe A

Preuve du théorème sur les domaines de causalité

Cette annexe présente les preuves des lemmes et du théorème proposés dans la section4.2.4, elles utilisent les définitions données dans cette même section. Une version en anglais de cette preuve est disponible dans notre article présenté à la conférence MIDDLEWARE 2001 [59].

A.1 Lemmes

Lemme 5 *Aucun domaine ne contient la source et la destination d'un chemin minimal de longueur supérieure ou égale à 3.*

Preuve : c'est une conséquence immédiate de la condition $i+1<j \Rightarrow \neg(\exists d \mid p_i \in d \wedge p_j \in d)$, qui peut s'appliquer dès que le chemin contient au moins 3 processus.

Lemme 6 *Si $(m_1,...,m_k)$ est une chaîne dont la source p et la destination q sont différentes, et si la trace est* correcte*, alors il existe une chaîne directe $(n_1,...,n_l)$ ayant même source et même destination, et telle que $m_1 \leq_p n_1$ et $n_l \leq_q m_k$.*

Preuve : Si $k=1$, le résultat est évident. Supposons par récurrence que le résultat est vrai pour les chaînes de longueur $k<K$, et considérons une chaîne $(m_1,...,m_K)$ de longueur K. Si cette chaîne est directe, le résultat est démontré. Si par contre elle ne l'est pas, alors, par définition, le chemin associé $(p_1,...,p_{K+1})$ n'est pas direct, c'est-à-dire qu'il existe $i<j$ tel que $p_i=p_j$. Considérons alors la chaîne $(n_1,...,n_l)$ définie par[1] :

- $(m_j,...,m_K)$ si $i=1 \wedge j<K+1$
- $(m_1,...,m_{i-1})$ si $i>1 \wedge j=K+1$
- $(m_1,...,m_{i-1},m_j,...,m_K)$ si $i>1 \wedge j<K+1$

C'est bien une chaîne. De plus, elle a même source et même destination que la chaîne de départ. Enfin, elle est de longueur strictement inférieure à K. On peut donc, en utilisant l'hypothèse de récurrence, en déduire l'existence d'une chaîne directe $(n'_1,...,n'_h)$ ayant même source et même destination, et telle que $n_1 \leq_p n'_1$ et $n'_h \leq_q n_l$.

De plus, $m_1 \leq_p n_1$ et $n_l \leq_q m_k$. En effet, dans le premier cas, $m_1 <_p m_j$ (sinon la chaîne définie par $(m_1,...,m_{j-1})$, "reçue" par p avant d'être "émise" par p, violerait l'hypothèse que la trace est correcte). De même, dans le second cas, $m_{i-1} <_q m_K$. Donc, dans tous les cas, $m_1 \leq_p n_1$ et $n_l \leq_q m_k$.

[1] le cas $i=1 \wedge j=K+1$ est impossible (car $p \neq q$).

On en déduit que $m_1 \leq_p n'_1$ et $n'_h \leq_q m_K$. Le lemme est donc démontré pour les chaînes de longueur K, et donc, par récurrence, pour toutes les chaînes.

Lemme 7 *Si m dépend causalement de n, alors soit il existe une chaîne $(n, ..., m)$, soit il existe une chaîne $(l, ..., m)$ où l est un message émis après n par le processus ayant émis n.*

La preuve de ce lemme se fait par induction sur les trois cas possibles de dépendance causale entre deux messages (cf définition 4) :

- soit m et n sont émis par un même processus p, et m est émis après n. Dans ce cas, il existe une chaîne de type $(l, ..., m)$ où l est un message émis après n (il suffit de prendre la chaîne (m)).
- soit m est émis par un processus ayant reçu précédemment n. Il existe alors une chaîne de type $(n, ..., m)$: il suffit de prendre la chaîne (n, m).
- soit enfin il existe un message k tel que $n \prec k$ et $k \prec m$. En appliquant l'hypothèse d'induction, on obtient quatre cas possibles, et on trouve dans tous les cas qu'il existe soit une chaine $(n, ..., m)$, soit une chaîne $(l, ..., m)$ où l est un message émis après n par le processus ayant émis n.

Lemme 8 *Si une trace correcte ne respecte pas la causalité, alors il existe deux processus p et q, un message n de p vers q, et une chaîne $(m_1, ..., m_n)$ de p à q telle que $n <_p m_1$ et $m_n <_q n$.*

Preuve : si une trace correcte ne respecte pas la causalité alors, par définition, il existe un processus q recevant un message m avant un message n, alors que m dépend causalement de n. D'après le lemme 7, soit il existe une chaîne $(n, ..., m)$, soit il existe une chaîne $(l, ..., m)$ où l est un message émis après n par le processus ayant émis n.

Dans le premier cas, la destination de n étant différente de la source de m, la chaîne $(n, ..., m)$ est nécessairement de la forme $(n, l, ..., m)$. Mais alors, la chaîne $(l, ..., m)$, "émise" par q après être "reçue" par q (puisque $m <_q n$ et $n <_q l$), viole l'hypothèse que la trace est correcte. Ce cas est donc impossible (figure A.1, trace du haut).

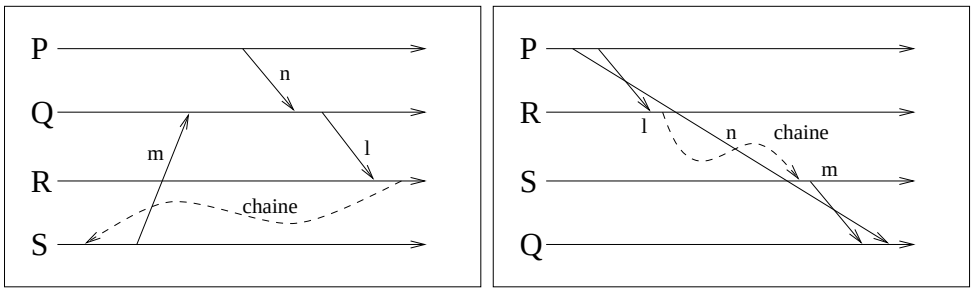

Figure A.1 – Les deux cas possibles

Dans le second cas, on a un message n de $p = src(n)$ vers q, ainsi qu'une chaîne $(l, ..., m)$ de p à q telle que $n <_p l$ et $m <_q n$ (figure A.1, trace du bas). Le lemme est donc démontré.

