Vers l'exogiciel – Une approche de la construction d'infrastructures logicielles radicalement configurables

Vivien Quema

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité : « Informatique : Systèmes et Communications »

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

Vivien Quéma

le 2 Décembre 2005

Vers l'exogiciel

Une approche de la construction d'infrastructures logicielles radicalement configurables

---

Directeur de thèse :

Jean-Bernard Stefani

JURY

M.	Gordon	Blair	Président
M.	Bertil	Folliot	Rapporteur
M.	Peter	Van Roy	Rapporteur
M.	Jean-Bernard	Stefani	Directeur de thèse
M.	Jacques	Mossière	Examinateur
M.	Gilles	Muller	Examinateur

Résumé

La problématique de cette thèse est celle de la construction de systèmes auto-administrables, c'est-à-dire de systèmes prenant eux-mêmes en charge les fonctions d'administration classiquement dévolues à des humains. La construction de tels systèmes requiert à la fois l'utilisation d'une technologie logicielle adaptée et la mise en place d'une algorithmique dédiée.

Concernant la technologie logicielle, nous proposons une démarche de construction d'infrastructures logicielles radicalement configurables, appelée *exogiciel*. Cette démarche, inspirée de la philosophie exo-noyaux, vise à minimiser le nombre d'abstractions — fonctionnelles, non fonctionnelles et architecturales — imposées au développeur d'applications. Nous illustrons ce concept d'exogiciel à travers la présentation de DREAM, un canevas logiciel à composants pour la construction d'intergiciels de communication.

Concernant les aspects algorithmiques de la construction de systèmes autonomes, une approche classique, adoptée par la théorie de la commande, est de mettre en place des boucles de commande. Dans cette thèse, nous présentons deux éléments de base des boucles de commande : LEWYS, un canevas logiciel à composants permettant de construire des systèmes d'observation de systèmes distribués, et FREECAST, un protocole de diffusion de groupe avec ordre total uniforme. Par ailleurs, nous montrons comment les différents logiciels présentés dans cette thèse peuvent s'intégrer dans JADE, un intergiciel développé au sein du projet SARDES pour construire des boucles de commande pour l'administration autonome de systèmes.

Abstract

This thesis focuses on the construction of self-administrable systems, i.e. systems that react to the occurrence of events, such as hardware and software faults, performance degradation, etc. Building such systems requires both a well-adapted software technology and dedicated algorithms.

Concerning the software technology, we propose a new approach to the construction of radically configurable software architecture, called *exoware*. This approach, insipired by the exokernel philosophy, aims at minimizing the number of abstractions — functional, non functional and architectural — imposed to the application developer. We illustrate the concept of exoware with the presentation of DREAM, a component-based software framework for the construction of asynchronous communication middleware.

Concerning the algorithms required for building autonomous systems, a classical approach based on control theory, is to deploy control loops in charge of the supervision and administration of the managed system. In this thesis, we describe two contributions to the design of such control loops : LEWYS, a component-based framework dedicated to the construction of monitoring systems and FREECAST, a group communication protocol implementing uniform total order broadcast. Moreover, we show how the various software elements described in the thesis can be integrated into JADE, a framework for autonomic system management developed by the SARDES project.

Remerciements

Je tiens tout d'abord à remercier Gordon Blair, professeur à l'Université de Lancaster, de me faire l'honneur de présider ce jury.

Je remercie Bertil Folliot, professeur à l'Université Pierre et Marie Curie, Paris VI, et Peter Van Roy, professeur à l'Université Catholique de Louvain, d'avoir accepté d'être rapporteurs de cette thèse. Ils ont évalué mon travail de manière approfondie et m'ont fait des remarques constructives. Je remercie également Jacques Mossières, professeur à l'Institut National Polytechnique de Grenoble et Gilles Muller, professeur à l'Ecole des Mines de Nantes d'avoir accepté d'être examinateurs de ce travail.

Je ne sais comment remercier Jean-Bernard Stefani, mon directeur de thèse ; malgré un emploi du temps bien chargé, il a toujours été très disponible. Ses conseils tout au long de cette thèse ont été précieux. Par ailleurs, son esprit curieux l'a encouragé à me laisser explorer différentes voies et à collaborer avec de nombreuses personnes, ce dont je lui suis très reconnaissant. Enfin, au long de ces trois années, il s'est toujours montré généreux, encourageant et motivant.

Une thèse est un long parcours que l'on n'effectue pas seul. Ainsi, je tiens à remercier toutes les personnes avec lesquelles j'ai pu collaborer durant cette thèse :

- Matthieu Leclercq a joué un rôle très particulier dans cette thèse. Depuis son arrivée en tant qu'ingénieur dans le projet SARDES en Février 2005, j'ai eu la chance et l'immense plaisir de travailler avec lui. Matthieu m'a tout d'abord été d'une aide précieuse en participant activement au développement de DREAM. Par ailleurs, dépassant le cadre de son travail d'ingénieur, Matthieu a montré un intérêt profond pour les aspects les plus scientifiques de DREAM, ne se lassant jamais des longues discussions d'architecture au tableau ; enfin, il a montré un enthousiasme pour le travail sans faille et fait preuve d'une très grande bienveillance à mon égard.

- Les membres de l'équipe SARDES pour la gentillesse de leur accueil et les nombreuses interactions que nous avons eues. Tout d'abord, Valérie Gardes et Elodie Toihein, assistantes de projet pour le travail conséquent qu'elles font. Egalement, Emmanuel Cecchet[1] qui est à l'origine de la collaboration avec l'EPFL et avec lequel j'ai eu le plaisir de travailler sur LEWYS ; Oussama Layaida qui a également travaillé sur LEWYS ; Philippe Bidinger et Alan Schmitt avec lesquels j'ai eu la chance de réaliser le système de types pour DREAM, en espérant que la collaboration formel-système continuera dans le futur ; Jakub Kornas avec qui nous avons travaillé sur la reconfiguration dans FRACTAL ; Sara Bouchenak, Fabienne Boyer et Noël De Palma, et les "nouveaux thésards" — Jérémy Philippe, Sylvain Sicard et Christophe Taton —, membres de l'équipe JADE qui ont montré leur intérêt pour

[1] Emmanuel a quitté SARDES pour rejoindre la société Continuent.

les travaux réalisés dans cette thèse et avec lesquels j'espère avoir des collaborations fructueuses dans le futur ; Valerio Schiavoni, stagiaire de l'Université de Rome III avec qui j'ai travaillé sur l'utilisation de la programmation par aspects dans FRACTAL ; Erdem Ozcan et Takoua Abdellatif pour les nombreuses discussions que nous avons eues sur FRACTAL ; enfin, Sébastien Jean, Sacha Krakowiak et Jacques Mossière pour leur disponibilité, leurs conseils et les nombreuses lectures d'articles qu'ils ont effectuées.

- Les membres de la société ScalAgent : Roland Balter, Luc Bellissard, David Féliot, André Freyssinet et Serge Lacourte qui ont montré avec constance leur intérêt pour DREAM et dont les nombreux conseils ont toujours été judicieux. J'exprime, en particulier ma gratitude à Roland, Luc et André qui ont guidé mes premiers pas dans la recherche, qui ont été très encourageants et m'ont toujours fait preuve d'une immense sympathie.

- Rachid Guerraoui, Ron Levy et Bastian Pochon de l'EPFL avec lesquels j'ai eu la chance de travailler sur FREECAST. Nous avons réellement passé d'excellents moments ensemble et j'espère pouvoir bénéficier longtemps de l'enthousiasme communicatif de Rachid. Par ailleurs, je tiens à lui dire un immense merci pour l'aide qu'il m'a apportée pour la recherche de mon post-doctorat.

- Les membres de l'équipe ASR de France Telecom R&D, et en particulier Eric Bruneton et Thierry Coupaye pour les travaux que nous avons menés en commun sur FRACTAL, ainsi que Sébastien Chassande-Barrioz et Pascal Déchamboux pour leur intérêt sur les travaux présentés dans cette thèse.

- Les membres du LSR, en particulier Pierre-Yves Cunin, Didier Donsez et Richard Hall de l'équipe Adèle avec lesquels nous avons collaboré sur certains aspects du déploiement et de la reconfiguration dans FRACTAL. Egalement Farid Ouabdesselam, directeur du laboratoire pour ses encouragements concernant mon cursus et les actions qu'il a initées au sein du laboratoire pour animer la vie scientifique des doctorants. Enfin, Pascal Poulet qui a toujours été très serviable, et ce avec le sourire !

- Les membres du département R&D de Schneider avec lesquels nous avons travaillé au sein du projet INSIDE. En particulier, je remercie Philippe Lalanda de l'intérêt porté à nos travaux et pour sa gentillesse.

Enfin, sur le plan non scientifique, je voudrais exprimer ma gratitude à :

- Mes amis thésards. Personne mieux qu'un thésard ne peut comprendre un thésard. J'ai eu la chance d'en côtoyer trois formidables — Oussama, Erdem et Takoua — avec lesquels les liens d'amitiés demeureront.

- Mes amis musiciens. "Sans la musique la vie serait une erreur" disait Nietzsche. Il avait bien raison. Un grand merci à Corinne Pothier-Denis, ma professeur de violon, pour sa patience face à mon manque de temps et sa gentillesse. Un merci tout spécial également pour Janne Sörensen, Martial Renard et les musiciens de l'Académie de Musique de Skopje qui m'ont permis de passer une semaine exceptionnelle en Macédoine au cours de ma rédaction.

- Ma famille : mes parents, ma sœur et mes grands-parents qui ont toujours été présents à mes côtés. Outre leur amour, il m'ont apporté un goût du travail qui est nécessaire pour parvenir au bout d'une thèse. Enfin, Emilie dont la présence à mes côtés est inestimable et avec qui j'ai eu la joie d'avoir une petite Mathilde.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Vers la construction de systèmes autonomes

Les systèmes informatiques modernes intégrent un nombre croissant de processeurs hétérogènes — depuis des stations de travail traditionnelles jusqu'à des équipements mobiles — interconnectés par des réseaux aux caractéristiques diverses. La construction d'applications pour ces environnements pose, entre autres, les deux défis suivants :

- le *passage à grande échelle* : l'application doit supporter une augmentation de plusieurs de ses paramètres sans dégradation significative de ses performances. Ces paramètres sont, par exemple, le nombre de machines, d'utilisateurs, ou encore de sites.
- la *dynamicité* : les applications construites doivent pouvoir être reconfigurées dynamiquement afin d'intégrer de nouvelles fonctions, de maintenir un niveau de qualité de service requis, ou encore de pallier l'occurence d'une faute.

Le caractère dynamique des systèmes visés implique de les bâtir de manière à ce qu'ils soient *configurables*. Il doit, en effet, être possible de changer des éléments du système à tout moment de son cycle de vie. La combinaison de ce caractère dynamique et de la prise en charge d'environnements d'exécution à grande échelle rend les systèmes modernes très complexes. Maîtriser cette complexité constitue un défi auquel on ne peut envisager de répondre qu'en rendant les systèmes auto-administrables, c'est-à-dire en faisant en sorte qu'ils prennent eux-mêmes en charge les fonctions d'administration classiquement dévolues à des humains. Un système ayant cette capacité est dit **autonome** ; il se reconfigure dynamiquement lors de l'occurence de certains événements (panne de machine, dégradation des performances, etc.).

La construction de systèmes autonomes requiert à la fois l'utilisation d'une **technologie logicielle adaptée** et la mise en place d'une **algorithmique dédiée**.

La technologie logicielle utilisée doit permettre la construction de *systèmes administrables*. Un système administrable est un système sur lequel il est possible d'effectuer (dynamiquement) des opérations de gestion de configuration. Ces opérations permettent de paramétrer le système afin d'optimiser ses performances, de détecter et réparer les pannes survenant en cours d'exécution, d'ajouter et de retrancher des fonctions, etc. Pour parvenir à construire des systèmes administrables, un moyen élégant est de rendre ces systèmes *réflexifs*. Un système est dit réflexif lorsqu'il a la capacité de maintenir et d'utiliser une représentation de lui-même. Un système réflexif se décompose en deux niveaux : le niveau de base qui implante les fonctions du système et un méta-niveau qui contient la représentation du niveau de base et qui sert typiquement à implanter des opérations d'administration. Ces deux niveaux sont causalement liés : toute modification de l'un se répercute sur l'autre. Pour la construction de systèmes administrables, il est primordial que le méta-niveau réifie l'architecture logicielle interne du système. En effet, la

connaissance de cette architecture facilite la mise en œuvre des reconfigurations : elle permet notamment d'identifier les éléments à reconfigurer et de les introduire en conformité avec le reste de la structure.

Concernant les aspects algorithmiques de la construction de systèmes autonomes, nous pensons qu'il est nécessaire de mettre en place des *boucles de commande* semblables à celles utilisées dans la théorie de la commande [Oga97]. Ces boucles sont en charge de la régulation et de l'optimisation du comportement du système administré. Ce sont des boucles fermées faisant intervenir différents composants : le *système administré*, des *actionneurs* permettant de collecter des informations sur ce dernier (charge CPU, occurence de fautes, données applicatives, etc.) et un composant *gestionnaire* qui analyse les données collectées par les actionneurs et ordonne à des *capteurs* d'effectuer des opérations sur le système administré.

1.2 Proposition

Cette thèse propose des contributions aux deux thématiques mentionnées précédemment :
- technologie logicielle pour le développement de systèmes administrables.
- construction de boucles de commande pour l'administration autonome de systèmes.

1.2.1 Contributions à une technologie logicielle pour la construction de systèmes administrables

Dans cette thèse, nous proposons une démarche de construction d'infrastructures logicielles radicalement configurables, appelée *exogiciel*. Cette démarche est inspirée de la philosophie exonoyaux [EKO95]. Elle vise à minimiser le nombre d'abstractions — fonctionnelles, non fonctionnelles et architecturales — imposées au développeur d'applications. Elle repose sur l'utilisation de canevas logiciels à composants. Un tel canevas fournit une bibliothèque de composants configurables et administrables qui peuvent être assemblés afin de construire des systèmes complexes. Par extension de la notion classique de canevas logiciel, nous entendons par canevas logiciel à composants l'ensemble des trois éléments suivants :

1. un **modèle de composants réflexif** permettant de construire des architectures logicielles distribuées. Afin de pouvoir modéliser des systèmes complexes, il est nécessaire que ce modèle de composants autorise la construction de structures hiérarchiques avec partage de composants. Par ailleurs, il doit offrir des possibilités réflexives étendues (et extensibles) : il doit permettre d'associer à chacun des composants un méta-niveau arbitrairement complexe implantant des fonctions de contrôle diverses et variées.
2. une **bibliothèque de composants** contenant divers composants à partir desquels on peut constuire des systèmes complexes. Cette bibliothèque comprend des composants ciblant un domaine applicatif donné.
3. un **ensemble d'outils** permettant de décrire, déployer et administrer des systèmes réalisés à l'aide de la bibliothèque de composants. Ces outils s'organisent autour d'un langage de description d'architectures extensible.

Nous illustrons ce concept d'exogiciel à travers la présentation de DREAM, un canevas logiciel à composants dédié à la construction d'intergiciels de communication. DREAM utilise et étend le modèle de composants FRACTAL [BCL+04]. En outre, il définit une bibliothèque de composants encapsulant les fonctions de base des intergiciels de communication : files de messages, routeurs, canaux de transport, composants de sérialisation, etc. Enfin, il fournit un ensemble d'outils

permettant de vérifier, déployer, et reconfigurer dynamiquement les intergiciels construits à l'aide de la bibliothèque.

Nous démontrons, au travers du canevas DREAM, que l'approche *exogiciel* de la construction d'infrastructures logicielles permet de construire des **systèmes radicalement configurables**. C'est-à-dire que d'une part ces systèmes peuvent être modifiés dynamiquement à tous grains et, d'autre part, qu'ils ne présupposent aucun ensemble de fonctions donné. Nous illustrons ce caractère hautement configurable à l'aide d'exemples : implantation de divers paradigmes de communication à partir de la même bibliothèque de composants, modifications des modèles de concurrence utilisés, changement des propriétés non fonctionnelles fournies (atomicité, ordonnancement causal), adaptation d'une architecture à des équipements aux ressources restreintes.

Nous montrons également que la philosophie *exogiciel* n'induit **pas de pertes de performances substantielles** sur les infrastructures logicielles construites. Au contraire, nous montrons qu'en facilitant le développement de configurations adaptées aux équipements sur lesquels elles sont déployées, elle permet d'obtenir des performances sensiblement meilleures que celles obtenues par des architectures monolithiques peu configurables.

1.2.2 Élements d'infrastructures logicielles pour l'administration autonome de systèmes

Dans le cadre de cette thèse, nous montrons que la philosophie *exogiciel* est une base intéressante pour la construction de systèmes autonomes. Pour ce faire, nous présentons deux éléments de base des boucles de commande :

- LEWYS est un canevas logiciel à composants dédié à la construction de systèmes d'observation de systèmes distribués. LEWYS utilise le modèle de composants FRACTAL et définit une bibliothèque de composants permettant de construire des systèmes de collecte et de propagation d'événements caractérisant l'état du système administré. Cette bibliothèque comprend, entre autres, des composants, appelés *sondes*, qui permettent de collecter divers indicateurs sur les ressources physiques du système (CPU, mémoire, disque, etc.) mais également des données applicatives. LEWYS utilise DREAM pour propager les informations collectées. Développé suivant la philosophie *exogiciel*, LEWYS permet de construire des systèmes d'observation hautement et dynamiquement configurables. Il est aisé, par exemple, de développer et de déployer de nouvelles sondes, ou encore de modifier la sémantique de transport et de traitement des événements générés par les sondes.
- FREECAST est un protocole de diffusion de groupe avec ordre total uniforme développé à l'aide de DREAM. FREECAST permet de répliquer de façon active des systèmes afin de les rendre tolérants aux fautes. Nous l'avons développé pour permettre la réplication active des boucles de commande développées avec JADE. Outre le fait qu'il est un exemple concret d'utilisation de DREAM, FREECAST est intéressant car il présente une alternative aux protocoles de diffusion avec ordre total uniforme proposés par la communauté d'algorithmique théorique. En effet, nous montrons que l'utilisation d'autres métriques de performances que celles généralement utilisées permet de développer un protocole ayant des performances meilleures que celles de protocoles considérés optimaux par la communauté théorique.

Par ailleurs, nous présentons JADE, un intergiciel qui permet de construire des boucles de commande pour l'administration autonome de systèmes. JADE est un prototype réalisé par l'ensemble des membres du projet SARDES[1]. Bien que n'ayant pas été développé dans le cadre de cette thèse, nous décrivons JADE car il illustre une utilisation possible des différents systèmes

[1] System Architecture for Reflective Distributed EnvironmentS

présentés dans cette thèse pour la construction de boucles de commandes. Notons que l'implantation actuelle de JADE n'utilise pas ces systèmes. Néanmoins DREAM pourrait l'être pour construire des canaux de communication entre les différents éléments de la boucle de commande. LEWYS pourrait également être intégré à JADE pour implanter les composants d'observation. Par ailleurs, nous présentons une contribution au système JADE : l'utilisation du protocole de diffusion de groupe FREECAST pour répliquer les boucles de commande et les rendre tolérantes aux fautes.

1.3 Contexte de travail

Ce travail a été réalisé au sein de l'équipe SARDES qui est à la fois un projet INRIA et une équipe de recherche du laboratoire LSR-IMAG (CNRS, INPG, UJF). Le projet SARDES a pour objectif l'étude de l'architecture et la construction d'infrastructures logicielles pour des environnements répartis à grande échelle, caractérisés par une très grande taille, une très grande hétérogénéité, et par une nature très dynamique. En particulier, le projet s'intéresse à la construction de systèmes autonomes. Pour ce faire, il se propose d'exploiter de manière systématique des techniques de réflexion et de construction par composants.