A.2 Théorème

On souhaite montrer que si une trace correcte respecte la causalité dans chaque domaine, alors toute trace abstraite associée respecte la causalité de façon globale. En fait, c'est faux dans le cas général, comme le montre l'exemple de la figure A.2 (en prenant comme trace abstraite la trace elle-même).

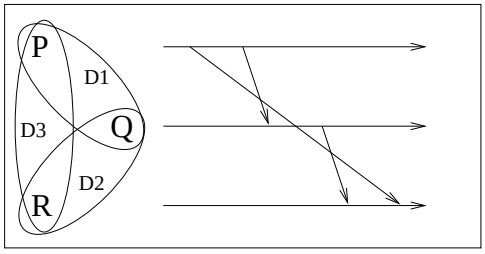

Figure A.2 – Un contre-exemple

A.2.1 Enoncé

Théorème 2 *Les deux propositions suivantes sont équivalentes :*

P1 toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, respecte la causalité de façon globale.

P2 il n'existe pas de cycles.

Un corollaire immédiat, en utilisant le fait qu'une trace peut être vue comme une trace abstraite associée à elle même, est que s'il n'existe pas de cycles alors toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale. D'autre part, la preuve de $P1 \Rightarrow P2$ ci-dessous montre aussi l'implication inverse : si toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale, alors il n'existe pas de cycles.

A.3 Preuve de $P1 \Rightarrow P2$

Pour prouver que $P1 \Rightarrow P2$, il est équivalent de prouver la contraposée, qui s'énonce ainsi : s'il existe un cycle, alors il existe une trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, qui ne respecte pas la causalité globale.

Puisqu'il existe un cycle, il existe, par définition, un chemin direct $[p_1, ..., p_c]$ tel qu'il existe un domaine contenant p_1 et p_c, et tel qu'il n'existe pas de domaine contenant tous les p_i. Considérons alors la trace représentée par la figure A.3.

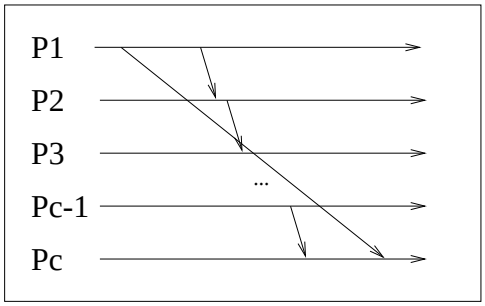

Figure A.3 – Trace correspondant à $[p_1, ..., p_c]$

Cette trace est correcte (car $[p_1, ..., p_c]$ est un chemin, et car il existe un domaine contenant p_1 et p_c). Par contre, elle ne respecte pas la causalité globale. Enfin, elle respecte la causalité dans chaque domaine. En effet, aucun domaine ne contient à la fois tous p_i. Donc aucune trace restreinte à un

domaine ne peut contenir tous les messages de la trace globale. Or il suffit de retirer un seul message (n'importe lequel) de la trace globale pour que la violation de la causalité disparaisse. On a donc construit une trace correcte qui respecte la causalité dans chaque domaine, ainsi qu'une trace abstraite associée (la trace elle-même) qui ne respecte pas la causalité de façon globale. Donc $\neg P2 \Rightarrow \neg P1$.

A.4 Preuve de $P2 \Rightarrow P1$

Pour montrer que $P2 \Rightarrow P1$, il suffit, d'après la contraposée du lemme 8, de montrer que :

> P : pour toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, il n'existe pas deux processus p et q, un message abstrait n' de p vers q, et une chaîne de messages abstraits $(m'_1, ..., m'_n)$ de p à q tels que $n' <_p m'_1$ et $m'_n <_q n'$.

Pour cela, on montre par récurrence généralisée que $P(x)$ est vrai pour tout x, avec $P(x)$ définie par :

> $P(x)$: pour toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, il n'existe pas deux processus p et q, un message abstrait n' de p vers q *correspondant à une chaîne minimale de longueur* x, et une chaîne abstraite $(m'_1, ..., m'_n)$ de p à q tels que $n' <_p m'_1$ et $m'_n <_q n'$.

Preuve de $P(1)$

Par l'absurde, supposons qu'il existe, pour une trace abstraite associée à une trace correcte respectant la causalité dans chaque domaine, deux processus p et q, un message abstrait n' de p vers q correspondant à une chaîne (n) de longueur 1, et une chaîne abstraite $(m'_1, ..., m'_n)$ de p à q telle que $n' <_p m'_1$ et $m'_n <_q n'$ (trace concrète similaire à celle du bas de la figure A.1).

Soit alors $(m_1, ..., m_k)$ la chaîne concrète correspondant à la chaîne $(m'_1, ..., m'_n)$. D'après le lemme 6, il existe une chaîne directe $(n_1, ..., n_h)$ telle que $m <_p n_1$ et $n_h <_q m$. Cette chaîne ne peut pas être de longueur 1, sinon la causalité serait violée dans le domaine contenant p et q (qui existe puisqu'il existe un message de p vers q). Soit donc $(p, r, ..., q)$ le chemin, direct par définition, associé à cette chaîne. Aucun domaine ne peut contenir tous les processus de ce chemin (sinon la causalité serait violée dans ce domaine). Par contre, il existe un domaine contenant p et q. $(p, r, ..., q)$ est donc un cycle, ce qui est contradictoire avec les hypothèses. Donc $P(1)$ est vrai.