Outre d'intenses collaborations avec les membres du projet SARDES, les travaux présentés dans cette thèse sont également le fruit de nombreuses collaborations avec d'autres institutions :

- Schneider Electric et Scalagent Distributed Technologies dans le cadre du projet RNTL INSIDE. Ce projet avait pour objectif la construction d'infrastructures pour la création de services Internet dans le domaine de la distribution électrique. La contribution de SARDES au projet INSIDE a été le développement de DREAM — et notamment la ré-ingénierie de la plate-forme ScalAgent présentée au chapitre 9. Notre rôle consistait, en effet, à concevoir un intergiciel asynchrone dynamiquement configurable dont le but était de fournir un ensemble de services de communication et d'exécution aux différentes entités impliquées dans les services (serveurs d'applications et passerelles Internet d'accès aux équipements électriques).
- France Telecom R&D dans le cadre du projet ITEA OSMOSE. L'un des objectifs de ce projet est la définition d'un modèle de composants hiérarchique et extensible. Une des contributions de SARDES au projet OSMOSE s'est traduite par les travaux sur la reconfiguration d'implantation présentés dans le chapitre 8.
- Le laboratoire de programmation distribuée de l'Ecole Polytechnique Fédérale de Lausanne (EPFL) dans le cadre d'une collaboration informelle dont le résultat est le protocole de diffusion de groupe FREECAST présenté au chapitre 11.

1.4 Organisation du document

Ce document s'organise en trois parties : la première partie décrit l'état de l'art des domaines considérés dans cette thèse. La deuxième partie est consacrée à la description du canevas DREAM. La troisième partie, enfin, présente nos contributions au développement de boucles de commande pour l'administration autonome de systèmes.

1.4.1 Organisation de la première partie

Dans la première partie, nous dressons un état de l'art de certains domaines couverts dans cette thèse. Nous avons décidé de ne traiter que les aspects relatifs à la technologie logicielle pour

la construction de systèmes administrables. L'étude de l'état de l'art concernant les problématiques abordées dans la troisième partie est effectuée indépendamment dans chaque chapitre.

Le but de la première partie est de motiver le développement du canevas logiciel à composants DREAM. Les trois sujets relatifs au canevas DREAM sont : la programmation par composants, la gestion de configuration et les intergiciels de communication adaptables. Nous avons donc naturellement effectué le découpage en chapitres comme suit : le chapitre 2 est dédié à l'étude de modèles de composants existants. Nous présentons des langages de description d'architectures au chapitre 3. Enfin, le chapitre 4 dresse un état de l'art des intergiciels de communication adaptables.

Pour étudier et comparer les différents systèmes décrits dans l'état de l'art, nous avons dressé une liste de l'ensemble des critères qui sont, selon nous, importants pour construire une technologie logicielle permettant la construction d'infrastructures logicielles dynamiquement configurables. Voici la liste de ces critères :

- **modèle de composition étendu :** il est important qu'une technologie logicielle permette de structurer le code des systèmes déployés sous forme de composants hiérarchiquement organisés. L'aspect hiérarchique facilite la prise en charge des fonctions de supervision, de contrôle, etc. Une caractéristique également primordiale est de pouvoir modéliser les composants partagés. Ces derniers sont, par exemple, particulièrement intéressants pour la modélisation des ressources.
- **possibilité de construire des systèmes de tailles variables :** la technologie logicielle doit avoir un champ d'application vaste, c'est-à-dire qu'elle doit permettre la construction de systèmes de tailles et d'ambitions variables. Par exemple, dans le cas des intergiciels de communication, il est nécessaire de pouvoir aussi bien bâtir des micro-protocoles que des systèmes implantant des sémantiques de communication complexes et fournissant des propriétés non fonctionnelles variées.
- **caractère administrable - configurabilité dynamique :** la technologie logicielle doit permettre de créer des architectures dynamiquement configurables. Elle doit faciliter la mise en place d'un méta-niveau implantant des opérations d'administration arbitrairement complexes.
- **minimisation des contraintes architecturales :** il est primordial qu'une technologie logicielle minimise les contraintes architecturales qu'elle impose. Il est, par exemple, souhaitable pour des raisons de performances que les modèles de concurrence utilisés ne soient pas imposés.
- **indépendance maximum envers des services déterminés :** ce critère signifie que la technologie logicielle doit avoir le moins de dépendances possibles envers des services fournis par l'environnement dans lequel elle s'exécute. Minimiser ces dépendances permet d'augmenter le nombre d'environnements dans lesquels la technologie peut être utilisée ; en particulier, une technologie avec des dépendances quasi nulle peut s'exécuter sur la plupart des équipements.
- **outils d'aide à la programmation :** pour faciliter son utilisation, il est important qu'une technologie logicielle fournisse des outils d'aide à la programmation. Outre la construction et le déploiement de systèmes, des outils particulièrement intéressants sont ceux permettant d'effectuer des vérifications sur les architectures déployées.

1.4.2 Organisation de la deuxième partie

La deuxième partie est consacrée à la description du canevas DREAM. Le chapitre 5 est une présentation générale du canevas. Il introduit les différents concepts présentés dans les chapitres

suivants. Nous présentons, dans le chapitre 6, le modèle de composants FRACTAL qui est utilisé comme base du canevas. Nous décrivons également le langage de description d'architectures associé à FRACTAL, car celui-ci sert de base aux outils de gestion de configuration du canevas DREAM. Le chapitre 7 est consacré à la description de la bibliothèque de composants DREAM. Ce chapitre présente à la fois des extensions du modèle FRACTAL pour la gestion des activités et les composants de la bibliothèque dédiés à la construction d'intergiciels de communication (e.g. files de messages, canaux de communication, composants de sérialisation, etc.). Par ailleurs, ce chapitre décrit un canevas permettant d'organiser les composants de la bibliothèque sous forme de piles de protocoles. Le chapitre 8 est dédié à la description des outils de gestion de configuration du canevas DREAM. Ces outils sont au nombre de trois : le premier outil permet d'effectuer des reconfigurations de structure (ajout de composants et de liaisons) sur une architecture en cours d'exécution ; le second outil permet la mise à jour du code des composants déployés ; le troisième outil est un outil de vérification de types qui sert à vérifier, avant le déploiement, que l'architecture de l'intergiciel est correcte (i.e. que chaque composant recevra des messages ayant la structure attendue). Enfin, nous concluons cette partie par une évaluation de DREAM dans le chapitre 9. Nous présentons les implantations de deux intergiciels réalisées à l'aide de la bibliothèque de composants. Pour chacun des intergiciels, nous procédons à une évaluation de performances et des bénéfices en configurabilité apportés par l'utilisation de DREAM.

1.4.3 Organisation de la troisième partie

La troisième partie du document est consacrée à la description de trois systèmes permettant de mettre en œuvre des boucles de commande. Chaque chapitre contient une section décrivant l'état de l'art des travaux connexes à ceux présentés dans le chapitre. Nous décrivons le canevas LEWYS au chapitre 10 : ce canevas est destiné à la construction de systèmes d'observation. Puis nous consacrons le chapitre 11 à FREECAST, le protocole de diffusion de groupe avec ordre total uniforme. Enfin, le chapitre 12 présente JADE, l'intergiciel qui permet de construire des systèmes autonomes. Les deux premiers systèmes ont été développés dans le cadre de cette thèse. Le dernier système est développé par d'autres membres du projet SARDES ; il est présenté pour mettre en perspective les différents travaux effectués lors de cette thèse.

Première partie

État de l'art

Chapitre 2

Modèles de composants

Ce premier chapitre de l'état de l'art est consacré à la description de divers modèles de composants. Nous commençons par définir ce qu'est un composant. Nous étudions ensuite deux catégories de modèles de composants : les modèles de composants standards — supportés par de grands groupes industriels — et d'autres modèles de composants, développés par diverses équipes de recherche. Enfin, nous concluons ce chapitre par une synthèse des forces et faiblesses des différents modèles présentés.

2.1 Qu'est-ce qu'un composant ?

Dès les années 70, une des préoccupations des développeurs était de bâtir des architectures modulaires. Néanmoins, c'est dans les années 90 que la programmation par composants a connu un véritable essor, du fait de nombreux travaux portant sur la construction d'outils pour la programmation distribuée. Les composants sont aujourd'hui utilisés pour développer aussi bien des applications, que des intergiciels, ou encore des systèmes d'exploitation. Chaque couche ayant des contraintes et des objectifs spécifiques, il est logique que les modèles de composants proposés diffèrent. Néanmoins, les principes de base de ces modèles restent les mêmes [Szy02] :

> *A component is a unit of composition that can be deployed independently and is subject to composition by a third party.*

L'intérêt majeur des composants réside dans le fait qu'ils sont des briques de base configurables qui vont permettre de construire des logiciels par composition. En effet, dans la plupart des modèles, un composant possède des interfaces qui définissent ses points d'interaction avec les autres composants. Les interfaces expriment à la fois les *services requis* et les *services fournis* par le composant. Par ailleurs, de nombreux modèles donnent la possibilité à un composant d'exporter des attributs permettant de le configurer (lors de son déploiement et/ou en cours d'exécution). En ce sens, on peut considérer les composants comme une étape vers les outils nécessaires pour la "programmation à gros grain" (*programming in the large* [DK76]).

Il existe diverses formes de composants répondant à différents besoins. Par exemple, certains modèles (e.g. EJB [Ent02], CCM [Gro02]) fournissent des supports d'exécution prenant en charge diverses propriétés non-fonctionnelles : persistance, transactions, protection, duplication, etc. De fait, ces modèles sont destinés à la construction d'applications nécessitant ces services, telles

les applications Web. D'autres modèles ont pour but de faciliter le déploiement d'applications patrimoniales. C'est notamment le cas de Jiazzi [MFH01] et OSGi [Ope03] qui permettent de gérer les dépendances entre classes Java patrimoniales via une notion de paquetage étendue. Ces modèles ne considèrent pas les composants comme des entités en cours d'exécution. Enfin, certains modèles (e.g. DCUP [PBJ97], FRACTAL [BCL+04], OpenCOM [CBCP01]) ciblent la construction de systèmes dynamiquement configurables ; ces modèles possèdent des caractéristiques évoluées en termes de structuration des applications et offrent la possibilité d'associer des contrôleurs aux composants afin de procéder à leur reconfiguration en cours d'exécution.

2.2 Modèles de composants standards

Dans cette section, nous présentons deux des modèles de composants industriels les plus utilisés : le modèle de composants CORBA (CCM pour *Corba Component Model*) standardisé par l'OMG, et les EJB (*Enterprise Java Beans*) dont la spécification est faite par Sun. Ces modèles permettent la construction d'applications de haut niveau, telles les applications Web.

2.2.1 CCM : le modèle de composants de CORBA

Le CCM (*Corba Component Model*) [Gro02] est un canevas de composants proposé par l'OMG. Nous commençons par présenter la structure des composants. Nous décrivons ensuite le rôle des conteneurs de composants. Enfin, nous étudions les langages de description de composants et les descripteurs de déploiement.

Structure d'un composant

La figure 2.1 représente la structure d'un composant. Un composant interagit avec les autres composants en utilisant des interfaces fonctionnelles (ou *ports*) dont il existe quatre types différents :

- Une **facette** est une interface serveur qui peut être utilisée par des clients en mode synchrone.
- Un **réceptacle** est une interface cliente en mode synchrone.
- Un **puits** d'événements est une interface serveur qui peut être utilisée par des clients en mode asynchrone.
- Une **source** d'événements est une interface cliente en mode asynchrone.

Par ailleurs, un composant possède des *attributs* qui permettent de le configurer lors de son déploiement ou en cours d'exécution.

Le conteneur de composants

Les composants CCM s'exécutent au sein de conteneurs qui fournissent certains services système (transactions, sécurité, tolérance aux fautes, etc.) et qui utilisent un bus CORBA pour l'acheminement des communications entre les ports. Les conteneurs prennent en charge un seul type de composants. Ils contiennent des *usines de composants* qui se chargent de la création et de la destruction d'instances de ce type de composant. Ils contiennent également des objets d'interposition pour les ports des composants. Les conteneurs permettent de gérer ces objets d'interposition à l'aide de deux interfaces : une interface d'introspection qui fournit des informations sur le type des composants, sur ses ports, ainsi que sur l'état de ses connexions aux autres composants ; une interface de gestion qui permet de créer des connexions entre les composants.

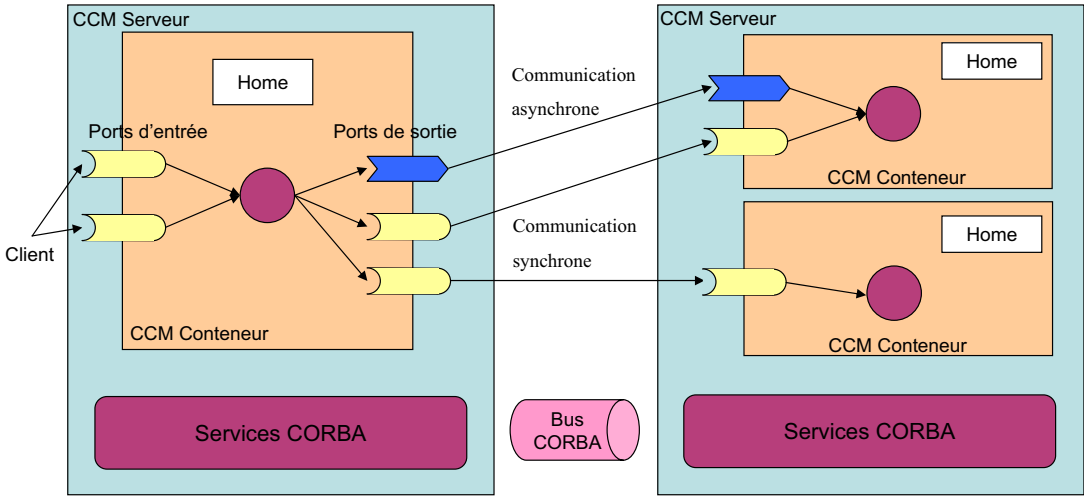

FIG. 2.1 – Architecture des conteneurs CCM.

Description de composants

Le type et l'implantation des composants CCM sont définis à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL (*Interface Definition Language*) de CORBA [cor02], ce qui permet à la fois de programmer les composants dans différents langages et de les exécuter dans des environnements hétérogènes. Les implantations de composants sont décrites à l'aide du langage de description CIDL (*Component Implementation Definition Language*). CIDL permet de décrire la structure logicielle de l'implantation des composants, ainsi que certains de leurs aspects non-fonctionnels : persistance, transaction et sécurité. La compilation des descriptions IDL et CIDL fournit, entre autres, un squelette prenant en charge la partie non-fonctionnelle d'un composant et un descripteur XML regroupant les contraintes techniques à respecter lors de son déploiement.

Déploiement de composants

Concernant le déploiement des composants, CCM utilise des *descripteurs de déploiement*. On distingue deux types de descripteurs :

- les **descripteurs de composants** permettent de décrire des composants "isolés". La description d'un composant contient sa structure logicielle, son type en termes de ports d'entrée et de sortie, ainsi que la catégorie à laquelle il appartient : processus, entité, session, service. Ces descripteurs sont générés lors de la compilation de la description des composants (faite à l'aide de CIDL), mais peuvent être modifiés pour paramétrer la gestion des aspects non-fonctionnels.
- les **descripteurs d'assemblage de composants** permettent de décrire l'architecture initiale d'une application. Ils spécifient les instances de composants constituant l'application, définissent des règles de placement sur des sites d'exécution, et indiquent les connexions à établir entre les différentes instances.

Synthèse

Le modèle CCM est destiné à la construcion d'applications ayant besoin de divers services pour s'exécuter. Des exemples typiques de telles applications sont les applications Web. L'avan-

tage de cette approche est qu'elle simplifie le développement d'applications. La complexité de la prise en charge des services non fonctionnels est masquée par des interfaces simplifiées et elle est prise en charge, d'une part par les conteneurs, d'autre part par les parties du composant qui sont générées par les compilateurs. Un des apports du modèle CCM est de proposer des outils nécessaires aux différentes étapes du cycle de construction d'une application (IDL, CIDL, descripteurs de déploiement).

Néanmoins, le CCM a un certain nombre de limites ; certaines sont citées dans [MM00]. Tout d'abord, tout conteneur doit prendre en charge un certain nombre de propriétés non fonctionnelles. De fait, cette dépendance vis-à-vis de services systèmes impose que les composants CCM soient exécutés sur des équipements ayant une certaine puissance. Par ailleurs, le modèle CCM a un modèle de composition limité : il n'est possible de créer ni composants composites, ni composants partagés. D'autre part, les capacités d'administration sont limitées : les seules possibilités de reconfiguration sont fournies par les interfaces d'introspection et de gestion de ports implantées par les conteneurs. Une autre limitation du modèle CCM est que les outils fournis permettent uniquement de générer et déployer du code. Il n'existe aucun outil de vérification des structures déployées. Enfin, CCM a un niveau de configurabilité très faible : les conteneurs sont monolithiques et ne supportent qu'un ensemble figé de propriétés non-fonctionnelles. Notons que des travaux académiques ont été effectués pour permettre la reconfiguration dynamique des services fournis par le conteneur [HMT+04]. Ces travaux proposent des mécanismes d'ajout de services dans le conteneur, basés sur un langage de reconfiguration dynamiquement adaptable, et sur un outil d'adaptation dynamique appelé CVM (*Container Virtual Machine*). Un des avantages de cette approche est qu'elle ne nécessite de modifier ni le code source des applications déployés, ni le code source du conteneur.

2.2.2 EJB : Enterprise Java Beans

Le modèle de composants EJB (*Enterprise Java Beans*) est spécifié par Sun Microsystems [Ent02]. Ce modèle est principalement dédié à la construction d'applications Web trois-étages [RAJ02]. Le premier étage est responsable de la présentation. Il s'exécute principalement dans une machine du côté de l'utilisateur. Le second étage, appelé étage applicatif, s'exécute dans un serveur. Enfin, le troisième étage, appelé étage de données, est souvent constitué d'une base de données. Le modèle EJB est destiné au développement du étage applicatif.

Architecture d'un EJB

Dans le modèle EJB, une application est constituée de composants implantés en Java, appelés EJB. Les EJB n'ont qu'une seule interface fonctionnelle serveur (appelée *remote*). On distingue trois types d'EJB : *session*, *entité*, ou *à messages*. Les instances de composants *session* sont créées lors de l'interaction avec les utilisateurs et leur durée de vie correspond à la durée de l'interaction. Une instance de composant session peut avoir un état (*stateful*) ou non (*stateless*). Les instances de composants *entité* représentent des données résidant dans une base de données et sont persistantes. Les instances de composant *message* sont semblables aux composants session mais supportent une communication de type asynchrone. Pour utiliser un EJB, il est simplement nécessaire d'obtenir sa référence. Celle-ci peut être communiquée par un autre EJB ou obtenue à l'aide d'un service de nommage.

Le conteneur d'EJB

Les instances d'EJB s'exécutent au sein de conteneurs qui prennent en charge les propriétés non-fonctionnelles nécessaires. Les conteneurs peuvent héberger plusieurs instances de différents types d'EJB. Ils gèrent les transactions, la sécurité et la persistance selon trois types de configurations prédéfinies correspondant aux trois types de composants.