Preuve de $P(x)$

Supposons que $P(y)$ est vrai pour tout $y < x$, et montrons que $P(x)$ est vrai. Par l'absurde, supposons qu'il existe, pour une trace abstraite associée à une trace correcte respectant la causalité dans chaque domaine, deux processus p et q, un message abstrait n' de p vers q correspondant à une chaîne minimale $(n_1, ..., n_x)$ de longueur $x \geq 2$, et une chaîne abstraite $(m'_1, ..., m'_n)$ de p à q telle que $n' <_p m'_1$ et $m'_n <_q n'$ (trace abstraite similaire à celle du bas de la figure A.1).

Soit $(p, a_1, ..., a_{x-1}, q)$ le chemin (minimal) associé à la chaîne $(n_1, ..., n_x)$. Soit $(m_1, ..., m_u)$ la chaîne concrète correspondant à la chaîne de messages abstraits $(m'_1, ..., m'_n)$. D'après le lemme 6, il existe une chaîne directe $(l_1, ..., l_v)$ telle que $m_1 <_p l_1$ et $l_v <_q m_u$. Soit alors $(p, b_1, ..., b_{v-1}, q)$ le chemin direct associé[2].

Premier cas possible : les processus du chemin $(a_{x-1}, ..., a_1, p, b_1, ..., b_{v-1}, q)$ sont 2 à 2 distincts. Dans ce cas, ce chemin est un cycle. En effet, il existe un domaine contenant a_{x-1} et q, et il n'existe

[2] $v \geq 2$, sinon $(n_1, ..., n_x)$ ne serait pas minimale

pas de domaine contenant p et q (lemme 5 appliqué au chemin minimal $(p, a_1, ..., a_{x-1}, q)$), et donc, a fortiori, il n'existe pas de domaine contenant tous les processus du chemin. Or l'existence d'un cycle est contradictoire avec les hypothèses, donc ce cas est impossible.

Deuxième cas possible : les processus du chemin $(a_{x-1}, ..., a_1, p, b_1, ..., b_{v-1}, q)$ ne sont pas 2 à 2 distincts. Les a_i étants 2 à 2 distincts et distincts de p et q, et les b_i étants 2 à 2 distincts et distincts de p et q, c'est donc qu'il existe i et j tels que $a_i = b_j$. Les chaînes $(n_1, ..., n_x)$ et $(l_1, ..., l_v)$ ne peuvent pas se « croiser » en a_i (comme dans la figure 4.8). Donc elles se croisent soit « avant », soit « après » ce processus (figure A.4). Dans le premier cas, la trace abstraite correspondant à l'abstraction $\{(n_1, ..., n_i), (l_1), ..., (l_i)\}$[3] ne vérifie pas $P(i), i < x$. Dans le second cas, la trace abstraite correspondant à l'abstraction $\{(n_{i+1}, ..., n_x), (l_{i+1}), ..., (l_v)\}$ ne vérifie pas $P(x - i), x - i < x$. Donc, dans les deux cas, on peut construire une trace abstraite qui ne vérifie pas $P(y)$ avec $y < x$, ce qui est contradictoire avec les hypothèses.

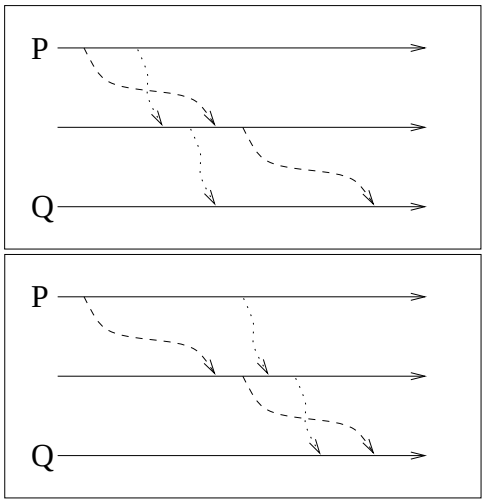

Figure A.4 – « Croisements » possibles des chaînes

Dans tous les cas, on aboutit à une contradiction. Donc $P(x)$ est vrai et, par récurrence, $P(n)$ est vrai pour tout n. Donc P est vrai, ce qui revient à dire que $P2 \Rightarrow P1$.

[3] c'est bien une abstraction, car $(n_1, ..., n_i)$, extraite d'une chaîne minimale, est elle aussi minimale. D'autre part, elle vérifie le critère principal des abstractions (définition 17) puisque la chaîne $(n_1, ..., n_x)$ le vérifie par hypothèse.

Annexe B

Implémentation des principaux composants de l'*Event-Based Service* de DREAM

Cette annexe donne les listings des principaux composants de l'expérimentation réalisée sur DREAM. Certaines parties du codes ont été enlevées pour plus de clarté.

B.1 AgentEngine

Listing B.1 – Boucle d'exécution de l'`AgentEngine`.

```
public class AgentEngineThreadImpl extends DreamThreadComponent {

/*...*/

  public void runningMethod(){

       /** 1. get first message */
       EbsMessage ebsMess =
         ((GetMessage)getFcBindings("getMessage")).getMessage();

       /** 2. give message to AgentRegistry */
      ((EBSMessagePush)getFcBindings("ebsMessagePush")).push(ebsMess);

       /** 3. delete message from queue */
       ((GetMessage)getFcBindings("getMessage")).popMessage();
  }
}
```

B.2 AgentReactor

Listing B.2 – Code du composant `AgentReactor`.

```
public class AgentReactorImpl extends DreamComponent implements
AgentPush {

/*...*/
```

```
  /**
   * this method is called by Agent class when an agent is
   * created and needs a new AgentID.
   */
  AgentId newAgentId(short to){
    AgentId agentId = ((A3Admin)getFcBindings("a3Admin")).newAgentId(to);
    if (Debug.DEBUG)
      if (Debug.agentReactorDebug)
    Debug.trace("AgentReactor","newAgentId: "+agentId);
    return agentId;
  }

  /**
   * this method is called by AgentFactory to add a new agent into AgentId list
   */
  void createAgent(AgentId id, Agent ag){
    ((A3Admin)getFcBindings("a3Admin")).addAgent(id,ag);
    if (Debug.DEBUG)
      if (Debug.agentReactorDebug)
    Debug.trace("AgentReactor","addAgent: "+id +" "+ag.getName());
  }

  /**
   * this method is called by AgentFactory to remove an agent from AgentId list
   */
  void deleteAgent(AgentId id) throws UnknownAgentException{
    ((A3Admin)getFcBindings("a3Admin")).removeAgent(id);
    if (Debug.DEBUG)
      if (Debug.agentReactorDebug)
    Debug.trace("AgentReactor","removeAgent: "+id);
  }

  /**
   * this method is called by Agent class when an agent wants to send an notifcation.
   */
  void sendTo(AgentId from, AgentId to, Notification not){
    EbsMessage ebsMessage = new EbsMessage(from,to,not);
    Message message = new Message(A3AdminImpl.getServiceId(),
      new EBServiceId(to.to),ebsMessage);
    ((Push)getFcBindings("push")).push(message);
  }

  /**
   * push the message to agentReactor
   * @param agent is the Agent object wich receive the notification
   * @param from is the AgentId of the sender Agent.
   * @param not is the notification
   */
  public void push(Agent agent, AgentId from, Notification not)
    throws Exception {
    agent.react(from,not);
  }
}
```

B.3 AgentRegistry

Listing B.3 – Implémentation de l'`AgentRegistry`.

```
public class AgentRegistryImpl extends DreamComponent implements
```

```
A3Admin, EBSMessagePush {

  // Agents table
  private Hashtable agents;

/*...*/

  public void push(EBSMessage ebsMess) throws Exception {

    /** 1. lookup for Agent reference */
    AgentId agentId = ebsMess.getTo();
    agent = (Agent)agents.get(agentId);

    /** 2.  Do react Agent */
    ((AgentPush)getFcBindings("agentPush"))
       .push(ag,ebsMess.getFrom(),ebsMess.getNotification());
  }

  public void addAgent(AgentId agentId, Agent agent){
    agents.put(agentId,agent);
  }
  public void removeAgent(AgentId agentId)
        throws UnknownAgentException{
    agents.remove(agentId);
  }
  public AgentId newAgentId(EBServiceId eventBasedServiceId){
    return new AgentId(eventBasedServiceId.getServiceId());
  }
}
```

B.4 AgentsTransaction

Note : ce listing ne montre pas la gestion des entrées/sorties sur disque pour la sauvegarde des agents.

Listing B.4 – Implémentation de l'`AgentsTransaction`.

```
public class AgentsTransactionImpl extends DreamComponent
implements AgentPush, EBSPush {

  Vector eBSMessageBuffer;

/*...*/

  public void push(Agent agent, AgentId from, Notification not) {
    Try{
      ((AgentPush)getFcBindings("agentPush")).push(agent,from,not);
    }catch(Excep){ abort(agent, to, Excep); }
    commit(agent);
  }

  void commit(Agent agent) {
    // Push all new notifications in gout
    for (Enumeration e = eBSMessageBuffer.elements(); e.hasMoreElements();) {
      (getFcBinding(EBSPush)).push(e.nextElement());
    }
    // Save on disk the agent's state.
    agent.save();
  }
```

```
  void abort(Agent agent, AgentId to, Exception exc) {
    // Reload the state of agent from disk save
    agent = reload();
    // clean all notification sent.
    eBSMessageBuffer.clear()
    // send an error notification to sender agent
    (getFcBinding(EBSPush)).push(new EBSMessage(agent.id, to, new
    ExceptionNotification(exc)));
  }

  public void push(EBSMessage ebsmess) throws Exception {
    eBSMessageBuffer.add(ebsmess);
  }
}
```

Annexe C

Description MDL de l'intergiciel DREAM avec un *Event-Based Service*

Cette annexe donne la configuration de l'intergiciel DREAM pour le service *Event-Based Service* présenté dans le chapitre 6. Cette configuration décrite à l'aide du MDL, contient un composite MOM permettant de transférer les messages et le composite service *Event-Based Service* qui fournit une API à agents A3.

La première section donne la configuration initiale de DREAM avec le service à agents *Event-Based Service*. La deuxième section présente la mise à jour (partielle) pour l'ajout du composant `AgentsTransaction`.