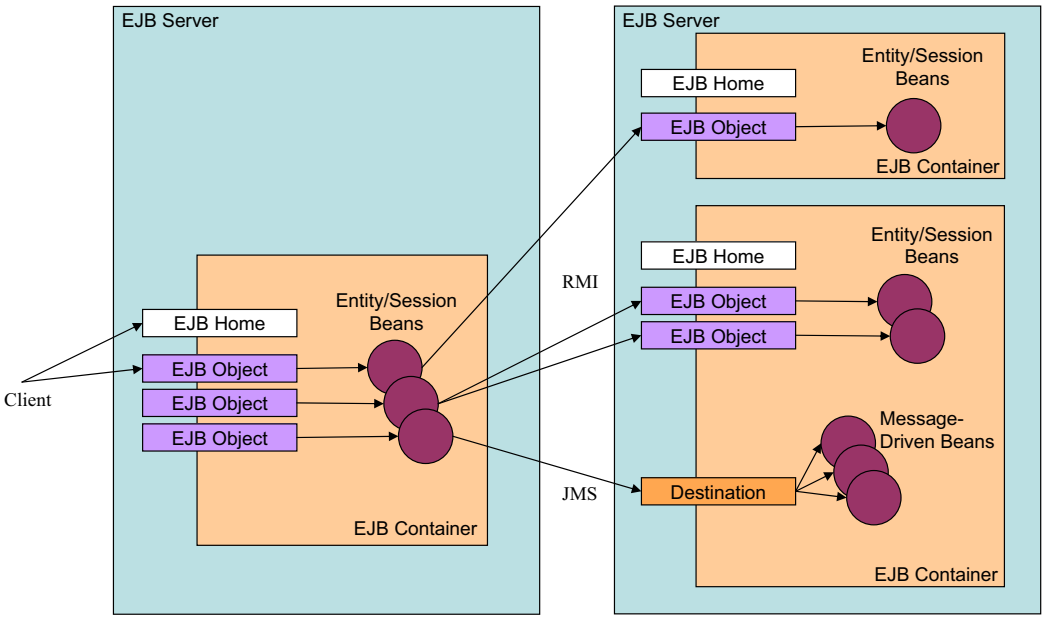

FIG. 2.2 – Architecture des conteneurs EJB.

Le conteneur est constitué d'objets d'interposition qui interceptent les invocations sur les EJB et qui gèrent les propriétés non-fonctionnelles par le biais de *pré-* et *post-traitements*. Des exemples de pré-traitements sont la vérification de droits d'accès, le démarrage de transactions, etc. Des exemples de post-traitements sont l'enregistrement des EJB sur support persistant, la terminaison de transactions, etc. Comme nous l'avons représenté sur la figure 2.2, les conteneurs s'exécutent au sein de serveurs fournissant divers services système nécessaires.

Contrôle au déploiement et à l'exécution

Les composants EJB sont livrés sous forme de fichiers JAR contenant, outre les classes du composant, un *descripteur de déploiement*. Ce descripteur contient des informations sur le code fonctionnel, ainsi que sur le code non-fonctionnel du composant. Concernant le code fonctionnel du composant, le descripteur contient une déclaration des classes d'implantation, ainsi que des interfaces d'accès à distance. En ce qui concerne les aspects non-fonctionnels, le descripteur contient une définition du type de composant (session, entité, à messages) et il précise les services système utilisés. Par ailleurs, ce descripteur permet de décrire des informations relatives aux assemblages de composants : il est possible de définir les dépendances entre EJB en spécifiant les types d'EJB qui sont référencés par un EJB donné.

Synthèse

Tout comme le modèle CCM, le modèle EJB a été conçu pour la construction d'applications métier nécessitant des services non fonctionnels. Son principal apport est donc la facilité de

développement induite par la prise en charge, par le conteneur, d'un certain nombre de propriétés non-fonctionnelles — le développeur pouvant se consacrer à l'écriture du code métier.

Le modèle EJB souffre des mêmes limites que le modèle CCM : dépendances vis-à-vis de services système qui nécessitent de déployer le conteneur sur des équipements puissants, modèle de composition limité, capacités d'administration limitées, faible niveau de configurabilité. Notons d'ailleurs que le modèle de composition est moins évolué que celui du modèle CCM : il ne permet, par exemple, pas de décrire de façon explicite les interfaces requises par les composants.

2.3 Autres modèles de composants

Cette section décrit divers modèles de composants développés par des équipes de recherche. Jiazzi propose une notion de paquetage étendue pour gérer les dépendances entre des classes Java patrimoniales. ArchJava fournit un langage pour la programmation des composants — extension de Java — associé à des outils garantissant que les composants communiquent exclusivement via des interfaces spécifiées ; DCUP, OpenCOM et Fractal ciblent la construction de systèmes dynamiquement configurables. Ces trois derniers modèles permettent tous de construire des architectures hiérarchiques à l'aide de composants composites. Ils n'ont, en revanche, pas les mêmes possibilités de reconfiguration : DCUP autorise la mise à jour du code des composants, tandis qu'OpenCOM et Fractal proposent de construire des systèmes réflexifs dans lesquels les composants possèdent divers contrôleurs permettant de les reconfigurer.

2.3.1 Jiazzi

Jiazzi [MFH01] est un système dédié à la construction d'applications Java à composants. Jiazzi ne nécessite ni extensions du langage Java, ni conventions particulières de codage. Les composants Jiazzi sont construits à partir de code source Java patrimonial ; ils peuvent être vus comme une généralisation des paquetages Java offrant la possibilité de liaisons externes et de compilation séparée. Jiazzi a des objectifs assez proches de systèmes comme OSGi [Ope03] et MJ [CBGM03].

Structure d'un programme Jiazzi

Un *paquetage* Jiazzi est similaire à un paquetage Java dans le sens qu'il permet de grouper un ensemble de classes. Un paquetage est défini par une *signature* qui permet de décrire les classes indépendamment de leur implantation. Les signatures de paquetage sont des sortes d'interfaces Java pour les paquetages ; elles sont nécessaires pour l'établissement de liaisons entre paquetages. La figure 2.3 donne l'exemple d'une signature d'un paquetage `program` comprenant une classe `Main`.

```
signature program = {
   class Main extends Object {
      static void main (String args[]);
   }
}
```

Fig. 2.3 – Exemple de signature d'un paquetage Jiazzi.

Les modules de code Java sont encapsulés dans des unités (*units*). Il existe deux types d'unités :

- les *atomes* (*atoms*) sont des unités primitives qui sont construites à partir de code Java. La figure 2.4 représente deux atomes : `driver` exporte le paquetage `main` dont la signature est `program` et importe le paquetage `maze` dont la signature est `mzbase` ; `base` exporte le paquetage `maze` dont la signature est `mzbase`. Les paquetages exportés correspondent aux services fournis par l'unité, alors que les paquetages importés représentent les services requis.
- les *composés* (*compounds*) sont des unités construites par assemblage d'autres unités. La figure 2.4 représente un composé `game` qui contient les atomes `base` et `driver` entre lesquels est définie une liaison à l'aide du mot-clé `link`. La signification de cette liaison est que les classes définies dans la signature `mzbase` et qui sont utilisées dans le code Java de l'atome `driver` sont implantées par l'atome `base`.

```
atom driver {
    export main : program;
    import maze : mzbase;
}

atom base {
    export maze : mzbase;
}

compound game {
    export main : program;
    export maze : mzbase;
    link unit base, driver;
}
```

FIG. 2.4 – Exemples d'unités (atome et composé) Jiazzi.

Vérification sur les architectures

Jiazzi permet d'effectuer différentes vérifications sur les architectures construites : il utilise un compilateur standard pour vérifier que les classes Java qui composent un atome sont conformes à la spécification du langage Java. Par ailleurs, il génère des souches (`stubs`) qui permettent aux classes importées de vérifier qu'elles sont correctement utilisées par les classes contenues dans l'atome. Enfin, un éditeur de lien permet d'assurer que les classes d'un atome sont conformes à sa signature et que les liaisons entre unités contenues dans un composé sont correctes.

Synthèse

Jiazzi est un modèle de composants permettant de structurer du code Java (éventuellement patrimonial) sous forme d'unités. Chaque unité peut importer et/ou exporter des paquetages. Un paquetage groupe un ensemble de classes et il est défini par une signature. Jiazzi distingue les unités primitives (atomes) qui encapsulent du code Java et les unités composites (composés) qui sont formées par composition d'unités.

Jiazzi a plusieurs limitations. Tout d'abord, son modèle de composition ne permet pas de créer des composants partagés. Notons néanmoins que ce dernier est plus complet que ceux des modèles CCM et EJB du fait qu'il permet de créer des unités composites. Par ailleurs, les composants Jiazzi ne peuvent pas être configurés dynamiquement. En effet, ils ne possèdent pas

de méta-niveau. Enfin, il est important de noter que les composants Jiazzi n'ont pas d'existence en tant que tels à l'exécution.

2.3.2 ArchJava

ArchJava [ACN02, ASCN03] est une extension du langage Java permettant de structurer du code en termes de *composants*, de *ports* et de *connecteurs*. Contrairement aux modèles de composants se basant sur des langages de description d'architectures pour découpler la description architecturale du code des composants, ArchJava propose d'intégrer les différents éléments de description d'architecture au sein du code des composants. L'objectif des concepteurs est de pouvoir certifier que les propriétés architecturales capturées dans la description sont respectées par l'implantation des composants.

Architecture d'une application ArchJava

ArchJava étend le langage Java avec les *composants* — qui décrivent les objets implantant les composants —, les *connexions* — qui permettent aux composants de communiquer —, et les *ports* — qui sont les points terminaux des connexions.

```
public component class Parser {

   public port in {
      provides void setInfo(Token symbol, SymTabEntry e);
      requires Token nextToken() throws ScanException;
   }

   public port out {
      provides SymTabEntry getInfo(Token t);
      requires void compile(AST ast);
   }

   void setInfo(Token t, SymTabEntry e) { ... };

   SymTabEntry getInfo(Token t) { ... };

   ...
}
```

FIG. 2.5 – Description d'une classe de composant ArchJava.

La figure 2.5 représente la description d'une classe de composant `Parser` à l'aide d'ArchJava. Ce composant définit deux ports, appelés `in` et `out`. Chaque port est décrit à l'aide d'un ensemble de signatures de méthodes. Chaque méthode peut être de type *requise*, *fournie*, ou *diffusion*. Une méthode requise est nécessaire au composant pour s'exécuter ; une méthode fournie est une méthode que le composant fournit aux autres composants ; une méthode de type *diffusion* est une méthode requise qui peut être connectée à plusieurs composants. Une telle méthode doit avoir `void` pour type de retour.

ArchJava permet de décrire des architectures logicielles hiérarchiques par le biais de *composants composites*. Un composant composite est constitué de sous-composants interconnectés par des connecteurs. La figure 2.6 décrit la classe d'un composant composite `Compiler`. Celui-ci contient trois sous-composants primitifs (notés par le mot-clé `final`). Les ports de ces composants primitifs sont connectés à l'aide du mot clé `connect`. Par exemple, le port `out` du composant

```
public component class Compiler {

   private final Scanner scanner = ... ;
   private final Parser parser = ... ;
   private final CodeGen codegen = ... ;

   connect scanner.out, parser.in;
   connect parser.out, codegen.in;

   public static void main (String args[]){
      new Compiler().compile(args);
   }

   public void compile(String args[]) {
      // for each file in args do:
   ... parser.parse(file); ...
   }
}
```

FIG. 2.6 – Description d'une architecture logicielle hiérarchique en ArchJava.

`scanner` est connecté au port `in` du composant `parser`. Notons qu'ArchJava permet de définir divers types de connecteurs [ASCN03] : de simples références Java, à des connecteurs plus sophistiqués (TCP, flux de données, transmission d'événements, etc.). Dans l'exemple de la figure 2.6, les connecteurs spécifiés sont des références Java.

Une des contributions majeures d'ArchJava est de garantir que les seules communications entre composants intervenant en cours d'exécution sont celles qui sont spécifiées dans une architecture (par le biais du mot-clé `connect`). Cette propriété, appelée *intégrité des communications* peut être vérifiée de façon formelle [ACN02].

Synthèse

ArchJava est destiné à la construction d'applications Java. Il propose une extension du langage Java permettant de coupler le code d'une application à la description de son architecture. Un des apports d'ArchJava est qu'il fournit des outils de vérification permettant de garantir l'adéquation de la description avec sa mise en œuvre. Il est, par exemple, possible de vérifier la propriété d'*intégrité des communications* qui stipule que les composants communiquent uniquement par le biais de connecteurs décrits dans l'architecture.

Parmi les critiques que l'on peut faire à ArchJava, citons le fait que son modèle de composition ne permet pas de créer des structures avec partage de composants. Par ailleurs, les composants ArchJava ne possèdent pas de méta-niveau. De fait, ces derniers ont des capacités de contrôle très limitées.

2.3.3 DCUP

DCUP (*Dynamic Component UPdating*) [PBJ97, PBJ98] est un modèle de composants qui a été conçu pour permettre la mise à jour dynamique des composants. Mettre à jour un composant consiste à le remplacer par une version plus récente. Pour permettre à ces mises à jour d'être transparentes, le modèle de composants a une architecture très spécifique que nous allons décrire.

Structure d'un composant DCUP

Une application DCUP est constituée par une hiérarchie en arbre de composants imbriqués. Chaque composant est composé d'un ensemble d'objets (Java dans la version actuelle de la plate-forme). Comme il a été représenté sur la figure 2.7, un composant se décompose différemment suivant l'aspect sous lequel il est considéré : sous l'aspect "nature des opérations fournies", le composant se décompose en une *partie fonctionnelle* — en charge des fonctions offertes par le composant —, et en une *partie contrôle* qui permet de gérer le composant : démarrage, mise à jour, etc. (figure 2.7 (a)). Considéré sous l'aspect "mise à jour dynamique", un composant est constitué d'une *partie permanente* et d'une *partie remplaçable* (figure 2.7 (b)).

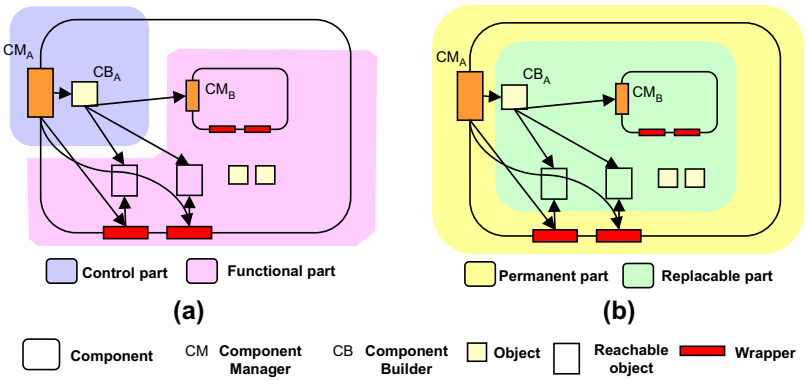

FIG. 2.7 – Structure d'un composant DCUP.

La partie fonctionnelle contient des objets, des sous-composants et des *wrappers*. Seuls les objets dits "accessibles" (*reachable objects*) peuvent être utilisés par les autres composants de l'application via les wrappers. Notons qu'à l'exception des wrappers, les autres constituants de la partie fonctionnelle appartiennent à la partie remplaçable du composant.

La partie contrôle contient le *component manager* et le *component builder*. Le component manager est le cœur de la partie permanente. Ses rôles sont (1) de gérer (stocker et disséminer) les références des wrappers des objets accessibles, et (2) de coordonner les mises à jour du composant. Le component builder est associé à une version particulière de composant. Il appartient donc à la partie remplaçable du composant. Ses rôles sont (1) de construire et détruire des composants de la version à laquelle il est associé, (2) de rendre accessible et de permettre la restauration de l'état du composant.

Mise à jour d'un composant

La mise à jour d'un composant est réalisée par la coopération du fournisseur du composant, du component manager, et des component builder des deux versions du composant concerné par la mise à jour. Le component manager héberge une entité responsable des mises à jour. Celle-ci interagit avec le fournisseur du composant pour obtenir les nouvelles versions du composant. Cette interaction peut se faire à l'initiative du component manager (mode pull) ou à l'initiative du fournisseur du composant (mode push).

Lorsqu'une nouvelle version du composant est disponible, le component manager télécharge les classes nécessaires. Il ordonne ensuite au component builder la destruction du composant. Avant de détruire les objets accessibles encapsulés, le component builder vérifie auprès des wrappers qu'ils ne sont plus utilisés. Par ailleurs, il exporte l'état du composant détruit : ceci consiste à sauvegarder les informations nécessaires sur le disque dur. Une fois le composant détruit, le

component manager instancie le component builder de la nouvelle version du composant et lui ordonne la création du composant. Le component builder peut éventuellement utiliser l'état du composant remplacé qui a été sauvegardé sur disque.

Synthèse

DCUP propose un modèle de composants qui permet la mise à jour dynamique des composants. Le point fort de DCUP est qu'il associe un méta-niveau à chacun des composants. Ce méta-niveau est néanmoins figé : il prend uniquement en charge la mise à jour des composants. Il n'est pas possible de l'étendre pour implanter d'autres fonctions d'administration.

Parmi les limitations de DCUP, citons son modèle de composition limité (absence de composants partagés), ainsi que l'absence d'outils pour la vérification des architectures déployées.

2.3.4 OpenCOM

Cette section décrit le modèle de composants OpenCOM [CBCP01], développé par l'Université de Lancaster dans le cadre du projet OpenORB[1].

Une version allégée de COM

OpenCOM est une version à la fois allégée et étendue du modèle COM de Microsoft. OpenCOM étant destiné au développement d'intergiciels, il ne fournit pas les propriétés de haut niveau — distribution, persistance, sécurité et transaction — offertes par COM. En revanche, OpenCOM conserve les caractéristiques principales de COM : le standard d'interopérabilité au niveau binaire, le langage de description d'interfaces (IDL), les identifiants globalement uniques et l'interface *IUnknown* qui permet de découvrir les autres interfaces du composant et de compter le nombre de références que possèdent les autres composants sur ce composant.

Les concepts-clés d'OpenCOM sont les suivants : interfaces, réceptables, et connexions. Une interface correspond à un service fourni. Un réceptacle représente un service requis. Une connexion permet de lier un service requis à un service fourni de même type. OpenCOM permet de déployer un support d'exécution par espace d'adressage. Ce support d'exécution permet de créer/détruire des composants et d'établir des liaisons entre ces composants. Ce support utilise deux interfaces qui doivent être implantées par tous les composants : l'interface *IReceptacles* permet de modifier les interfaces auxquelles un réceptacle est lié ; l'interface *ILifeCycle* définit des méthodes qui sont appelées lorsqu'un composant est créé ou détruit.

Reflexivité dans OpenCOM

La contribution fondamentale d'OpenCOM est qu'il permet d'associer un méta-espace à chaque composant. Ce méta-espace permet l'introspection et l'adaptation du composant[2]. Chaque méta-espace est structuré en différents méta-modèles, offrant différentes formes de réflexion (figure 2.8) :

- la **réflexion structurelle** fournit une représentation du contenu et de l'architecture du composant. Elle est mise en œuvre par deux méta-modèles : le méta-modèle *Interface* donne accès à une représentation du composant en termes d'interfaces fournies et requises.

[1] OpenORB est présenté dans le chapitre 4.

[2] Le méta-espace est construit suivant le même modèle de composants, ce qui permet de lui associer un méta-espace, appelé méta-méta-espace. Il en est de même pour tous les méta-espaces. En pratique, cependant, il n'y a qu'un méta-espace.