C.1 Configuration initiale

```
<composite name="dream" type="fr.inrialpes.sardes.dream.DreamType"
  host="nickel.inrialpes.fr">
  <struct>
    <components>
    <composite name="MOM" type="fr.inrialpes.sardes.dream.MOMType">
      <struct>
        <components>
          <composite name="BasicMom"
            type="fr.inrialpes.sardes.dream.MOMType">
          <struct>
            <components>
              <component name="NetworkIn" threaded="true"
                template="fr.inrialpes.sardes.dream.mom.NetworkInTmpl"
                type="fr.inrialpes.sardes.dream.mom.NetworkInType"/>
              <component name="NetworkOut" threaded="true"
                template="fr.inrialpes.sardes.dream.mom.NetworkOutTmpl"
                type="fr.inrialpes.sardes.dream.mom.NetworkOutType"/>
              <component name="MessageQueueOut"
                template="fr.inrialpes.sardes.dream.
                  mom.MessageQueueNetworkOutTmpl"
                type="fr.inrialpes.sardes.dream.mom.MessageQueueType"/>
              <component name="DispatcherIn"
                template="fr.inrialpes.sardes.dream.
                  mom.DispatcherInTmpl"
                type="fr.inrialpes.sardes.dream.mom.DispatcherInType" />
              <component name="DispatcherOut"
                template="fr.inrialpes.sardes.dream.
                  mom.DispatcherOutTmpl"
```

```
            type="fr.inrialpes.sardes.dream.mom.DispatcherInType" />
          <component name="RemoteServiceDesc"
            template="fr.inrialpes.sardes.dream.
              mom.RemoteServicesDescTmpl"
            type="fr.inrialpes.sardes.dream.
              mom.RemoteServiceDescType" />
        </components>
        <bindings>
          <binding client="NetworkIn.push"
            server="DispatcherIn.inPush" />
          <binding client="NetworkOut.getMessage"
            server="MessageQueueOut.getMessage" />
          <binding client="DispatcherOut.outPush"
            server="MessageQueueOut.push" />
          <binding client="NetworkIn.serviceDesc"
            server="RemoteServiceDesc.serviceDesc" />
          <binding client="NetworkOut.serviceDesc"
            server="RemoteServiceDesc.serviceDesc" />
          <binding client="DispatcherIn.serviceDesc"
            server="RemoteServiceDesc.serviceDesc" />
        </bindings>
        <controller>
          <component-controller desc="momComposite" />
        </controller>
      </struct>
      <inits />
      <stop-policy type="automatic" />
    </composite>
    <component name="InBuffer"
      template="fr.inrialpes.sardes.dream.
        common.PushPushNotThreadedBufferTmpl"
      type="fr.inrialpes.sardes.dream.
        common.PushPushBufferType" />
    </components>
    <bindings>
      <binding client="this.inPush"
        server="InBuffer.inPush" />
      <binding client="InBuffer.outPush"
        server="BasicMom.inPush" />
      <binding client="BasicMom.outPush"
        server="this.outPush" />
    </bindings>
  <controller>
    <component-controller desc="momAdministrableComposite" />
  </controller>
</struct>
<inits>
  <init-component component="BasicMom">
    <parameter name="listenningPort"
      value="composite->listenningPort" />
  </init-component>
</inits>
<stop-policy type="automatic" />
</composite>
  <composite name="EventBasedService"
    type="fr.inrialpes.sardes.dream.PushServiceType">
  <struct>
    <components>
    <composite name="FonctionalService"
      type="fr.inrialpes.sardes.dream.PullServiceType">
    <struct>
      <components>
        <component name="EBSMessageTranslator"
          template="fr.inrialpes.sardes.dream.
          services.ebs.EBSMessageTranslatorTmpl"
```

```
            type="fr.inrialpes.sardes.dream.services.
              ebs.EBSMessageTranslatorType" />
          <component name="AgentEngine"
            template="fr.inrialpes.sardes.dream.
            services.ebs.AgentEngineTmpl"
            type="fr.inrialpes.sardes.dream.services.
              ebs.AgentEngineType" />
          <component name="AgentsRegistry"
            template="fr.inrialpes.sardes.dream.
            services.ebs.AgentsRegistryTmpl"
            type="fr.inrialpes.sardes.dream.services.
              ebs.AgentsRegistryType" />
          <component name="AgentReactor"
            template="fr.inrialpes.sardes.dream.
            services.ebs.AgentReactorTmpl"
            type="fr.inrialpes.sardes.dream.services.
              ebs.AgentReactorType" />
        </components>
        <bindings>
          <binding client="this.inPull"
            server="EBSMessageTranslator.inPull" />
          <binding client="this.outPush"
            server="EBSMessageTranslator.outPush" />
          <binding client="AgentEngine.getMessage"
            server="EBSMessageTranslator.getMessage" />
          <binding client="AgentEngine.ebsMessagePush"
            server="AgentsRegistry.ebsMessagePush" />
          <binding client="AgentRegistry.agentPush"
            server="AgentReactor.agentPush" />
          <binding client="AgentReactor.a3Admin"
            server="AgentsRegistry.a3Admin" />
          <binding client="AgentReactor.agentPush"
            server="EBSMessageTranslator.agentPush" />
        </bindings>
      </struct>
      <stop-policy type="automatic" />
      </composite>
      <component name="InBuffer"
        template="fr.inrialpes.sardes.dream.
          common.PushPullBufferTmpl"
        type="fr.inrialpes.sardes.dream.
          common.PushPullBufferType" />
      </components>
      <bindings>
        <binding client="this.inPush" server="InBuffer.inPush" />
        <binding client="FonctionalService.inPull"
          server="InBuffer.outPull" />
        <binding client="FonctionalService.outPush"
          server="this.outPush" />
      </bindings>
     <controller>
       <component-controller
         desc="pullServiceAdministrableComposite" />
     </controller>
  </struct>
  <inits>
    <init-component component="FonctionalService" />
  </inits>
  <stop-policy type="automatic" />
  </composite>
  </components>
  <bindings>
    <binding client="MOM.outPush"
      server="EventBasedService.inPush" />
    <binding client="EventBasedService.outPush"
```

```
      server="MOM.inPush" />
  </bindings>
  <controller>
    <component-controller desc="dreamComposite" />
  </controller>
</struct>
<inits>
  <init-component component="MOM">
    <parameter name="listenningPort" value="1234" />
  </init-component>
  <init-component component="EventBasedService" />
</inits>
<stop-policy type="custom" />
</composite>
```