Le méta-modèle *Architecture* fournit une représentation de l'implantation du composant : un graphe des composants — qui représente l'architecture du composant en termes d'interconnexions de composants —, et des contraintes architecturales.

– la **réflexion comportementale** permet de contrôler les activités du système. Elle est mise en œuvre par le méta-modèle *Interception* qui permet l'ajout dynamique de pré- et post-traitements aux interactions entre composants. Ce méta-modèle s'avère notamment utile pour l'observation (*monitoring*) des composants.

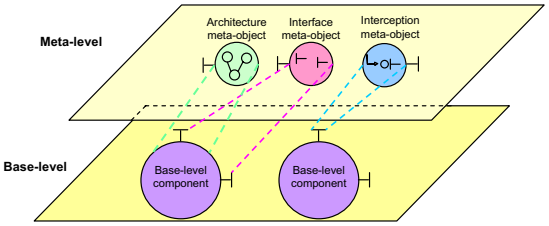

FIG. 2.8 – Les méta-espaces des composants OpenCOM.

Synthèse

Le modèle OpenCOM est un modèle dédié à la construction d'intergiciels. Son principal apport est de permettre l'association d'un méta-niveau à chacun des composants. Ce méta-niveau permet d'introspecter et de reconfigurer dynamiquement les composants. Il est, par exemple, possible de retrouver et de modifier l'ensemble des liaisons d'un composant. La version initiale d'OpenCOM était destinée à Windows et permettait uniquement d'associer aux composants les trois méta-niveaux sus-cités. Ces limitations vont disparaître avec la version 2 d'OpenCOM qui devrait être indépendante de l'environnement et qui ne fixe plus les méta-niveaux des composants. En revanche, une contrainte persiste : c'est l'impossibilité de construire des composants partagés, ce qui limite le modèle de composition proposé. Par ailleurs, OpenCOM ne fournit pas d'outils de vérification des architectures à déployer.

2.3.5 Fractal

Le modèle FRACTAL [BCL+04] est dédié à la construction et à la configuration dynamique de systèmes. FRACTAL est un modèle général qui n'est pas dédié à un langage ou à un environnement d'exécution particulier. Dans le cadre de ce chapitre, nous nous limiterons à la description de l'implantation Java du modèle.

Le modèle

FRACTAL distingue deux types de composants : les *composants primitifs* et les *composants composites* qui encapsulent un groupe de composants primitifs et/ou composites. Une caractéristique originale du modèle est qu'un composant peut être encapsulé simultanément dans plusieurs composites. Un tel composant est appelé *composant partagé*. Les composants partagés sont particulièrement adaptés à la modélisation des ressources.

Un composant est constitué de deux parties : la *partie de contrôle* — qui expose les interfaces du composant et comporte des objets contrôleurs et intercepteurs —, et la *partie fonctionnelle* — qui peut être soit une classe Java (pour les composants primitifs), soit des sous-composants (pour les composants composites).

Les interfaces d'un composant correspondent à ses points d'accès. On distingue deux types d'interfaces : les *interfaces serveurs* sont des points d'accès acceptant des appels de méthodes entrants. Elles correspondent donc aux services fournis par le composant. Les *interfaces clients* sont des points d'accès permettant des appels de méthodes sortants. Elles correspondent aux services requis par le composant.

La communication entre composants FRACTAL est uniquement possible si leurs interfaces sont liées. FRACTAL supporte deux types de liaisons : primitives et composites. Une *liaison primitive* est une liaison entre une interface client et une interface serveur appartenant au même espace d'adressage. Une *liaison composite* est un chemin de communication entre un nombre arbitraire d'interfaces de composants. Ces liaisons sont construites à l'aide d'un ensemble de liaisons primitives et de composants de liaisons (stubs, skeletons, adaptateurs, etc.).

Le modèle FRACTAL ne contraint pas la nature des contrôleurs contenus dans la partie de contrôle. Il est ainsi possible d'exercer un contrôle adapté sur les composants. La librairie FRACTAL fournit actuellement quatre contrôleurs :

- Le **contrôleur d'attributs** permet de modifier les attributs primitifs d'un composant. Ces attributs incluent les types primitifs Java (booléens, entiers, etc.) et les chaînes de caractères.
- Le **contrôleur de liaisons** permet d'établir ou de rompre une liaison primitive entre les interfaces clients du composant qui possède le contrôleur, et une ou plusieurs interfaces serveurs d'autres composants.
- Le **contrôleur de contenu** permet d'ajouter et de retrancher des sous-composants à un composant composite.
- Le **contrôleur de cycle de vie** permet de contrôler le cycle de vie du composant qui possède le contrôleur. Le cycle de vie d'un composant est représenté par un automate à deux états : *started* et *stopped*. Le contrôleur permet de passer d'un état à l'autre.

Exemple

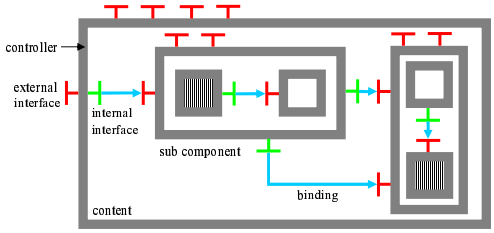

FIG. 2.9 – Exemple de composant FRACTAL.

La figure 2.9 représente un exemple de composant FRACTAL. Les composants sont représentés par des carrés. Le tour gris du carré correspond à la partie de contrôle du composant. L'intérieur du carré correspond à la partie fonctionnelle du composant. Les interfaces sont représentées par des "T" (verts pour les interfaces clients ; rouges pour les interfaces serveurs). Notons que les interfaces internes permettent à un composite de contrôler l'exposition de ses interfaces externes à ses sous-composants. Les interfaces externes apparaissant au sommet des composants sont les interfaces de contrôle du composant. Les flèches représentent les liaisons entre composants. Enfin, les deux composants hachurés correspondent à un seul et même composant, partagé entre les deux composites l'encapsulant.

Outil de déploiement

FRACTAL fournit un langage de description d'architectures extensible, appelé FRACTAL ADL. Ce dernier est composé d'un langage permettant de décrire les configurations de composants à déployer à l'aide d'une syntaxe XML et d'une usine en charge du traitement des définitions réalisées à l'aide du langage. Dans sa version actuelle, l'usine ADL permet uniquement de procéder au déploiement des architectures FRACTAL. Néanmoins, le langage et l'usine sont extensibles. Il est donc possible de les modifier pour prendre en charge d'autres aspects du cycle de vie de l'application comme la vérification ou la reconfiguration dynamique[3].

Synthèse

FRACTAL est un modèle de composants particulièrement flexible. Il définit un modèle de composition étendu dans lequel il est possible de créer des architectures hiérarchiques avec partage de composants. Par ailleurs, FRACTAL permet d'associer un méta-niveau arbitrairement complexe à chacun des composants. De fait, il est possible de configurer dynamiquement les architectures déployées. D'autre part, FRACTAL n'impose aucun service de base. Il permet donc un continuum entre configuration statique et reconfiguration dynamique. Il est ainsi possible d'instancier des composants avec des contrôleurs minimaux, ayant une empreinte mémoire faible et très peu ou même pas du tout d'impact sur les temps d'exécution, mais n'offrant pas de fonctions d'introspection et de reconfiguration. Il est également possible d'instancier des composants avec des contrôleurs plus évolués, permettant l'introspection et la reconfiguration dynamique, mais ayant un impact plus important sur les performances et empreinte mémoire des applications. Enfin, FRACTAL fournit un langage de description d'architectures extensible permettant de déployer des applications réparties. En revanche, FRACTAL ne fournit aucun outil pour procéder à la vérification des architectures à déployer.

2.4 Synthèse

Dans ce chapitre, nous avons présenté divers modèles de composants standards et académiques. Parmi les modèles présentés, les modèles CCM et EJB mettent l'accent sur le support d'exécution des composants. Celui-ci prend en charge diverses propriétés non-fonctionnelles : persistance, transactions, sécurité, etc. Ces supports fournissent, par ailleurs, plusieurs types de connecteurs (synchrones et asynchrones). Ces modèles ont un nombre important de limitations : tout d'abord, le conteneur doit prendre en charge de nombreuses propriétés non fonctionnelles ; cette dépendance envers des services systèmes rendent impossible le déploiement de composants sur des équipements aux ressources restreintes. Par ailleurs, les modèles de compositions proposés sont simplistes : ils n'offrent ni la possibilité de créer des composants composites, ni celle de créer des composants partagés. Enfin, les composants n'ont aucun méta-niveau associé. Les possibilités de configuration sont donc limitées et figées. Néanmoins, des travaux académiques sont effectués pour améliorer ces capacités de configuration [HMT+04].

Les modèles de composants Jiazzi et ArchJava sont tous deux des extensions du langage Java. Jiazzi étend le modèle de paquetage Java afin de gérer les dépendances entre des classes Java patrimoniales. ArchJava propose de coupler le code d'une application à la description de son architecture. L'apport majeur d'ArchJava est un langage de vérification dont le but est de garantir différentes propriétés sur l'architecture, telle que l'*intégrité des communications*. Par ailleurs, ArchJava et Jiazzi définissent des modèles de composition plus évolués que les modèles

[3] Des exemples d'extensions de FRACTAL ADL sont décrits dans le chapitre 8.

précédents du fait qu'ils permettent de créer des composants composites. En revanche, ils ne permettent pas de modéliser des composants partagés. Enfin, la limite majeure de ces deux modèles réside dans le fait qu'il n'est pas possible d'associer de méta-niveau aux composants, ce qui limite les possibilités de configuration dynamique.

Les trois derniers modèles présentés — DCUP, OpenCOM et FRACTAL — sont dédiés à la construction de systèmes dynamiquement configurables. Ils définissent un modèle de composition permettant de construire des architectures hiérarchiques à l'aide de composants composites. En revanche, seul FRACTAL permet de modéliser les composants partagés. Par ailleurs, ces modèles permettent tous d'associer un méta-niveau aux composants afin d'effectuer des reconfigurations sur les applications en cours d'exécution. Néanmoins, les possibilités qu'ils offrent sont différentes : DCUP limite la fonction du méta-niveau à la mise à jour du code des composants ; OpenCOM (dans sa deuxième version) et FRACTAL ne limitent pas les fonctions du méta-niveau. Enfin, notons qu'aucun de ces modèles ne fournit d'outils de vérification des architectures à déployer.

Chapitre 3

Langages de description d'architecture

Dans ce chapitre, nous présentons certains langages de description d'architectures qui existent actuellement pour faciliter la construction d'architectures logicielles à base de composants. L'architecture logicielle a toujours joué un rôle déterminant dans le succès des systèmes informatiques complexes. En effet, si le choix de l'architecture est approprié, cela permet au système de répondre aux exigences attendues et d'être facilement modifiable. Au contraire, une architecture mal pensée peut poser un certain nombre de problèmes notamment lorsque des modifications doivent être apportées au système. Malgré son importance, la structure des systèmes complexes est souvent conçue rapidement, sans règles précises. Ceci rend les logiciels moins performants et moins évolutifs. Pour répondre à ces problèmes, des outils formels de représentation d'architectures de systèmes ont été mis au point. Ces outils sont appelés langages de description d'architectures (ADL pour *Architecture Description Language*). Ce chapitre est dédié à l'étude des ADL. Nous commençons par une présentation générale des ADL. Nous étudions ensuite deux catégories d'ADL — permettant respectivement la génération d'exécutables et l'analyse des applications —, avant de conclure par une synthèse.

3.1 Qu'appelle-t'on ADL ?

Les ADL ont pour objectif de fournir une vue structurée d'un système informatique. Ils sont basés sur des concepts communément acceptés par les architectes d'applications [ISZ98, MT00] :

- les *composants* définissent les entités de base du système.
- les *connecteurs* définissent les types d'interaction entre les composants.
- la *configuration* définit une architecture d'application en terme d'interconnexions de composants par l'intermédiaire de connecteurs.

Les ADL existants diffèrent par l'exploitation qu'ils font 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 Polylith [Pur94], Darwin [MDEK95], UniCon [SDK+95], Olan [BBB+98], Aster [IB96b], ou encore C2 [MRT99] 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 [LKA+95] et Wright [AG97] 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.

Les sections suivantes sont consacrées à l'étude de représentants de ces deux catégories d'ADL.

3.2 ADL pour la génération et le déploiement d'exécutables

Dans cette section, nous présentons Darwin et Aster, deux ADL exploitant la description de l'architecture du système pour le déployer. Les particularités de Darwin sont qu'il permet de décrire des créations dynamiques de composants et qu'il fournit une infrastructure permettant de reconfigurer les applications. Aster se distingue des autres ADL par le fait qu'il définit un formalisme permettant de générer le bus logiciel nécessaire à la communication des composants déployés.

3.2.1 Darwin

Structure d'une application

Darwin [MDEK95] permet de construire des applications par interconnexion de composants. Il fournit un langage permettant de décrire la configuration d'une application, c'est-à-dire l'ensemble des composants logiciels ainsi que leurs interconnexions par le biais de *services fournis* et de *services requis*. Darwin distingue deux types de composants : les *composants primitifs* encapsulent du code fonctionnel. Ils possèdent des interfaces qui représentent les services requis et fournis par le composant. Les *composants composites* sont des composants encapsulant d'autres composants (primitifs ou composites). Leur rôle est de structurer les applications de façon hiérarchique. Les composants Darwin sont mis en oeuvre par des classes C++ et les communications entre composants sont mises en oeuvre par le support d'exécution Regis [MDK94].

```
component Compte {
   provide solde <port, int> ;
}

component Client {
   require consultation <port, int> ;
}

component Banque (int n) {
   // Pas d'interfaces

   // Mise en œuvre : définition d'un ensemble d'instances de clients et de comptes
   Array : Cl[n] : Client ;
   Array : Cpt[n] : Compte ;

   Forall k : 0..n-1 {
      Inst Cl[k] ; // instanciation d'un client
      Inst Cpt[k] ; // instanciation d'un compte
      Bind Cl[k].consultation - Cpt[k].solde ; // liaison client → compte
   }
}
```

FIG. 3.1 – Description d'une application Darwin.

La figure 3.1 présente un exemple de description d'application à l'aide de Darwin. Elle contient la définition de deux types de composants primitifs (`Compte` et `Client`) et d'un type de composant

composite `Banque`. Une particularité intéressante de Darwin est qu'il permet la description de créations dynamiques de composants. Pour ce faire, Darwin autorise l'utilisation de structures de contrôles classiques au sein de la description des composants composites : deux exemples sont l'itérateur `forall` et l'opérateur de test de condition `when`. Il est possible de spécifier, au sein de ces structures de contrôle, la création dynamique de composants. Néanmoins, il n'est pas possible de faire communiquer les composants faisant initialement partie de l'application avec les composants créés dynamiquement. La figure 3.1 illustre l'utilisation de l'itérateur `forall` pour le composite `Banque` ; celui-ci contient un ensemble d'instances de composants de type `Client` et un ensemble d'instances de composants de type `Compte` qui sont créées et liées dynamiquement.

Reconfiguration dynamique

L'un des apports de Darwin est de fournir une infrastructure permettant de reconfigurer dynamiquement une application. Cette infrastructure est issue de travaux antérieurs sur le système Conic [KM90]. Conic définit trois états dans lesquels un composant peut se trouver :

- *état actif* : le composant est en cours d'exécution : il peut démarrer, accepter et exécuter des requêtes.
- *état passif* : le composant n'est, et ne sera pas initiateur de requêtes. En revanche, il peut continuer à accepter et à satisfaire des requêtes.
- *état gelé* : le composant n'est impliqué dans aucune requête (que ce soit en initiateur ou en destinataire). Pour ce faire, il faut que le composant soit passif et que les composants à l'origine d'une connexion dont il est le destinataire soient également passifs.

Suivant l'état dans lequel le composant se trouve, il est possible d'effectuer différentes opérations de reconfiguration :

- *modification des interconnexions* : il est possible de créer, ou de supprimer une liaison entre deux composants. Pour ce faire, il faut que le composant à l'origine de la connexion soit gelé, de sorte qu'aucune requête utilisant une connexion en cours de modification ne puisse être initiée.
- *l'ajout d'un composant* : Darwin autorise l'ajout d'un nouveau composant dans la configuration d'une application. Cette opération ne requiert aucune condition particulière car elle n'affecte pas les composants présents.
- *le retrait d'un composant* : il est possible de supprimer un composant de la configuration. Cette opération nécessite que le composant soit gelé et que toutes ces liaisons aient été supprimées.

Ces opérations de reconfiguration sont mises en œuvre à l'aide d'un protocole dont la définition est donnée sur la figure 3.2 :

Ce protocole est mis en œuvre par un gestionnaire centralisé qui utilise la description ADL de l'application pour réaliser les deux premières étapes.

Synthèse

Darwin est un langage de description d'architectures permettant de modéliser une application sous la forme d'un assemblage de composants. Il permet de spécifier des configurations complexes, paramétrables, et autorise la spécification d'instanciations dynamiques de composants. L'intérêt de Darwin réside dans le fait qu'il propose un algorithme de reconfiguration, hérité de Conic, qui résout un certain nombre de problèmes difficiles tels que l'obtention d'un état cohérent avant d'entreprendre les reconfigurations. Les limites de cet algorithme de reconfiguration sont au nombre de deux : il repose sur le fait que les composants ont un cycle de vie imposé, constitué de trois états, et il est figé.

1. Déterminer l'ensemble QS (*Quiescent Set*) des composants à geler.
2. Déterminer l'ensemble CPS (*Change Passive Set*) des composants à passiver pour que les composants appartenant à QS soient gelés.
3. Procéder aux changements dans l'ordre suivant :
   (a) Passiver les composants appartenant à CPS.
   (b) Déconnecter les composants.
   (c) Retirer les composants.
   (d) Créer les composants.
   (e) Connecter les composants.
4. Réactiver les composants

FIG. 3.2 – Protocole de reconfiguration de Darwin.

3.2.2 Aster

Aster [IB96a, IB96b] permet de développer des applications distribuées ayant des exigences de qualité de service différentes. Partant du principe qu'un système doit permettre de supporter l'exécution d'un grand nombre de classes d'applications (des applications multimédias aux applications massivement parallèles), Aster propose un ensemble d'outils permettant de composer des applications à l'aide de composants logiciels patrimoniaux.

Une application Aster est composée de deux parties (figure 3.3) : un *bus logiciel* — constitué par un assemblage de composants —, et un ensemble de composants logiciels interagissant par l'intermédiaire du bus logiciel.

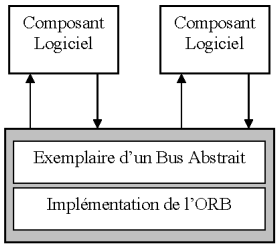

FIG. 3.3 – Structure d'une application Aster.

Un langage de spécification des propriétés d'exécution

Le but d'Aster est de permettre de sélectionner et de déployer l'ensemble des composants du bus logiciel nécessaires aux composants applicatifs. Pour ce faire, Aster fournit un langage formel qui permet de spécifier les propriétés d'exécution fournies et requises par les composants du bus logiciel et par les composants applicatifs. Ces propriétés sont définies en terme 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*).