C.2 Configuration pour ajout du composant `AgentsTransaction`

```
<composite name="EventBasedService"
    type="fr.inrialpes.sardes.dream.PushServiceType">
  <struct>
    <components>
    <composite name="FonctionalService"
      type="fr.inrialpes.sardes.dream.PullServiceType">
    <struct>
      <components>
        <component name="EBSMessageTranslator"
          template="fr.inrialpes.sardes.dream.
          services.ebs.EBSMessageTranslatorTmpl"
          type="fr.inrialpes.sardes.dream.services.
            ebs.EBSMessageTranslatorType" />
        <component name="AgentEngine"
          template="fr.inrialpes.sardes.dream.
          services.ebs.AgentEngineTmpl"
          type="fr.inrialpes.sardes.dream.services.
            ebs.AgentEngineType" />
        <component name="AgentsRegistry"
          template="fr.inrialpes.sardes.dream.
          services.ebs.AgentsRegistryTmpl"
          type="fr.inrialpes.sardes.dream.services.
            ebs.AgentsRegistryType" />
        <component name="AgentReactor"
          template="fr.inrialpes.sardes.dream.
          services.ebs.AgentReactorTmpl"
          type="fr.inrialpes.sardes.dream.services.
            ebs.AgentReactorType" />
        <component name="AgentsTransaction"
          template="fr.inrialpes.sardes.dream.
          services.ebs.AgentsTransactionTmpl"
          type="fr.inrialpes.sardes.dream.services.
            ebs.AgentsTransactionType" />
      </components>
      <bindings>
        <binding client="this.inPull"
          server="EBSMessageTranslator.inPull" />
        <binding client="this.outPush"
          server="EBSMessageTranslator.outPush" />
        <binding client="AgentEngine.getMessage"
          server="EBSMessageTranslator.getMessage" />
        <binding client="AgentEngine.ebsMessagePush"
          server="AgentsRegistry.ebsMessagePush" />
```

```
        <binding client="AgentRegistry.agentPush"
          server="AgentsTransaction.agentPush" />
        <binding client="AgentsTransaction.agentPush"
          server="AgentReactor.agentPush" />
        <binding client="AgentReactor.a3Admin"
          server="AgentsRegistry.a3Admin" />
        <binding client="AgentReactor.agentPush"
          server="AgentsTransaction.agentPush" />
        <binding client="AgentsTransaction.agentPush"
          server="EBSMessageTranslator.agentPush" />
      </bindings>
    </struct>
    <stop-policy type="automatic" />
</composite>
```

Bibliographie

[1] *UDDI : Universal Description, Discovery and Integration of web services, specification release 3*, 2003, http ://www.uddi.org.

[2] N. Adly, Nagi M., and J. Bacon, *A hierarchical asynchronous replication protocol for large-scale systems*, IEEE Workshop on Parallel and Distributed Systems, October 1993, pp. 152–157.

[3] G. Agha, *Adaptive middleware*, Communications of the ACM **45** (2002), no. 6, 31–32.

[4] G. A. Agha, *Actors : A model of concurrent computation in distributed systems*, no. ISBN 0-262-01092-5, The MIT Press, Cambridge, Massachussets, 1986.

[5] R. Allen, D. Garlan, and R. Douence, *Specifying dynamism in software architectures*, Proceedings of the Workshop on Foundations of Component-Based Software Engineering (Zurich, Switzerland), September 1997.

[6] D. H. Andrews, *IBM's San Francisco Project : Java frameworks for application developers*, Tech. report, IBM Corp., décembre 1996.

[7] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, *The design and performance of a scalable ORB architecture for CORBA asynchronous messaging*, Middleware'00, 2000, pp. 208–230.

[8] L. Avignon, T. Brethes, C. Devaux, and P. Pezziardi, *Le livre blanc de l'EAI*, Tech. report, OCTO technology, Octobre 1999.

[9] Ö. Babaoğlu and K. Marzullo, *Consistent global states of distributed systems : Fundamental concepts and mechanisms*, Distributed Systems (S. Mullender, ed.), Addison-Wesley, 1993, pp. 55–96.

[10] R. Baldoni, R. Friedman, and R. Van Renesse, *The hierarchical Daisy architecture for causal delivery*, IEEE International Conference on Distributed Systems, May 1997, pp. 570–577.

[11] G. Banavar, T. Chandra, R. Strom, and D. Sturman, *A case for message-oriented middleware*, Lecture Notes in Computer Science, vol. 1693, Distributed Computing 13th International Symposium, September 1999, ISBN 3-540-66531-5, pp. 1–18.

[12] G. Banavar, T. Chandra, D. Sturman, B. Mukherjee, J. Nagarajaro, and J. Storm, *An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems*, 19th International Conference on Distributed Computing Systems, IBM - http ://www.research.ibm.com/gryphon/, June 1999, pp. 262–272.

[13] L. Bellissard, *Construction et configuration d'applications réparties*, Ph.D. thesis, Institut National Polytechnique de Grenoble, ENSIMAG, 1997.

[14] L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill, *Component-based programming and application management with Olan*, Proceedings of Workshop on Distributed Computing Systems, May 1996, pp. 579–595.

[15] ———, *Distributed application configuration*, International Conference on Distributed Computing Systems, 1996, pp. 579–585.

[16] L. Bellissard, F. Boyer, M. Riveill, and J. Y. Vion-Dury, *System services for distributed application configuration*, May 1998.

[17] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte, *An agent platform for reliable asynchronous distributed programming*, Symposium on Reliable Distributed Systems, October 1999.

[18] P.A. Bernstein, *Middleware : A model for distributed system services*, Communications of the ACM **39** (1996), no. 2, 86–98.

[19] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, E. Fiiczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility, safety and performance in the SPIN operating system*, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'1995), décembre 1995, pp. 267–284.

[20] C. Bidan, V. Issarny, and T. Saridakis, *Designing an open-ended distributed file system in Aster*, Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems (Dijon, France), Septembre 1996, pp. 163–168.

[21] K. Birman, R. Constable, M. Hayden, C. Kreitz, O. Rodeh, R. van Renesse, and W. Vogels, *The horus and ensemble projects : Accomplishments and limitations*, Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00), January 2000.

[22] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, , and K. Saikoski, *The Design and Implementation of Open ORB v2*, IEEE Distributed Systems Online Journal, vol. 2 no. 6, Special Issue on Reflective Middleware, November 2001.

[23] G. Blair, G. Coulson, P. Robin, and M. Papathomas, *An architecture for next generation middleware*, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware'98, November 1998.

[24] G. S. Blair, L. Blair, V. Issarny, P. T., and A. Zarras, *The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00) (New York, USA), LNCS, no. 1795, Springer-Verlag, April 2000, pp. 164–184.

[25] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, and D. Winer, *Simple object access protocol (soap) 1.1*, Specification, W3C Note, May 2000, http ://www.w3.org/TR/SOAP.

[26] A. Brodsky, D. Brodsky, I. Chan, Y. Coady, J. Pomkoski, and G. Kiczales, *Aspect-oriented incremental customization of middleware services*, Tech. report, May 2001.

[27] E. Bruneton, T. Coupaye, and J.B. Stefani, *The Fractal Composition Framework, interface specification 1.0*, France Telecom RD ed., 2002, http ://www.objectweb.org/fractal.

[28] E. Bruneton, R. Lenglet, and T. Coupaye, *ASM : a code manipulation tool to implement adaptable systems*, Adaptable and extensible component systems (Grenoble, France), Novembre 2002.

[29] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, *Interfaces and Algorithms for a Wide-Area Event Notification Service*, Tech. Report CU-CS-888-99, Department of Computer Science, University of Colorado, october 1999, revised May 2000.

[30] ———, *Achieving scalability and expressiveness in an Internet-scale event notification service*, Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000), July 2000.

[31] K. Chen and L. Gong, *Programming open service gateways with Java embedded server technology*, no. ISBN 0-201-71102-8, August 2001.

[32] D. Cheriton and D. Skeen, *Understanding the limitations of causally and totally ordered communication*, Proceedings of the ACM SIGOPS'93, 1993, pp. 44–57.

[33] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web services description language (WSDL) 1.1*, Tech. report, W3C Note, March 2001.

[34] T. Coupaye and J. Estublier, *Foundations of enterprise software deployment*, Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR2000) (Zurich, Suisse), IEEE Computer Society Press, Mars 2000.

[35] G. Cugola, E. Di Nitto, and A. Fuggetta, *The JEDI event-based infrastructure and its application to the development of the OPSS WFMS*, IEEE Transactions on Software Engineering **27** (2001), no. 9, 827–850.

[36] N. De Palma, L. Bellissard, D. Féliot, A. Freyssinet, M. Herrmann, and S. Lacourte, *An Agent Based Message Oriented Middleware*, Tech. report, SIRAC Project and AAA-DYADE, INRIA Rhône-Alpes, 1999.

[37] L.G. Demichiel, L.U. Yalçinap, and S. Krishnan, *Entreprise Java Beans specification - version 2.0*, Specification, Sun Microsystems Inc., avril 2001.

[38] F. DeRemer and H.H. Kron, *Programming-in-the-large vs. programming-in-the-small*, IEEE Trans. Software Engineering, Juin 1976, pp. 114–121.

[39] P. Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, *The many faces of publish/subscribe*, ACM Computing Surveys **45** (2003), no. 2, 114–131.

[40] O. Fambon, A. Freyssinet, and S. Lacourte, *Procédé itératif de sérialisation d'objets logiciels structurés*, SchlumbergerSema, 2002, Brevet FR-3939/PR.

[41] O. Fambon, A. Freyssinet, and P. Laumay, *Procédé de communication réseau avec une carte à puce par messages asynchrones*, SchlumbergerSema, 2002, Brevet FR-3958/PR.

[42] D. Garlan, R. T. Monroe, and D. Wile, *Acme : Architectural description of component-based systems*, Foundations of Component-Based Systems (Gary T. Leavens and Murali Sitaraman, eds.), Cambridge University Press, 2000, pp. 47–68.

[43] R.S. Hall, D. Heimbigner, A. van der Hoek, and A.L. Wolf, *An architecture for post-development configuration management in a wide-area network*, Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS 97) (Baltimore, USA), Mai 1997.

[44] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, *Java Message Service (JMS), specification v1.1*, SUN Microsystems, Inc, 2002.

[45] C.A.R Hoare, *Communicating sequential processes*, Prentice Hall, 1985.

[46] A. Homer and D. Sussman, *Programmation MTS et MSMQ avec Visual Basic et ASP*, ch. 5, Eyrolles, Mars 1999.

[47] International Standards Organization, *ODP : Recommendations X.903 : basic reference model of open distributed processing*, 1992, ISO/IEC JTC1/SC21/WG7.

[48] V. Issarny, *Architectures logicielles pour systèmes distribués*, Habilitation à diriger des recherches, IRISA et IFSIC université de Rennes 1, Octobre 1997.

[49] V. Issarny and C. Bidan, *Aster : A Corba-based software interconnection system supporting distributed system customization*, Proceedings of the International Conference on Configurable Distributed Systems (Annapolis, Maryland, USA), Mai 1996, pp. 194–201.

[50] V. Issarny, C. Bidan, and T. Saridakis, *Achieving middleware customization in a configuration-based development environment : Experience with the Aster prototype*, Proceedings of the 4th International Conference on Configurable Distributed Systems (Annapolis, Maryland, USA), Mai 1998, pp. 207–214.

[51] G. Kiczales, J. Des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[52] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science **2072** (2001), 327–355.

[53] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP), June 1997.

[54] C. Kloukinas and V. Issarny, *Automating the composition of middleware configurations*, Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'2000), Septembre 2000.