Par exemple, il est possible de décrire que le bus logiciel fournit une sémantique de défaillance *au plus une fois* (ou *At-Most-Once*) de la façon suivante : soit C_1 et C_2 deux composants applicatifs et *req* une requête émise par C_1 à destination de C_2

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

De façon similaire, il est possible de définir une sémantique de défaillance *meilleur effort* (ou *Best-Effort*) de la façon suivante :

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

Spécialisation par appariement de spécifications formelles

Etant donné la spécification des propriétés fournies et requises par les composants du bus logiciel et par les composants applicatifs, Aster propose de construire un système spécialisé par appariement de spécifications formelles. On peut définir deux types d'appariements, 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*)**

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

Aster utilise cette définition de l'appariement de spécifications pour 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 implantant 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 $(\mathcal{P}_B \cup \mathcal{P}_A) \lhd_{plug\text{-}in} (\mathcal{R}_B \cup \mathcal{R}_A)$.

Déploiement d'une application

Outre le langage de spécification présenté dans les sections précédentes, Aster fournit deux outils, appelés *sélecteur* et *générateur*, qui sont respectivement en charge de rechercher les composants du bus logiciels nécessaires à la satisfaction des besoins de l'application, et de produire une application exécutable. Notons que le sélecteur effectue sa recherche de composants dans une base de données répertoriant les différents composants disponibles.

Synthèse

Le projet Aster permet de déployer une application après avoir selectionné certains des composants logiciels qui sont nécessaires à son exécution. Le choix de ces composants repose sur des descriptions formelles des propriétés requises et fournies par les différents composants.

Cependant, comme les concepteurs le soulignent dans [Iss97], l'environnement Aster ne garantit la correction sémantique du bus logiciel généré qu'à la condition que les spécifications des composants du bus soient correctes. Or le formalisme employé est assez complexe et peut donc s'avérer difficile d'utilisation pour le développeur d'applications réparties. Par ailleurs, il est à noter que les possibilités de spécifications offertes par le langage Aster sont assez limitées.

3.3 ADL pour l'analyse des applications

Dans cette section, nous présentons deux ADL, Rapide et Wright, exploitant la description de l'architecture du système pour procéder à son analyse. L'ADL Rapide repose sur la génération de collections d'événements survenant dans le système décrit. Ces collections sont ensuite analysées pour vérifier des propriétés sur l'ordonnancement des événements qu'elles contiennent. L'ADL Wright propose de décrire l'architecture à l'aide d'un formalisme proche de CSP pour vérifier que les interconnexions entre composants sont valides.

3.3.1 Rapide

Rapide [LKA+95, LV95] est un langage de description d'architectures dont le but est de décrire et de simuler des systèmes informatiques, tant au niveau matériel que logiciel. Un système est modélisé comme un ensemble de composants interconnectés qui peuvent générer des événements ou réagir à ceux-ci.

Description d'un composant Rapide

Chaque composant Rapide est décrit par une interface. Une interface est constituée d'un ensemble de services fournis et requis. Rapide distingue les services synchrones (clauses `provide` et `require`), des services asynchrones (clauses `action out` et `action in`). L'utilisation d'un service est modélisée par la génération d'un événement. La figure 3.4 présente la description de deux interfaces de composants Rapide. L'interface `Producer` contient deux services asynchrones : le service `Send` est fourni, ce qui signifie que le composant peut générer un événement de type `Send` ; le service `Reply` est requis, ce qui signifie que le composant peut recevoir des événements de type `Reply`.

Outre la description d'un ensemble de services, une interface contient une description de son comportement (clause `behavior`). Cette description explicite les interactions entre les services de cette interface. Elle consiste en un ensemble de règles de type événement – condition – réactions. Chaque règle est composée d'une partie droite et d'une partie gauche (séparées par le mot clé `=>`). La partie gauche correspond à un patron d'événements[1] (*event pattern*) dont l'interface attend l'occurence. La partie droite correspond à un patron d'événements généré lorsque la partie gauche est vérifiée. Ces patrons peuvent utiliser différents opérateurs permettant d'exprimer des dépendances entre événements. Parmi ces opérateurs, citons l'opérateur de dépendance causal (e_1 `->` e_2 si l'événement e_2 dépend causalement de e_1), d'indépendance ($e_1 \parallel e_2$ si e_1 et e_2 ne sont

[1] Ces événements sont à la fois ceux correspondant aux services et des événements de contrôle modélisant le comportement des composants de l'application (apparition d'un nouveau composant, valeur particulière d'un attribut, etc.).

```
type Producer (Max : Positive) is interface
   action out Send(N : Integer)
   action in Reply(N : Integer)
behavior
   Start => Send(0);
   (?X in Integer) Reply(?X) where ?X < Max => Send(?X + 1);
end Producer

type Consumer is interface
   action in Receive(N : Integer)
   action out Ack(N : Integer)
behavior
   (?X in Integer) Receive(?X) => Ack(?X);
end Consumer
```

FIG. 3.4 – Description d'un composant Rapide.

pas causalement dépendants), de simultanéité (e_1 `and` e_2 si e_1 et e_2 sont vérifiés simultanément), etc. Sur l'exemple de la figure 3.4, l'interface `Producer` définit deux règles de comportement. La première, par exemple, spécifie que lors de l'occurence de l'événement `Start`[2], le composant doit envoyer l'événement `Send(0)`.

Description d'une architecture

Rapide permet également de décrire des *architectures*, i.e. des assemblages de composants et leurs interconnexions. Un exemple d'une telle architecture est représenté sur la figure 3.5. Cette architecture contient deux composants `Prod` et `Con`. Les interconnexions entre ces composants sont représentées à l'aide de règles semblables à celles décrites dans la section précédente.

```
architecture ProdCons() return SomeType is
   Prod : Producer(100)
   Cons : Consumer
connect
   (?n in Integer)
   Prod.Send(?n) => Cons.Receive(?n);
   Cons.Ack(?n) => Prod.Reply(?n);
end architecture ProdCons
```

FIG. 3.5 – Description d'une configuration Rapide.

Vérifications sur les architectures

Rapide permet d'effectuer des vérifications sur les architectures décrites. Pour ce faire, un simulateur génère une collection d'événements partiellement ordonnés, appelée *poset* (*partially ordered sets of events*). Cette collection permet de savoir quelles sont les relations de causalités entre les différents événements qui se produisent.

[2]L'événement `Start` est un événement de contrôle qui est généré lors de la création de chaque composant.

Les collections générées pouvant être complexes à interpréter, Rapide définit 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.

Synthèse

Rapide permet de décrire des architectures logicielles distribuées et de spécifier des contraintes sur ces architectures. Il fournit un simulateur qui permet de simuler l'exécution de ces descriptions, afin de vérifier des propriétés sur l'ordonnancement des événements se produisant. Un autre apport de Rapide que nous n'avons pas présenté est qu'il permet de modéliser le comportement dynamique d'un système. Il est, par exemple, possible de décrire des créations dynamiques de composants et d'interconnexions.

La principale limitation de Rapide est qu'il ne permet d'effectuer des vérifications que sur des occurences d'événements et de patrons d'événements. Il n'est pas possible, par exemple, de valider des interconnexions de composants en se basant sur des vérifications sur le type des paramètres qu'ils s'échangent via leurs services requis et fournis.

3.3.2 Wright

Wright [AG97, AGD97] est un ADL permettant d'effectuer des vérifications formelles sur une architecture (absence d'interblocages, etc.). Pour ce faire, il propose de modéliser les composants et connecteurs à l'aide d'un calcul de processus proche de CSP [Hoa85].

Description d'une configuration

Composant La description d'un composant Wright est constituée de deux parties : la partie *Interface* et la partie *Computation*. La partie *Interface* décrit un ensemble de ports. Chaque port modélise une interaction à laquelle le composant participe. La description d'un port spécifie le protocole d'interaction que le composant utilise pour communiquer via ce port. La partie *Computation* modélise le comportement global du composant. Elle décrit les relations entre les protocoles des différents ports et explicite leur lien avec les actions internes du composant.

```
component Client
   port cport = (requete?x → reponse!y → cport) ⊓ ✓
   computation (calculInterne → cport.requete?x →
                cport.reponse!y → computation) ⊓ ✓
```

FIG. 3.6 – Description d'un composant Wright.

La figure 3.6 présente la description d'un composant `Client` possèdant un port `cport` qu'il utilise pour effectuer une requête vers un composant `Serveur`. L'interaction associée à ce port décrit le fait que le client émet des requêtes (`requete?x`) et reçoit en retour une réponse (`reponse!y`). Par ailleurs, la description du port spécifie que le composant peut, de façon arbitraire, mettre fin au processus représentant ce port (choix non déterministe $\sqcap$ et opérateur de fin de processus $\checkmark$). La partie *Computation* met en relation le comportement interne du composant avec le comportement observé au travers de ses différents ports. Dans le cas du composant `Client`, la

partie *Computation* est très semblable à la modélisation du port `cport` étant donné qu'il est le seul port du composant.

Connecteur Les connecteurs permettent de modéliser l'assemblage entre deux composants via leur ports de communication. La description d'un connecteur Wright est divisée en un ensemble de *roles* et une *glue*. Un connecteur est constitué de deux rôles : un rôle requis et un rôle fourni. Ces rôles définissent les protocoles d'interaction auxquels les ports des composants reliés par le connecteur devront se conformer. La glue d'un connecteur décrit pour sa part les interactions entre les différents rôles assurés par le connecteur.

```
connector C-S
   role client = (requete !x → reponse ?y → client) ⊓ ✓
   role serveur = (demande ?x → resultat !y → serveur) □ ✓
   glue = (client.requete ?x → serveur.demande !x → serveur.resultat ?y →
          client.reponse !y → glue) □ ✓
```

FIG. 3.7 – Description d'un connecteur Wright.

La figure 3.7 représente un connecteur qui permet de relier le composant `Client` à un composant `Serveur`. Le rôle `client` décrit le comportement d'un composant qui utilise ce connecteur pour envoyer des requêtes. Ce composant est modélisé par un processus qui peut (i) envoyer une requête et en recevoir le résultat, ou (ii) se terminer (✓). L'opérateur ⊓ spécifie que le choix de terminer le processus est effectué par le composant utilisant ce rôle. Le rôle serveur définit un processus qui peut (i) accepter répétitivement une requête et retourner une réponse, ou (ii) terminer (✓). L'opérateur □ spécifie que le choix est effectué par l'environnement de ce rôle (qui se résume à la glue et aux autres rôles). En d'autres termes, ce n'est pas le composant `Serveur` qui décide d'arrêter de fournir son service, mais c'est le composant `Client` qui arrête de l'utiliser. Enfin, la glue décrit la coordination des activités des deux rôles.

Configuration L'architecture d'une application Wright est décrite sous la forme d'une configuration. Une configuration est un ensemble de composants et de connecteurs entre ces composants. La modélisation d'une configuration est effectuée en deux étapes. La première correspond à la définition de l'ensemble des types de composants et de connecteurs qui pourront être utilisés dans la configuration ainsi que les contraintes d'interconnexion qui leur sont associées. Ces informations forment un *style architectural*. La deuxième étape dans la modélisation d'une configuration consiste à définir les instances de composants et de connecteurs, ainsi que les interconnexions de composants via ces connecteurs.

La figure 3.8 donne un exemple de configuration : les instances `client1` et `serveur1` sont liées par le connecteur `connecteur`.

Analyse des descriptions

Wright fournit des outils permettant de faire deux types de vérifications :

- **compatibilité des ports :** Wright permet de s'assurer que les interconnexions entre composants sont correctes. Une interconnexion entre deux composants est correcte si, et seulement si, chaque port prenant part à l'interconnexion correspond au rôle attendu par le connecteur. Cette propriété est vérifiée à l'aide de la notion de *refinement* du langage CSP.

```
configuration Application
   style Application
   instances
      client1 : Client
      serveur1 : Serveur
      connecteur : C-S
   attachments
      client1.client as connecteur.client
      serveur1.serveur as connecteur.serveur
end Application
```

FIG. 3.8 – Description d'une configuration Wright.

– **absence d'interblocages :** Wright permet de vérifier qu'aucun composant n'attendra d'actions d'autres composants qui ne se produiront pas. Il s'agit de vérifier que chacun des connecteurs ne cause pas d'interblocage, c'est-à-dire que lorsqu'un processus d'un connecteur est dans une situation dans laquelle il ne peut plus progresser, le dernier événement qui a eu lieu est l'événement *fin de processus* (✓).

Synthèse

Wright est un langage de description d'architectures qui permet de décrire l'architecture d'une application à l'aide d'un formalisme proche de CSP. La contribution majeure de Wright est de rendre possible des vérifications sur les architectures décrites : il est, par exemple, possible de détecter les éventuels interblocages, ou encore de vérifier que les interconnexions entre composants sont valides. Par ailleurs, nous ne l'avons pas décrit dans les sections précédentes, mais Wright permet de spécifier le comportement dynamique des applications : création et suppression de composants, modifications d'interconnexions, etc.

Cependant Wright a un certain nombre de limitations. Parmi celles-ci, citons le fait qu'il ne permet pas de déployer les applications décrites. Il ne permet également pas de décrire des composants répartis sur différents sites. Enfin, Wright autorise uniquement la description d'applications non hiérarchiques, ce qui le rend inadapté à la description d'applications de grande taille.

3.4 Synthèse

Dans ce chapitre, nous avons présenté divers ADL. Les ADL ont tous pour objectif de fournir une vue structurée d'un système informatique. Ils sont basés sur les concepts de composants, de connecteurs et de configurations. Nous avons distingué deux grandes catégories d'ADL qui diffèrent par l'exploitation qu'elles font de la description de l'application : génération et déploiement d'un exécutable ou analyse du système.

Les langages Darwin et Aster utilisent la description pour générer un exécutable et le déployer. L'originalité de Darwin est qu'il fournit un mécanisme de reconfiguration basé sur la description ADL de l'application. Ce mécanisme utilise un algorithme de reconfiguration qui résout un certain nombre de problèmes difficiles, tels que l'obtention d'un état cohérent avant d'entreprendre les reconfigurations. Néanmoins, le gestionnaire responsable des reconfigurations est mis en œuvre de façon centralisée, ce qui exclut son utilisation pour des systèmes à grande

échelle. Aster se distingue des autres ADL par le fait qu'il fournit un langage formel de description des composants. Ce langage, inspiré de la logique du premier ordre, permet de générer les composants de communication utilisés par les composants du système à déployer. Les principales limites d'Aster sont la complexité du langage et son expressivité limitée.

Les ADL Rapide et Wright permettent au développeur de l'application de spécifier le comportement dynamique des différents composants du système. Ils utilisent cette description pour modéliser l'application et analyser les scénarios envisageables. Ces ADL permettent d'effectuer des vérifications sur les architectures décrites : il est, par exemple, possible de détecter les éventuels interblocages, ou encore de vérifier que les interconnexions entre composants sont valides. Ces ADL n'ont néanmoins jamais été intégrés à des systèmes permettant de déployer et d'exécuter les applications.

Chapitre 4

Intergiciels de communication adaptables

Sommaire

Ce chapitre est consacré à l'étude des intergiciels de communication adaptables. Nous commençons par une définition des intergiciels de communication. Nous décrivons ensuite des intergiciels qui se limitent à la construction de protocoles de communication. Nous présentons ensuite quelques intergiciels adaptables, c'est-à-dire des intergiciels qui peuvent être reconfigurés dynamiquement.

4.1 Qu'est-ce qu'un intergiciel de communication ?

Comme nous le représentons sur la figure 4.1, l'intergiciel est une couche logicielle située entre le système d'exploitation et les applications. Les intergiciels de communication sont une forme particulière d'intergiciels qui offrent un ensemble de services de gestion de la distribution permettant aux applications de coopérer à l'aide d'un modèle et d'une interface de programmation.

Il existe différents modèles d'intergiciels de communication : client-serveur, communication par messages, communication par événements, code mobile, mémoire virtuelle partagée, etc. Nous n'entrerons pas dans le détail de ces différents modèles ; le lecteur intéressé pourra se référer à [Bis02] pour obtenir plus d'informations. Dans la suite de ce chapitre, nous présentons différents intergiciels de communication. Les deux premiers, Cactus et Appia, sont dédiés à la

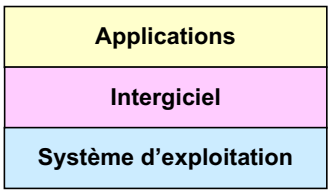

FIG. 4.1 – Positionnement de la couche intergiciel.

construction de piles de protocoles. Les autres intergiciels que nous présentons sont dynamiquement configurables. Ils utilisent des mécanismes de réflexivité afin d'autoriser leur introspection et leur reconfiguration dynamique.

4.2 Noyaux de protocoles

Cette section décrit deux intergiciels de communication destinés à la construction de piles de protocoles : Cactus et Appia. Cactus permet de construire des protocoles par assemblage de micro-protocoles. L'originalité de Cactus vient du fait qu'il propose un formalisme de description de contraintes entre micro-protocoles qui permet de garantir qu'un assemblage de micro-protocoles est correct. Appia se distingue des autres canevas de construction de piles de protocoles par la possibilité qu'il offre de multiplexer un ou plusieurs protocoles.

4.2.1 Cactus

Architecture

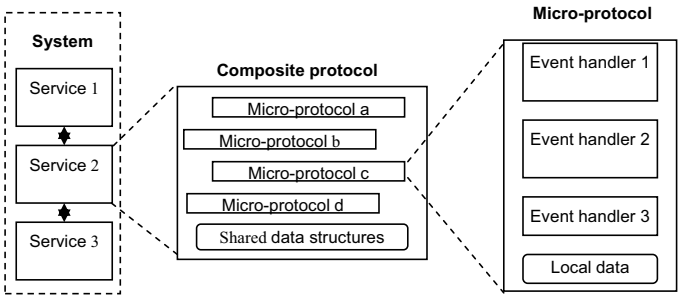

FIG. 4.2 – Architecture de Cactus.

Cactus [Hil98, HS00] est un système dédié à la construction de piles de protocoles réseaux configurables. Comme on peut le voir sur la figure 4.3, l'architecture de Cactus suit un modèle de composition à deux niveaux : un service (i.e. un protocole) est mis en œuvre par une collection de modules appelés *micro-protocoles*. Chaque micro-protocole est lui-même constitué d'un ensemble de traitants d'événements (*event handlers*). Les traitants d'événements sont exécutés lors de l'occurence de certains événements. Les événements sont générés par le système en exécution (e.g. lors de l'arrivée d'un message en provenance du réseau) ou par les micro-protocoles.

Cactus fournit une infrastructure d'exécution qui permet de gérer les événements et les traitants d'événements : il est, par exemple, possible de faire une liaison (ou une rupture de liaison) entre un traitant et un type d'événements. Notons que les micro-protocoles ne communiquent

pas uniquement par l'intermédiaire d'événements. En effet, ils peuvent également communiquer par appels de méthodes ou par accès à un espace de mémoire partagée.

Formalismes et outils d'aide à la génération de services "corrects"

Cactus fournit une aide à la construction de services "corrects", c'est-à-dire qu'il permet de garantir que les micro-protocoles assemblés pour créer un service sont compatibles. Pour ce faire, Cactus propose de raisonner sur les propriétés implantées par les micro-protocoles et sur les relations entre ces propriétés. Les relations pouvant exister entre deux propriétés p_i et p_j sont :

- le **conflit** : p_i et p_j sont en conflit, noté $con(p_i, p_j)$, si un service ne peut les satisfaire simultanément.
- la **dépendance** : p_i dépend de p_j, noté $dep(p_i, p_j)$, si la satisfaction de p_i dépend de celle de p_j.
- l'**indépendance** : p_i et p_j sont indépendants, noté $ind(p_i, p_j)$, si p_i et p_j ne sont ni en conflit, ni dépendantes.

Ces relations entre propriétés permettent d'établir des relations entre micro-protocoles. Chaque micro-protocole pouvant offrir plusieurs propriétés, on distingue quatre relations possibles entre deux micro-protocoles m_1 et m_2 :

- le **conflit** : m_1 et m_2 sont en conflit si deux au moins des propriétés qu'ils offrent sont en conflit.
- l'**indépendance** : m_1 et m_2 sont indépendants si les propriétés qu'ils offrent sont indépendantes.
- la **dépendance** : m_1 dépend de m_2 si ce dernier doit être présent dans le service pour permettre l'exécution de m_1.
- l'**inclusion** : m_1 inclut m_2 si m_1 implante toutes les propriétés implantées par m_2. Dans ce cas, les deux micro-protocoles ne doivent pas être utilisés dans le même service.

Exemple

Cet exemple illustre comment les relations entre propriétés peuvent être utilisées pour définir l'ensemble des associations correctes de micro-protocoles. L'exemple choisi est celui du service d'appel de procédure à distance de groupe (*group RPC*). Un tel service définit différentes propriétés : l'appel de procédure peut être bloquant ou non bloquant ; différents ordonnancements peuvent être respectés dans les exécutions d'appels de méthodes concurrents sur un serveur (fifo, total) ; les communications peuvent être fiables ou non ; les temps d'exécution peuvent être bornés ou non ; etc.

Les relations entre propriétés sont schématisées à l'aide d'un graphe (figure 4.3 (a)). La dépendance entre propriétés est schématisée par une flèche et le conflit entre propriétés est représenté par une boîte grisée. Par exemple, il est indiqué sur la figure 4.3 (a) que les propriétés d'ordonnancement (fifo et total) dépendent de la propriété de fiabilité sur les communications, ou encore qu'il n'est pas possible de créer un service de RPC de groupe qui soit à la fois bloquant et non-bloquant. Par ailleurs, la boîte en pointillés signifie que tout mécanisme de RPC nécessite au minimum une politique de blocage, et une implémentation des propriétés *acceptance* et *collation*. Le détail des propriétés et de leurs relations est donné dans [Hil98].

Les relations entre micro-protocoles sont schématisées sur la figure 4.3 (b). La relation d'inclusion entre micro-protocoles est représentée par une inclusion de boîte. Par ailleurs, la boîte en pointillés *User* permet de spécifier les micro-protocoles nécessaires pour obtenir un RPC de groupe. Il est intéressant de noter que les relations entre micro-protocoles peuvent différer de celles existant entre propriétés : en effet, respecter l'indépendance entre propriétés peut s'avérer

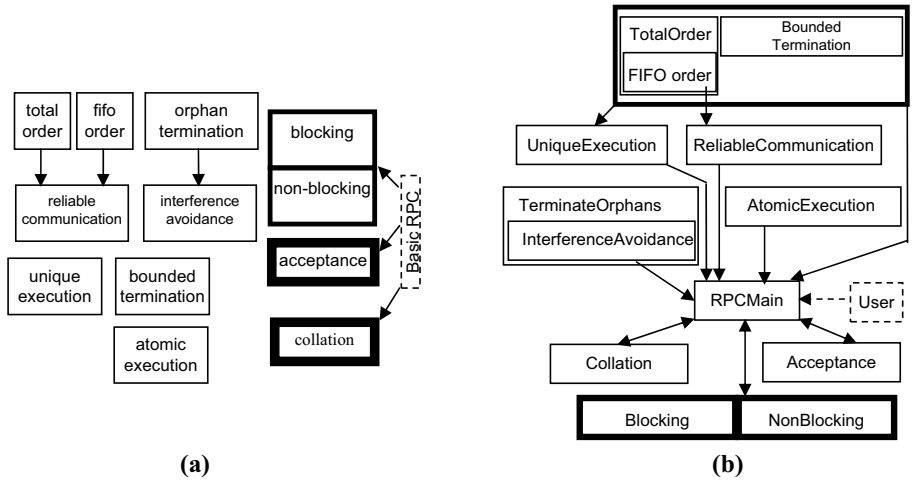

FIG. 4.3 – Graphes de relations entre **(a)** propriétés, et **(b)** micro-protocoles.

difficile à réaliser et coûteux à l'exécution. L'architecture des micro-protocoles peut donc imposer certaines contraintes sur les propriétés. On constate, par exemple, que le micro-protocole responsable de la propriété d'ordonnancement total requiert celui responsable de garantir l'exécution unique, et qu'il est en conflit avec celui responsable de garantir des temps d'exécution limités.

Synthèse

Cactus est une architecture permettant de composer des protocoles par assemblage de micro-protocoles. L'apport majeur de Cactus est un outil de vérification qui repose sur une méthode de définition de contraintes entre micro-protocoles permettant d'assurer que les assemblages réalisés sont "corrects". Notons néanmoins qu'il serait intéressant d'avoir un pouvoir d'expression des contraintes supérieur : il serait, par exemple, utile de pouvoir exprimer des relations temporelles entre les propriétés, ou encore des indications sur les performances, de manière à donner un moyen de choisir automatiquement une composition qui minimise la surcharge engendrée par l'intergiciel.

Cactus a un certain nombre de limitations. Tout d'abord, il utilise un modèle de composition simpliste : il n'est pas possible de construire des micro-protocoles composites par assemblages d'autres micro-protocoles. D'autre part, Cactus impose des contraintes architecturales : les seuls assemblages réalisables sont des piles de protocoles. De plus, l'architecture de ces protocoles est figée : elle consiste en un ensemble de micro-protocoles qui sont eux-mêmes composés de traitants d'événements. Enfin, une dernière limitation de Cactus est qu'il ne permet pas de reconfigurer dynamiquement les assemblages de micro-protocoles. En effet, les piles de protocoles construites ne possèdent pas de méta-niveau autorisant leur introspection et leur reconfiguration.

4.2.2 Appia

Motivations

Appia [MPR01] est un canevas logiciel destiné à la construction de piles de protocoles réseaux. Appia cible les applications ayant besoin de transmettre simultanément plusieurs types de données avec des exigences de qualité de service différentes. Un exemple typique d'application ayant ce besoin est une application multimédia : celle-ci utilise simultanément des protocoles de

communication différents pour chaque type de média : audio, vidéo et texte. Par ailleurs, il est nécessaire de synchroniser les différents flux. Pour ce faire, Appia propose de construire des piles de protocoles dans lesquelles certains protocoles sont partagés entre plusieurs protocoles. Par exemple, dans le cas de l'application multimédia, Appia permet de construire la pile représentée sur la figure 4.4 : les protocoles de détection de fautes et de synchronisation sont utilisés par les protocoles en charge des différents médias.

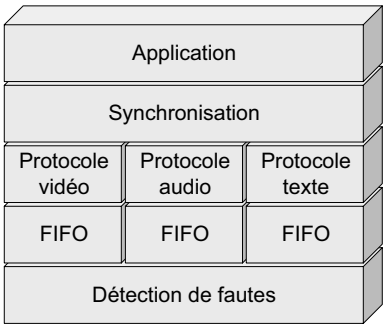

FIG. 4.4 – Architecture d'une pile de protocoles Appia.

Organisation d'une pile de protocoles

Une pile de protocoles est constituée d'un ensemble de protocoles. Chaque protocole communique avec les autres par l'intermédiaire d'événements. Pour ce faire, tous les protocoles implantent la même interface de gestion des événements. Les protocoles hébergent des *sessions*. Une session est une instance du protocole. Elle stocke des informations d'état ; par exemple, dans le cas d'un protocole d'ordonnancement causal, chaque session stocke une matrice de causalité. Un *canal de communication* est composé d'une pile de sessions, chaque session étant associée à un protocole de la pile.

Appia n'utilise qu'un seul thread par pile de protocoles. De fait, les échanges de messages entre sessions se font toujours dans un ordre FIFO. Par ailleurs, notons qu'Appia n'oblige pas les protocoles à effectuer des traitements différents sur les messages provenant de différents canaux. En d'autres termes, un protocole peut n'utiliser qu'une session pour l'ensemble des canaux de communication qui l'utilisent.

Notons que pour des raisons de performances, chaque protocole spécifie l'ensemble des types d'événements qui l'intéressent. En conséquence, lors de la création d'un canal de communication, Appia est capable de déterminer l'ensemble des protocoles que chaque type d'événement doit traverser. Cela évite ainsi que tous les événements traversent systématiquement tous les protocoles, ce qui permet d'améliorer les performances.

Synthèse

Appia est un système permettant de construire des piles de protocoles avec multiplexage de protocoles, ce qui le rend adapté aux applications nécessitant des transferts de données simultanés avec des exigences de qualité de service différentes.

Parmi les limitations d'Appia, citons le fait qu'il définit un modèle de composition simpliste qui ne permet de créer ni composants composites, ni composants partagés. Par ailleurs, il impose des contraintes architecturales. D'une part, il est uniquement possible de réaliser des piles de protocoles. D'autre part, le modèle de concurrence (i.e. un seul thread par pile de protocoles)

est imposé par le canevas. Si ce modèle facilite grandement l'implantation des protocoles, il peut néanmoins dégrader de façon considérable les performances, notamment si certains protocoles de la pile effectuent des entrées/sorties bloquantes. Par ailleurs, Appia ne définit aucun mécanisme de vérification des piles de protocoles réalisées. Enfin, les protocoles ne possèdent pas de méta-niveau programmable. En conséquence, il n'est pas possible de les reconfigurer dynamiquement.

4.3 Intergiciels de communication adaptables

Dans cette section, nous allons présenter quelques intergiciels de communication adaptables, c'est-à-dire des intergiciels qui sont dynamiquement reconfigurables. Le terme "reconfiguration dynamique" a été défini par Kramer et Magee dans [KM85, KM90] : une reconfiguration dynamique consiste à faire passer un système d'une configuration i à une configuration $i+1$, une *configuration* étant définie comme un ensemble d'entités et d'interconnexions entre ces entités.

Nous présentons tout d'abord trois intergiciels utilisant des technologies réflexives pour reconfigurer l'intergiciel : FlexORB, CompOSE|Q et QuO. Nous présentons, ensuite, deux intergiciels qui combinent l'utilisation de technologies réflexives et de programmation par composants : *dynamic*TAO et OpenORB.

4.3.1 FlexORB

FlexORB [OFT04] est un micro-ORB flexible destiné à la prise en charge des communications au sein d'environnements informatiques ubiquitaires. De tels environnements sont caractérisés par un haut niveau de dynamisme et la présence d'équipements mobiles aux ressources restreintes. Pour prendre en compte ces caractéristiques, les développeurs de FlexORB proposent de déployer un micro-ORB flexible s'exécutant sur NEVERMIND, un environnement d'exécution minimal et dynamiquement adaptable. Nous présentons tout d'abord NEVERMIND ; nous décrivons ensuite FlexORB.

L'environnement d'exécution NEVERMIND

NEVERMIND est un environnement d'exécution flexible et minimal basé sur un compilateur réflexif dynamique [POF01] et sur une couche logicielle, appelée HAL (*Hardware Abstraction Level*) [FSLM02], dont le rôle est de réifier les ressources matérielles de façon neutre, c'est-à-dire sans ajouter de sémantique. Le compilateur implante une chaîne de compilation ouverte et dynamique qui est utilisée pour ajouter de façon incrémentale des abstractions de plus haut niveau. En ce sens, NEVERMIND adopte la philosophie exo-kernel [EKO95], c'est-à-dire qu'il définit un environnement d'exécution minimal qui est étendu par ajout d'extensions dynamiquement compilées.

L'environnement d'exécution NEVERMIND fournit également un ensemble de drivers (clavier, réseau, etc.) et un protocole similaire à TFTP qui permet de charger les extensions à partir d'une bibliothèque de modules (*modules repository*).

La figure 4.5 représente la construction dynamique d'une application à l'aide de NEVERMIND. Au plus bas niveau, le HAL réifie les ressources physiques à l'aide d'un ensemble de composants tels le MMU, le firmware, ou encore les vecteurs d'interruptions. L'allocateur de mémoire repose sur le composant qui a la charge de la réification des accès à la mémoire physique. Cet allocateur mémoire minimal est à son tour utilisé par le compilateur dynamique. Une fois l'initialisation de ces composants terminée, NEVERMIND exécute un script qui permet d'initialiser les périphériques utiles tels le clavier, le réseau, ou encore l'écran. L'administrateur peut ensuite charger des scripts

(étape 1) (e.g. `tftp-get package-name`) qui sont dynamiquement compilés (étape 2) et exécutés : une requête est émise par le composant implantant le protocole TFTP (étape 3). Cette requête a pour résultat le téléchargement de code qui est dynamiquement compilé (étape 5) afin de constituer un nouveau service.

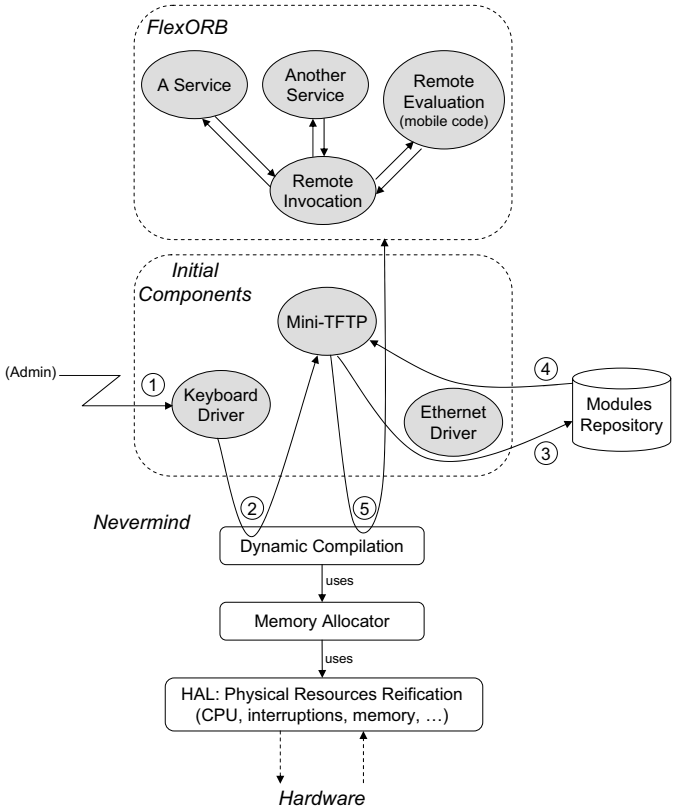

FIG. 4.5 – Architecture du micro-ORB FlexORB.

Le micro-ORB FlexORB

FlexORB est un micro-ORB qui définit un protocole d'invocations à distance basique. Les composants de FlexORB interagissent directement avec la carte réseau, ce qui le rend plus efficace que des ORB implantés en Java, par exemple. FlexORB implante un contrôle d'intégrité à base de checksum. Par ailleurs, il supporte la fragmentation des messages. Comme les ORB classiques, FlexORB utilise des pairs de stubs et skeleton intercalés entre les objets distants ; néanmoins, contrairement aux ORB classiques, ces stubs et skeletons ne sont pas compilés statiquement, mais dynamiquement. Il est ainsi possible d'exporter dynamiquement des interfaces d'objets afin de les rendre accessibles à distance, ce qui rend FlexORB particulièrement adapté aux environnements ubiquitaires.

FlexORB permet également de reconfigurer dynamiquement l'ORB. Supposons, par exemple, qu'il soit nécessaire de faire migrer un objet ; FlexORB permet de générer dynamiquement un *proxy de redirection* dont le rôle est de rediriger les requêtes à destination de l'objet ayant migré. Par ailleurs, FlexORB permet d'exécuter du code mobile, reçu sous forme sérialisée. Il est ainsi possible de reconfigurer dynamiquement les deux parties d'une liaison (déployées sur les sites clients et serveurs).

Synthèse

FlexORB est un micro-ORB destiné aux environnements informatiques ubiquitaires. Il repose sur l'utilisation de NEVERMIND, un environnement d'exécution minimal, dynamiquement adaptable. Le principal point fort de FlexORB est qu'il utilise des technologies réflexives afin de permettre la compilation dynamique de code. Ces techniques permettent de reconfigurer dynamiquement l'ORB pour prendre en compte, par exemple, la migration d'objets. Par ailleurs, du fait que FlexORB n'impose aucune contrainte architecturale et ne repose sur aucun service prédéterminé, ses performances sont nettement meilleures que celles d'ORB implantés en Java ou reposant sur divers abstractions logicielles.

En revanche, FlexORB ne fournit pas d'éléments d'architecture logicielle pour structurer les composants de l'intergiciel. Ainsi, il n'existe aucun moyen de spécifier les relations entre les différents composants déployés pour former un service donné. Par ailleurs, il n'est pas possible de construire des architectures logicielles hiérarchiques.

4.3.2 CompOSE|Q

CompOSE|Q (*Composable Open Software Environment with QoS*) [VDM+01] est un intergiciel destiné à la construction d'applications réparties ayant des exigences de qualité de service diverses. CompOSE|Q est une implémentation du TLAM (*Two Level Actor Machine*), une architecture réflexive à deux niveaux faisant intervenir des *acteurs*. Nous décrivons tout d'abord le TLAM, puis montrons son utilisation pour construire un canevas de construction de protocoles de communication dynamiquement reconfigurables.

Le TLAM

Le TLAM [Ven02] est une architecture utilisant des *acteurs* [Agh86]. Les acteurs sont des entités autonomes qui communiquent par envoi de messages, suivant un modèle "événement → réaction". Sur réception d'un message, un acteur peut (i) créer un nouvel acteur, (ii) changer son comportement, et (iii) envoyer un message à un acteur. Dans le modèle TLAM, un système est composé de deux types d'acteurs : les acteurs du niveau de base — qui implémentent la logique applicative —, et les acteurs du méta-niveau (appelés méta-acteurs) — qui fournissent des services aux acteurs, contrôlent leur comportement et gèrent les ressources du système.

L'objectif du modèle TLAM est de fournir un moyen de composer de manière correcte les propriétés offertes par les méta-acteurs, c'est-à-dire de garantir qu'elles n'interfèrent pas. Pour y parvenir, la stratégie adoptée est d'identifier des services de base — qui interagissent avec les acteurs du niveau de base et dont les contraintes sont bien connues —, et d'utiliser ces services pour spécifier et développer d'autres services à l'aide de méta-acteurs. Les détails du formalisme utilisé pour garantir l'absence d'interférence entre services peuvent être trouvés dans [Ven98].

Les trois services de base, représentés sur la figure 4.6, sont :

- la création à distance (*remote creation*) qui permet de créer des acteurs à distance.
- la capture d'état distribuée (*distributed snapshot*) dont le rôle est de capturer l'état des acteurs et de leurs communications.
- l'annuaire (*directory services*) qui autorise l'enregistrement des acteurs.

Des services comme la réplication, la migration, ou encore la découverte de ressources peuvent ensuite être construits en utilisant ces services de base. Ces services définissent leurs propres contraintes en termes d'interaction et d'interférence avec les autres services.

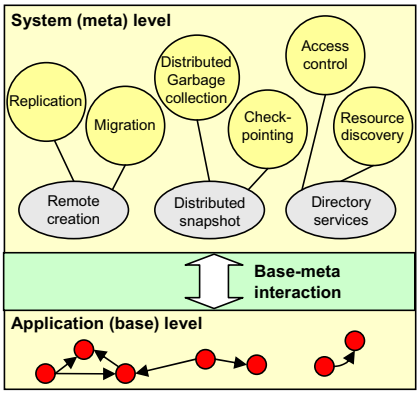

FIG. 4.6 – Le modèle TLAM.