[55] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, *Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000) (New York), LNCS, no. 1795, Springer-Verlag, April 2000, pp. 121–143.

[56] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communications of the ACM **21** (1978), 558–565.

[57] P. Laumay, *Déploiement d'un bus à messages sur un réseau à grande échelle*, Rapport de DEA, University Joseph Fourier, Juin 2000.

[58] P. Laumay, E. Bruneton, L. Bellissard, and S. Krakowiak, *Preserving causality in a scalable message-oriented middleware (short version)*, Proceedings of ERSADS'01 (Bertinoro, Italie), Mai 2001.

[59] P. Laumay, E. Bruneton, N. De Palma, and S. Krakowiak, *Preserving causality in a scalable message-oriented middleware*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'2001) (Heidelberg, Allemagne), no. 1795, Novembre 2001, pp. 311–328.

[60] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann, *Specification and analysis of system architecture using Rapide*, IEEE Transactions on Software Engineering, Special Issue on Software Architecture, Vol. 21, No. 4, April 1995, pp. 336–355.

[61] P. Maes and D. Nardi, *Meta-level architectures and reflection*, North-Holland ed., Elsevier Science Publishers B.V., 1988.

[62] M. Mansouri-Samani and M. Sloman, *Network and distributed systems management*, ch. Monitoring distributed systems, pp. 303–347, Addison-Wesley Longman Publishing Co., Boston, MA, USA., 1994.

[63] P. Marguerite, *Déploiement d'un intergiciel en environnement hétérogène à grande échelle*, Rapport de DEA, Université Joseph Fourier - Laboratoire Sardes, Juin 2003.

[64] M. Marinilli, *Java deployment with JNLP and WebStart*, ch. An Abstract Model for Deployment, pp. 24–59, SAMS Publisher, 2001.

[65] J. McAffer, *Metalevel architecture support for distributed objects*, Proceedings of Reflection'96 (San Francisco), 1996, pp. 39–62.

[66] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, *A Language and Environment for Architecture-Based Software Development and Evolution*, Proceedings of the 21st International Conference on Software Engineering (ICSE'99), 1999, pp. 44–53.

[67] Microsoft Corp., *The component object model specification*, Mars 1995.

[68] Object Management Group, Inc., *Corba component model*, 2001, OMG TC Document ptc/2001-11-03.

[69] Object Management Group, Inc., *Corba notification service, specification 1.0.1*, Août 2002.

[70] B. Oki, M. Pflueg, A. Siegel, and D. Skeen, *The information bus - an architecture for extensible distributed systems*, In Proceedings of the Fourteenth ACM Symposium on Operating System Principles, décembre 1993, pp. 58–68.

[71] Open Services Gateway Initiative, *OSGi service gateway specification, release 2*, Octobre 2001, http ://www.osgi.org.

[72] Open Source Initiative, *Basic reference model for open systems interconnection (iso 7498)*, 1984.

[73] F. Plasil, D. Balek, and R. Janecek, *SOFA/DCUP : Architecture for component trading and dynamic updating*, Proceeding of the 4th International Conference of Configurable Distributed Systems (ICCDS'98), Mai 1998, pp. 43–51.

[74] S. Quaireau and P. Laumay, *Ensuring applicative causal ordering in autonomous mobile computing*, Proceedings of Workshop : Middleware for Mobile Computing (in association with Middleware'2001) (Heidelberg, Germany), Novembre 2001.

[75] V. Quema, R. Balter, L. Bellissard, D. Feliot, A. Freyssinet, and S. Lacourte, *Déploiement asynchrone hiérarchique d'applications réparties à composants*, 3ème Conférence Française sur les Systèmes d'Exploitation CFSE'3, Octobre 2003.

[76] M. Raynal and M. Singhal, *Logical time : Capturing causality in distributed systems*, Computer **29** (1996), no. 2, 49–56.

[77] S. Reiss, *Connecting tools using message passing in the field environment*, IEEE Software, juillet 1990.

[78] G. Ribière, *Communication et traitement en mode message avec MQSeries*, Techniques de l'Ingénieur, traité Informatique **HA** (1997), no. H2768.

[79] Frederic Ruget, *Cheaper matrix clocks*, Proceedings of the 8th internationnal WorkShop on Distributed Algorithms (WDAG-8), 1994.

[80] D. C. Schmidt, *Middleware for real-time and embedded systems*, Communications of the ACM **45** (2002), no. 6, 43–48.

[81] D. C. Schmidt and C. Cleeland, *Applying patterns to develop extensible ORB middleware*, IEEE Communications Magazine Special Issue on Design Patterns (1999).

[82] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, *Abstractions for software architecture and tools to support them*, Software Engineering **21** (1995), no. 4, 314–335.

[83] C. Small and M. Seltzer, *Vino : An integrated platform for operating systems and database research*, Tech. Report TR-30-94, Harvard University, Cambridge, Massachussets, 1994.

[84] B.C. Smith, *Procedural Reflection in Programming Languages*, Thèse de doctorat, MIT, 1982, Available as MIT Laboratory of Computer Science Technical Report 272.

[85] J.B. Stefani, *A calculus of higher-order distributed components*, INRIA Research Report 4692, SARDES project, Janvier 2003.

[86] Sun Microsystems, Inc., *Java card^{TM} platform specification 2.2*, 2003.

[87] S. Tai and I. Rouvellou, *Strategies for integrating messaging and distributed object transactions*, IFIP/ACM International Conference on Distributed systems platforms (Middleware 2000), Springer-Verlag New York, Inc., 2000, pp. 308–330.

[88] France Telecom, *Jonathan - white paper*, Tech. report, Avril 2002.

[89] R. van Renesse, K. P. Birman, and S. Maffeis, *Horus, a flexible group communication system*, Communications of the ACM **39** (1996), no. 4.