CompOSE|Q

CompOSE|Q est une implémentation en Java du modèle TLAM. Il comprend :
- un ensemble de modules qui implantent les trois services de base décrits dans la section précédente. Ces modules permettent également la communication entre acteurs.
- un ensemble de services construits en utilisant les services de base : migration d'acteurs, ordonnanceur, service de nom, etc.
- des mécanismes permettant de garantir le respect de la qualité de service [VT95] par (i) gestion du placement réparti des agents et (ii) gestion des requêtes adressées aux agents.

Un canevas de construction de protocoles dynamiquement reconfigurables

Un mécanisme a été développé pour permettre la modification dynamique des protocoles de communication entre acteurs [GNV02]. Ce mécanisme, appelé RCF (*Reflective Communication Framework*), a pour objectif de construire des protocoles "corrects", c'est-à-dire :
- **sans interférence avec les autres protocoles :** deux protocoles peuvent être incompatibles : il est parfois nécessaire d'assurer un ordre d'exécution ou de n'utiliser qu'un des deux protocoles.
- **sans interférence avec les autres services de l'intergiciel :** un protocole peut être en conflit avec un service intergiciel. Ce peut être le cas, par exemple, entre un protocole dont le rôle est d'assurer un transfert fiable des messages échangés entre acteurs, et un service de migration d'acteurs.

L'architecture du RCF est représentée sur la figure 4.7. A chaque acteur, le RCF associe un méta-acteur, appelé *messenger*. Le messenger communique avec l'acteur via deux files de messages : *up* et *down*. Par ailleurs, chaque nœud du système héberge un gestionnaire de communication (*communication manager*) dont le rôle est de mettre des protocoles à disposition des messengers et de garantir que les compositions de protocoles requises par les messengers sont valides. Pour communiquer avec le gestionnaire de communications, un messenger utilise deux files de messages : *in* et *out*.

Un gestionnaire de communications est composé de deux types d'entités, implémentées par des agents : les protocoles et les coordonnateurs de messages (*message coordinator*). Un coordonnateur est associé à chaque message ; son rôle est d'assurer une composition de protocoles "correcte" qui satisfait les besoins exprimés par le messenger émetteur du message. Ces besoins sont exprimés, dans chaque message, par une liste de services nécessaires (*msl*). Notons que les protocoles étant mis en œuvre par des acteurs, il est possible de les ajouter ou de les retrancher

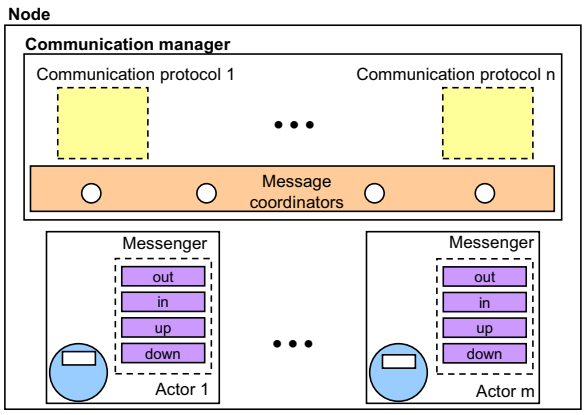

FIG. 4.7 – Architecture du RCF.

dynamiquement. Le coordonnateur utilise des informations fournies par les protocoles pour garantir que les compositions effectuées sont correctes. Ces informations sont des listes fournies par chaque protocole, spécifiant ses dépendances et ses incompatibilités avec les autres protocoles. Des détails sur l'utilisation de ces listes sont présentés dans [GNV02].

Synthèse

Le RCF propose un canevas de construction de protocoles dynamiquement reconfigurables intégré à l'intergiciel CompOSE|Q. Ces protocoles sont construits par un assemblage de protocoles élémentaires.

L'intérêt majeur du RCF réside dans le fait qu'il propose des mécanismes permettant d'exprimer les contraintes associées aux différents protocoles — dépendances et incompatibilités avec les autres protocoles —, et d'assurer que ces contraintes sont respectées. Cet outil a néamoins certaines limitations : il n'assure pas la compatibilité entre les protocoles composés et les services intergiciels. Par ailleurs, il n'est pas possible d'exprimer des contraintes distribuées sur les protocoles : les contraintes spécifiées concernent les protocoles s'exécutant localement.

La principale critique que l'on peut adresser au RCF est qu'il impose des contraintes architecturales fortes : chaque message doit être associé à un coordonnateur de messages — implémenté par un acteur —, avant d'être traité par des protocoles, eux-mêmes implémentés par des acteurs. Chaque message transite donc au minimum par trois acteurs. Ce mode de fonctionnement peut s'avérer incompatible avec des environnements aux ressources limitées. Par ailleurs, une autre limitation de CompOSE|Q est que l'architecture TLAM impose des dépendances envers trois services de base, ce qui peut s'avérer inutile (et coûteux) dans certains cas.

4.3.3 QuO

QuO [ZBS97, LSZ+01] est un intergiciel destiné à la construction d'applications réparties à objets ayant des exigences de qualité de service (QoS). Il permet de spécifier (i) les besoins en QoS de l'application, (ii) les éléments du système qui doivent être observés et contrôlés pour mesurer et fournir la QoS requise, et (iii) le comportement à adopter pour adapter l'intergiciel et l'application aux variations de QoS. Ces caractéristiques font de QuO un intergiciel réflexif : il fournit le contrôle nécessaire pour permettre l'adaptation de sa structure en cours d'exécution.

Architecture

Il existe différentes implémentations de QuO. La plus aboutie repose sur l'utilisation de l'intergiciel CORBA. Son architecture est représentée sur la figure 4.8.

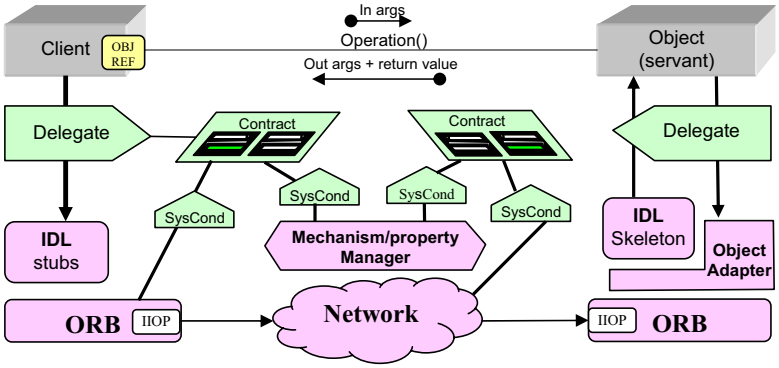

FIG. 4.8 – Architecture de QuO.

De façon simplifiée, QuO peut être considéré comme une couche logicielle située entre CORBA et les applications. Cette couche est composée des éléments suivants :

- les **contrats** (*contracts*) permettent de spécifier le niveau de service désiré par un objet client, le niveau de service qu'un objet serveur peut fournir, un ensemble de régions — qui sont définies comme des états possibles au regard de la QoS —, et les actions à effectuer lorsque le niveau de QoS change (i.e. lors des changements de régions). Les contrats sont spécifiés à l'aide d'un langage appelé CDL (*Contract Description Language*).
- les **délégués** (*delegates*) sont des intermédiaires placés entre le stub et l'objet client, et entre le skeleton et l'objet serveur. Les délégués prennent des décisions d'adaptation à l'aide des informations fournies par les contrats sur l'état de la QoS. Ces décisions sont spécifiées à l'aide d'un langage appelé SDL (*Structure Description Language*). Le code des délégués est généré par un compilateur à l'aide des descriptions CDL et SDL.
- les **objets de condition** du système (*system condition objects*) fournissent une interface vers les ressources, les objets applicatifs et les constituants de l'ORB que les contrats observent et contrôlent.

Reconfiguration dynamique

QuO définit deux mécanismes de déclenchement d'adaptations : en ligne (*in-band*) et hors-ligne (*out-of-band*). Le déclenchement en ligne est effectué par les délégués lors des invocations et des retours de méthodes. Ces derniers vérifient l'état de la QoS auprès des contrats, et choisissent un comportement adapté : filtrage des paramètres de l'invocation, choix d'une méthode alternative, exécution d'une fonction locale, etc. Le déclenchement hors-ligne est provoqué par les contrats et les objets de condition du système. Lors de changements significatifs des conditions d'exécution, les objets de condition du système déclenchent une évaluation (dite asynchrone) des contrats. S'il y a changement de région, les contrats déclenchent l'adaptation de l'application.

Synthèse

QuO permet de construire des applications ayant des exigences de QoS. C'est un intergiciel qui fait office de couche de médiation entre les applications et l'ORB. Le développeur spécifie les

contrats qui doivent être respectés entre les objets de l'application et l'intergiciel, ainsi que des politiques d'adaptation. QuO se charge de respecter ces contrats.

Parmi les points forts de QuO, citons le fait qu'il utilise des technologies réflexives afin de reconfigurer dynamiquement l'intergiciel. Ce méta-niveau est paramétrable à l'aide de langages permettant d'exprimer la QoS requise par les clients et fournie par les serveurs. Ceci permet une séparation nette du code applicatif et du code d'administration. Par ailleurs, il établit une distinction intéressante entre adaptations synchrones (lors d'une invocation) et asynchrones (hors invocations).

QuO a plusieurs limitations. D'une part, les possibilités du méta-niveau sont limitées et figées : les politiques d'adaptation ne semblent pas pouvoir être modifiées dynamiquement. De plus, il n'est pas possible de rajouter dynamiquement des fonctions à l'intergiciel. En effet, QuO ne fournit que le support pour le choix entre des alternatives fournies par l'ORB. Une autre limitation de QuO est que son champ d'application est assez restreint : il permet uniquement de gérer des interactions client-serveur au sein d'un ORB.

4.3.4 *dynamic*TAO

*dynamic*TAO [RKC99, KRL+00, KCBC02] est une extension de TAO [SC99], un ORB flexible et extensible. Il a été développé en C++ et met en œuvre un certain nombre de patrons de conceptions orientés objets. Il utilise notamment le patron *strategy* [GHJV95] qui permet de séparer les différents aspects de l'ORB : concurrence, démultiplexage des requêtes, gestion des connexions, etc. Un fichier de configuration permet de spécifier les stratégies que l'ORB doit implémenter. TAO ayant été conçu pour les applications temps réel dans des systèmes tels que ceux que l'on embarque dans les avions, il n'a pas été prévu de mécanismes de reconfiguration dynamique de l'ORB : il n'est pas possible de modifier dynamiquement les stratégies utilisées.

*dynamic*TAO utilise le même concept de "stratégies", mais fournit des mécanismes permettant de modifier dynamiquement ces stratégies. Ces mécanismes permettent l'introspection et la reconfiguration de l'ORB, mais aussi des objets serveurs (*servants*) accédés par les objets clients via l'ORB.

Architecture

Pour introspecter et reconfigurer l'intergiciel, *dynamic*TAO utilise des entités appelées configurateurs de composants (*component configurators*). Un configurateur de composant a la charge de gérer les dépendances entre un composant et les autres composants du système. Nous détaillons le fonctionnement des configurateurs de composants dans la section suivante.

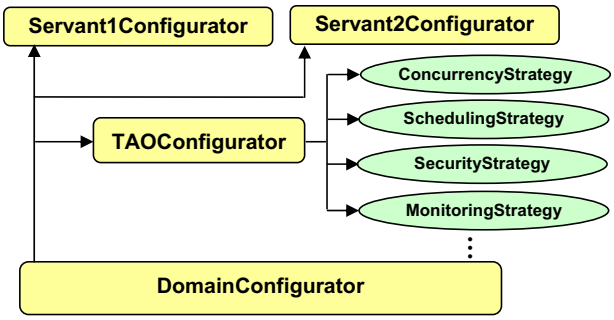

FIG. 4.9 – Organisation des configurateurs de composants de *dynamic*TAO.

Comme nous l'avons illustré sur la figure 4.9, chaque processus exécutant *dynamic*TAO possède un configurateur de composant particulier, appelé *DomainConfigurator*. Celui-ci est responsable des dépendances entre les ORB et les objets serveurs accédés via ces ORB. Par ailleurs, chaque ORB possède un configurateur de composant appelé *TAOConfigurator*. Celui-ci permet de gérer les dépendances entre l'ORB et les différentes stratégies qu'il utilise. Enfin, notons que certaines stratégies peuvent posséder un configurateur de composant afin de gérer leurs dépendances avec les autres stratégies, avec les instances d'ORB, ou encore avec les connexions des clients qui dépendent de la stratégie.

Gestion des dépendances entre composants

La notion de dépendance entre composants est fondamentale dans *dynamic*TAO. Celle-ci est en effet à la base des processus de configuration et de reconfiguration. Dans [KC00, KYH+01], les auteurs distinguent deux types de dépendances :

- les **pré-requis** sont les dépendances d'un composant par rapport aux composants matériels et logiciels permanents dans le système.
- les **dépendances dynamiques** sont les dépendances d'un composant par rapport aux composants présents dans le système à un moment donné. Chaque composant peut dépendre de composants "serveurs" — qui lui fournissent des services —, et des composants "clients" peuvent dépendre de lui — ceux auxquels il fournit des services.

Les pré-requis sont exprimés dans un fichier suivant un format appelé SPDF (*Simple Prerequisite Description Format*). Quant aux dépendances dynamiques, elles sont gérées par les configurateurs de composants. Pour ce faire, ces derniers implémentent des méthodes qui permettent de manipuler dynamiquement la liste des dépendances (ajout ou retrait de dépendances) et de définir des actions à effectuer lors de changements dans les dépendances des composants.

Reconfiguration dynamique dans *dynamic*TAO

Dans ce paragraphe, nous explicitons le rôle des configurateurs de composants dans la reconfiguration dynamique. Nous présentons ensuite l'architecture globale mise en œuvre dans *dynamic*TAO pour permettre sa reconfiguration dynamique.

Un configurateur de composant permet de remplacer le composant de façon sûre. Il permet, en effet, de résoudre deux problèmes fondamentaux :

- **assurer que l'ancien composant n'est pas — et ne sera plus — utilisé**. Pour ce faire, le configurateur utilise la liste des dépendances pour informer les composants dépendant du composant remplacé que celui-ci ne sera plus accessible.
- **effectuer un transfert d'état de l'ancien vers le nouveau composant**. Cette opération n'est pas toujours aussi triviale que ce qu'elle peut paraître, et nécessite donc un certain travail du développeur du configurateur de composant. Par exemple, il est décrit dans [KRL+00] le cas du remplacement d'un composant responsable de la gestion des threads.

Il est important de noter qu'un configurateur de composant garantit le remplacement sûr du composant qu'il gère, mais que rien n'est en revanche garanti concernant l'intégrité du système distribué après le remplacement.

Pour faciliter les reconfigurations dynamiques, *dynamic*TAO fait intervenir plusieurs entités qui sont représentées sur la figure 4.10.

Le répertoire persistant (*persistent repository*) permet de stocker les implémentations de composants sur le système de fichiers local. Le service de reconfiguration dynamique (*dynamic service configurator*) fournit une interface permettant de manipuler les abstractions définies

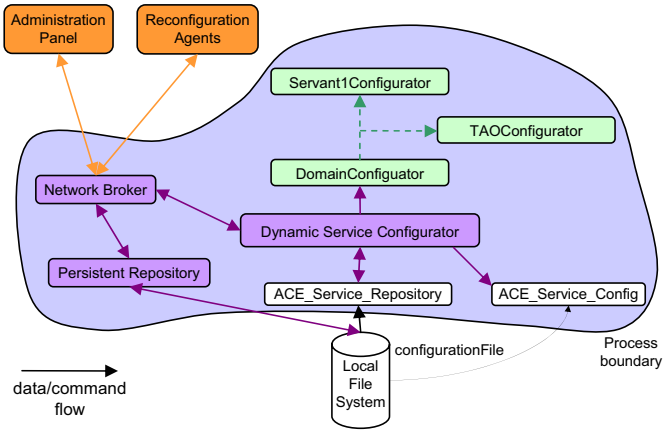

FIG. 4.10 – Les composants de *dynamic*TAO.

par *dynamic*TAO. Il délègue les opérations de configuration et de reconfiguration aux différents configurateurs de composants, ainsi qu'à des services fournis par ACE [SBS93]. Ces derniers sont notamment responsables de la configuration de l'ORB, de la création des liaisons dynamiques, etc. Enfin, le composant réseau (*network broker*) permet l'accès au service de configuration dynamique via le réseau.

Plusieurs outils d'interfaçage avec les administrateurs du système ont été réalisés. Le plus intéressant est un outil basé sur l'utilisation d'agents [KGCM00]. Celui-ci permet d'effectuer à distance un ensemble d'opérations sur les différents sites hébergeant une instance de *dynamic*TAO. Les bénéfices apportés par l'utilisation d'agents sont la parallélisation de la reconfiguration et l'optimisation des flots de reconfiguration obtenue en adaptant la topologie de propagation des agents aux caractéristiques physiques du réseau. Cependant le système ne permet pas de garantir que le même ensemble d'actions est effectué sur tous les sites, ce qui exclut son utilisation pour la reconfiguration de fonctions "délicates" de l'ORB, comme les protocoles de communication.

Synthèse

*dynamic*TAO est une évolution de l'ORB TAO qui offre des possibilités de reconfiguration dynamique de sa structure. Le point fort de *dynamic*TAO est qu'il utilise une technologie logicielle réflexive à base de composants. La structure à composants, héritée de TAO, permet de séparer de façon nette les différentes fonctions de l'ORB. L'utilisation de la réflexivité permet d'avoir un contrôle sur l'ORB en cours d'exécution : les configurateurs réifient les dépendances entre composants, ce qui est utile pour garantir qu'un composant reconfiguré n'est pas en cours d'utilisation. Le modèle de dépendances entre composants est intéressant : il réalise une synthèse du travail effectué par la communauté du génie logiciel (dépendances entre composants) et du travail effectué dans le domaine de la gestion de la QoS (dépendances d'un système avec les caractéristiques physiques de l'environnement).

Parmi les limitations de *dynamic*TAO, notons qu'il impose des contraintes architecturales : par exemple, un ORB impose la même configuration aux différents servants qu'il héberge. Si ces derniers ont des besoins différents, il est nécessaire de déployer deux instances d'ORB dans le même processus. Notons que pour résoudre ces problèmes, les développeurs ont conçu LegORB [RMKC00], un ORB entièrement à composants. LegORB permet, par exemple, de construire des assemblages de composants minimaux autorisant l'exécution de l'ORB sur des équipements embarqués.

Une autre limitation est le modèle de réflexivité utilisé qui se limite à de la réflexivité structurale : seules les dépendances entre composants d'un même serveur sont réifiées. Il n'est pas possible d'étendre le méta-niveau pour qu'il prenne en compte les dépendances distribuées entre composants ou encore pour qu'il permette de réifier d'autres éléments comme les ressources, par exemple. En conséquence, *dynamic*TAO a des capacités de reconfiguration limitées.

Enfin, une dernière limitation de *dynamic*TAO est son modèle de composition qui est simpliste. Ce dernier ne permet de créer ni composants composites, ni composants partagés.

4.3.5 OpenORB

OpenORB [BCA+01] est un intergiciel réflexif à composants développé à l'Université de Lancaster. L'objectif d'OpenORB est de fournir un canevas de construction d'intergiciels dynamiquement reconfigurables. Il a été spécialement conçu pour les applications multimédia et les applications s'exécutant dans des environnements mobiles.

Architecture

Une instance locale d'OpenORB est un assemblage de composants. Cet assemblage est déterminé lors de la construction de l'intergiciel ; il peut ensuite être modifié en cours d'exécution afin de reconfigurer dynamiquement l'intergiciel. Pour ce faire, chaque composant possède un méta-espace permettant son inspection et son adaptation. Ce méta-espace supporte la réflexivité *structurelle* — représentation de l'architecture du composant — et la réflexivité *comportementale* — contrôle des activités du composant et gestion des ressources.

Mise en œuvre

La première implémentation d'OpenORB a été réalisée à l'aide du langage interprété Python. Dans cette section, nous décrivons la seconde implémentation d'OpenORB. Celle-ci a été réalisée à l'aide du modèle de composant OpenCOM, présenté au chapitre 2.

Les composants sont organisés au sein de canevas (CF pour *Component Framework*). La notion de CF a été ajoutée de manière à permettre une sorte de composition de composants. En effet, le modèle OpenCOM ne permet pas de créer des composants composites (i.e. des composants encapsulant d'autres composants). L'absence de composition est un choix délibéré des concepteurs qui estiment qu'aucun modèle de composition n'est suffisamment général pour pouvoir s'appliquer à l'ensemble d'un système. Le but des CF est de regrouper des composants appartenant à un même domaine (liaisons, communications, etc.). Un CF définit une interface abstraite et gère plusieurs implémentations de cette interface, fournies par différents composants OpenCOM. Par ailleurs, un CF fournit une interface pour la reconfiguration dynamique des composants qu'il gère. Son rôle est de permettre des reconfigurations tout en préservant l'intégrité de l'intergiciel.

La figure 4.11 représente l'organisation des CF dans OpenORB. Une instance locale d'OpenORB est représentée par un CF appelé *top-level CF*. L'intergiciel construit doit respecter une architecture en trois couches : liaisons, communications et ressources. Le respect de cette contrainte est assuré par le *top-level CF* qui contraint chaque CF à n'utiliser que des CF de la même couche ou des couches inférieures.

Reconfiguration dynamique

OpenORB autorise des reconfigurations dynamiques de son architecture. Des expériences ont

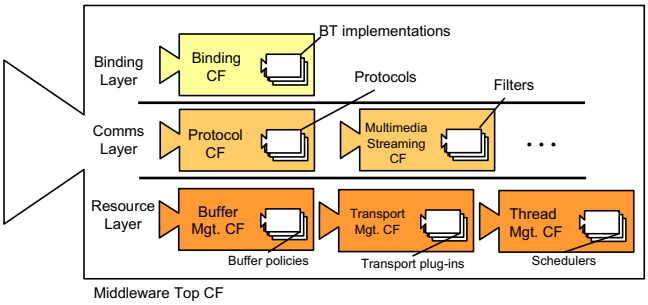

FIG. 4.11 – Les canevas de composants d'OpenORB.

notamment été menées sur la gestion de la QoS. Celles-ci consistent à introduire dynamiquement un ensemble de composants permettant (i) d'observer le système en cours d'exécution, et (ii) d'exploiter les résultats des composants d'observation pour déterminer et effectuer les reconfigurations dynamiques nécessaires. L'insertion de ces composants est rendue possible par la structure réflexive d'OpenORB.

Ces reconfigurations sont facilitées par un système de verrou fourni par OpenCOM qui garantit un accès en exclusion mutuelle aux interfaces serveurs d'un composant. Il est ainsi aisé d'assurer qu'un composant n'est pas en cours d'utilisation au moment de sa reconfiguration. Par ailleurs, chaque CF peut imposer un certain nombre de contraintes sur les reconfigurations dynamiques des composants qu'il gère. Il peut, par exemple, exiger qu'ils implémentent une interface particulière.

Cependant, comme il est souligné dans [BBI+00], ces mécanismes ne sont pas suffisants pour garantir l'intégrité de l'intergiciel après la reconfiguration. En effet, il est possible d'effectuer des changements arbitraires sur l'architecture de l'intergiciel. De tels changements peuvent porter atteinte à l'intégrité du système : par exemple, le changement d'une pile de protocoles sur un site requiert souvent un changement analogue sur un autre site de l'application distribuée. Pour garantir l'intégrité du système après reconfiguration, les auteurs proposent l'utilisation d'un langage de description d'architecture. [BBI+00] montre comment l'utilisation du système Aster[1] permettrait de contraindre les adaptations d'OpenORB. Ces propositions sont intéressantes, mais elles n'ont pas été implantées.

Synthèse

OpenORB est un canevas de composants pour la construction d'intergiciels dynamiquement reconfigurables. Un intergiciel réalisé avec OpenORB est un assemblage de composants OpenCOM. L'atout majeur d'OpenORB est qu'il utilise une technologie réflexive à base de composants : chaque composant possède un méta-espace permettant son inspection et son adaptation. Il est, par exemple, aisé de connaître les dépendances d'un composant envers les autres composants de l'intergiciel. De plus, l'utilisation de composants communiquant via des interfaces définies permet de mettre en place des mécanismes — comme les verrous associés aux interfaces serveurs — qui facilitent les reconfigurations dynamiques.

Parmi les limitations d'OpenORB, citons le fait que le modèle de composition utilisé ne permet pas de créer des composants partagés. En revanche, le modèle est beaucoup plus complet que dans les autres intergiciels présentés dans ce chapitre. Il permet notamment de créer des composants composites. Une autre limitation d'OpenORB est que l'architecture de son méta-niveau

[1] Aster est décrit au chapitre 3.

est assez rigide : il n'est, par exemple, pas possible d'ajouter des méta-modèles. Par ailleurs, une limitation d'OpenORB est qu'il impose des contraintes architecturales : tout intergiciel doit être composé en trois couches, ce qui n'est pas forcément adapté aux environnements contraints. Enfin, notons que bien qu'applicable à différents types d'intergiciels de communication, l'architecture d'OpenORB a majoritairement été utilisée pour la construction de personnalités synchrones de type ORB.

4.4 Synthèse

Dans ce chapitre, nous avons présenté divers intergiciels de communication que nous avons classés en deux catégories : les intergiciels dédiés à la construction de piles de protocoles et les intergiciels adaptables. Ces intergiciels diffèrent par leur champ d'application, les modèles de composition qu'ils définissent, leurs possibilités de reconfiguration, les contraintes architecturales qu'ils imposent et les outils qu'ils définissent pour vérifier les architectures déployées.

Concernant le champ d'application, Cactus et Appia se limitent à la création de piles de protocoles. QuO, *dynamic*TAO et FlexORB sont des implantations de courtiers à objets. L'intergiciel le plus flexible est OpenORB. Celui-ci n'implante pas un modèle de communication particulier, mais fournit un canevas de composants destiné à la construction d'intergiciels dynamiquement reconfigurables. Néanmoins, OpenORB a majoritairement été utilisé pour la construction de personnalités synchrones de type ORB.

Tous les intergiciels présentés dans ce chapitre sont organisés à l'aide de composants. Néanmoins, la plupart des intergiciels définissent des modèles de composition simplistes qui ne permettent pas de structurer l'intergiciel de façon hiérarchique. La seule exception est OpenORB qui permet de créer des structures hiérarchiques à l'aide de composants composites. Par ailleurs, aucun modèle ne prend en charge les composants partagés.

Les intergiciels diffèrent également selon leurs possibilités de reconfiguration dynamique. Cactus et Appia ne permettent pas d'associer de méta-niveau aux composants des piles de protocoles. De fait, il est impossible de reconfigurer dynamiquement les piles de protocoles. Les autres intergiciels présentés dans ce chapitre ont la particularité de mettre en œuvre des mécanismes réflexifs pour autoriser des reconfigurations dynamiques de l'intergiciel. La réflexivité permet de séparer le niveau de base — qui met en œuvre les fonctions de l'intergiciel —, des méta-niveaux qui permettent d'effectuer des opérations sur le niveau de base. La réflexivité offre un moyen puissant d'accéder à la structure interne de l'intergiciel, aussi bien pour l'introspecter que pour la modifier. Néanmoins, les intergiciels ne font pas tous le même usage de la réflexivité et n'ont donc pas les mêmes possibilités de reconfiguration dynamique. Ainsi, QuO limite le méta-niveau à la mise en œuvre de politiques d'adaptation figées. FlexORB permet de mettre en œuvre des politiques d'adaptation plus complexes, basées sur l'utilisation d'un compilateur dynamique. *dynamic*TAO limite le méta-niveau à de la réflexivité structurale : seules les dépendances entre composants sont réifiées. Il n'est, par exemple, pas possible d'étendre le méta-niveau pour réifier les ressources consommées au sein de l'intergiciel. Le système ayant les capacités réflexives les plus étendues est OpenORB. Celui-ci permet d'associer aux composants un méta-niveau prenant en charge la réflexivité structurale et comportementale. Néanmoins, il ne permet pas d'ajout arbitraire de fonctions.

Par ailleurs, notons que les intergiciels présentés imposent souvent un certain nombre de contraintes architecturales. Par exemple, Cactus fige totalement l'architecture des protocoles (sous forme de micro-protocoles encapsulant des traitants d'événements). Appia impose un modèle de concurrence avec un seul thread par pile de protocoles. Le RCF impose que chaque message soit associé à un coordonnateur de messages — implémenté par un acteur —, avant

d'être traité par des protocoles, eux-mêmes implémentés par des acteurs. Chaque message transite donc au minimum par trois acteurs. OpenORB, bien que définissant le modèle de composition le plus flexible et ayant le plus vaste champ d'application, impose certaines contraintes : tout intergiciel doit être composé en trois couches. Dans de nombreux cas, ces contraintes peuvent engendrer des pertes de performances.

Enfin, les intergiciels présentés différent par les outils d'aide à la construction qu'ils fournissent. Tous permettent de déployer des configurations distribuées. Néanmoins peu d'entre eux offrent des outils pour la vérification des configurations déployées. Seuls Cactus et le RCF font exception. Cactus propose un outil de vérification reposant sur un formalisme permettant de composer des protocoles de façon "correcte". Ce formalisme comprend une méthode de définition de contraintes entre protocoles permettant de déterminer les différentes compositions réalisables. Le RCF définit des mécanismes d'expression des contraintes associées avec les différents protocoles — dépendances et incompatibilités avec les autres protocoles —, et d'assurer que ces contraintes sont respectées. Ces outils ont néamoins certaines limitations. Il ne permettent pas, par exemple, d'exprimer des contraintes distribuées sur les protocoles : les contraintes spécifiées concernent uniquement les protocoles assemblés localement pour composer la pile.

Deuxième partie

Dream : un canevas logiciel à composants pour la construction d'intergiciels de communication

Chapitre 5

Présentation générale

Sommaire

La deuxième partie de ce manuscrit est consacrée à la description de DREAM, un canevas logiciel à composants pour la construction d'intergiciels de communication. Ce chapitre donne une présentation générale de DREAM.

5.1 Motivations

5.1.1 Limitations des intergiciels de communication existants

Le chapitre 4 a présenté un certain nombre de travaux portant sur la construction d'intergiciels de communication adaptables. Le constat que l'on peut dresser est double :

- les intergiciels présentés imposent diverses abstractions aux développeurs d'applications. Ces abstractions sont de deux natures : *fonctionnelles* et *architecturales*.
- les intergiciels présentés sont peu, voire pas administrables.

Concernant les **abstractions fonctionnelles**, les intergiciels présentés sont majoritairement destinés à la construction d'un *seul* type d'intergiciel de communication : Cactus et Appia sont des canevas permettant de construire des protocoles de communication ; *dynamic*TAO, OpenORB[1] et QuO sont des implantations de courtiers à objets (ORB) ; de nombreux autres intergiciels — non présentés au chapitre 4 car peu configurables et non reconfigurables — implantent une

[1] Les concepteurs d'OpenORB sont actuellement en train d'étudier l'applicabilité des concepts développés au sein d'OpenORB à d'autres domaines : routeurs programmables, réseaux d'overlay, intergiciel pour Grilles, etc.

sémantique de communication asynchrone figée : JMS [BEA05, jor05a, son02], événement/réaction [BdPF+99], publication/abonnement [CRW01, SBC+98]. Ces intergiciels reposent sur des fonctions similaires : gestion du cycle de vie des messages, sérialisation/désérialisation, émission/-réception à l'aide de divers protocoles (TCP, UDP, HTTP, SOAP, etc.), routage, etc. Néanmoins, ils implantent tous une interface de programmation figée à laquelle les développeurs d'applications doivent se conformer.

Concernant les **abstractions architecturales**, les intergiciels de communication présentés ont des structures rigides et contraignantes. Par exemple, OpenORB impose une structure en 3 couches de l'ORB (liaisons, communications et ressources) ; Appia impose un modèle de concurrence rigide (une pile de protocole est mono-threadée) ; Cactus impose un modèle de composition de micro-protocoles à plat (pas de construction possible de micro-protocoles composites), etc. Il en résulte que ces intergiciels ne sont souvent adaptés qu'à un environnement d'exécution particulier (PC standard dans la plupart des cas). Les utiliser sur des équipements aux ressources (mémoire et CPU) restreintes nécessite une ré-ingénierie de leur code.

Par ailleurs, les intergiciels présentés sont **peu, voire pas administrables**. Par exemple, Cactus et Appia ne sont pas administrables. Il est impossible de modifier dynamiquement les configurations en cours d'exécution. Les autres intergiciels présentés ont des possibilité de reconfiguration dynamiques limitées et figées. Par exemple, QuO ne permet pas de rajouter dynamiquement des fonctions à l'intergiciel. Il permet uniquement de choisir une des alternatives fournies par l'ORB. Ces alternatives sont définies à l'aide de politiques d'adaptation qui ne sont pas modifiables dynamiquement. L'intergiciel qui offre les possibilités d'administration les plus évoluées est OpenORB : ces fonctions reposent sur des méta-objets implantant des fonctions réflexives permettant d'agir sur la structure et le comportement de l'ORB. Néanmoins, OpenORB fixe les méta-objets que possèdent les composants de l'ORB. Il n'est pas possible d'en ajouter pour prendre en compte d'autres aspects d'administration.

5.1.2 Vers une suppression des abstractions : l'exogiciel

Notre ambition est de construire des intergiciels de communication hautement administrables n'imposant aucune des abstractions sus-citées (i.e. fonctionnelles et architecturales). Pour ce faire, nous proposons de nous inspirer de la philosophie *exo-noyaux* [EKO95] présentée par Engler et al. pour la construction de systèmes d'exploitation. Nous proposons de construire des canevas logiciels à composants constitués de trois parties :

- un *modèle de composants* généraliste permettant de construire des architectures flexibles et administrables.
- une *bibliothèque de composants* pouvant être assemblés pour former des *personnalités*. Une bibliothèque contient des composants pour un domaine applicatif déterminé. Dans le cadre de cette thèse, nous présentons une bibliothèque de composants pour la construction d'intergiciels de communication[2].
- des *outils de gestion de configuration* permettant de gérer différentes étapes du cycle de vie des personnalités construites à l'aide de la bibliothèque : description, configuration, déploiement, administration, etc.

[2]Nous présentons également une bibliothèque de composants pour la construction de systèmes d'observation au chapitre 10.

5.2 Le canevas DREAM

DREAM est un canevas constitué des trois éléments cités dans la section précédente : modèle de composants, bibliothèque et outils de gestion de configuration. Dans la suite de cette section, nous décrivons succinctement ces trois éléments.

5.2.1 Le modèle de composants

Le modèle de composants doit être le plus généraliste possible afin de permettre la modélisation de diverses formes de systèmes. Il est impératif qu'il possède un modèle de *composition étendu* et qu'il autorise des *formes arbitraires de contrôle* des architectures déployées. Par modèle de composition étendu, nous entendons la possibilité de créer des structures hiérarchiques avec partage de composants. Par formes arbitraires de contrôle, nous désignons la possibilité d'associer à chaque composant un méta-niveau arbitrairement complexe, composé de méta-objets implantant diverses formes de contrôle (cycle de vie, liaison, contenu, ressources, etc.). L'état de l'art dressé au chapitre 2 nous a montré que le modèle de composants FRACTAL répondait à ces exigences. Aucun autre modèle ne permet de créer des composants partagés, ni d'associer un méta-niveau arbitrairement complexe aux composants.

Dans le cadre de cette thèse, nous avons effectué un certain nombre d'extensions au modèle FRACTAL concernant la gestion des liaisons entre composants, la journalisation des événements se produisant au sein de l'application, et les activités des composants. Ces extensions, bien que développées dans le cadre de DREAM, sont applicables à toute application FRACTAL. La plus importante de ces extensions est un canevas pour la gestion des activités des composants. Cette extension est décrite dans le chapitre 7. Nous proposons de considérer deux types de composants : les *composants actifs* et les *composants passifs*. Les composants actifs définissent des tâches à exécuter. Ces tâches permettent au composant de posséder son propre flot d'exécution. Au contraire, les composants passifs ne peuvent effectuer d'appels sur leurs interfaces clients que dans une tâche d'un composant appelant une de leurs interfaces serveurs.

Pour qu'une tâche soit exécutée, un composant actif doit l'enregistrer auprès d'un gestionnaire d'activités (*activity manager*). Les gestionnaires d'activités sont des composants partagés qui encapsulent des tâches et des ordonnanceurs (*schedulers*). Les ordonnanceurs ont la charge d'associer des tâches de haut niveau à des tâches de bas niveau. Les tâches de plus haut niveau sont les tâches applicatives. Les tâches de plus bas niveau encapsulent des threads Java.

5.2.2 La bibliothèque de composants

La bibliothèque DREAM contient des composants implantant diverses fonctions pouvant être assemblées pour former des intergiciels de communication. La bibliothèque DREAM se décompose en deux parties : un ensemble de composants permettant de manipuler des *messages* et un canevas pour la création de protocoles et de sessions à l'aide des composants de la bibliothèque.

La bibliothèque

Les composants de la bibliothèque traitent des *messages* qu'ils reçoivent par des interfaces particulières appelées *entrées* et qu'ils délivrent sur des interfaces appelées *sorties*. Les messages sont des objets Java encapsulant des *chunks* et des sous-messages. Chaque chunk est un objet Java implantant des accesseurs (*getter*) et des mutateurs (*setter*). Par exemple, un message qui doit être causalement ordonné doit posséder un chunk `CausalChunk` qui définit des méthodes pour consulter/modifier une horloge matricielle.

La bibliothèque possède des composants encapsulant les fonctions que l'on trouve de façon commune dans les différents intergiciels de communication. Ces composants doivent avoir un grain suffisamment fin pour être facilement réutilisables dans plusieurs personnalités. Suivent des exemples de composants : les *files de messages* servent à stocker les messages ; les *transformateurs* modifient les messages reçus (ajout/suppression/modification de chunks) ; les *pompes* ont une activité qui consiste à récupérer un message sur leur entrée et à le délivrer sur leur sortie ; les *routeurs* routent les messages reçus sur leur entrée sur une ou plusieurs de leurs sorties ; les *sérialiseurs* (resp. *désérialiseurs*) prennent en charge la sérialisation (resp. désérialisation) des messages ; les *canaux* permettent l'échange de messages entre différents espaces d'adressages à l'aide de sockets TCP, UDP, Multicast ; les *composants de protocole* sont en charge de faire respecter un protocole par n processus — par exemple, un ordonnancement causal des messages échangés.

Le canevas pour la création de liaisons

La bibliothèque DREAM définit des abstractions facilitant l'établissement de liaisons entre composants. DREAM permet de construire des protocoles organisés en pile ou en graphe. Chaque protocole utilise des services fournis par les protocoles de niveaux inférieurs dans la pile (ou dans le graphe). Un protocole permet l'ouverture de *sessions* au sein desquelles des messages peuvent être échangés. Dans ce document, nous illustrons les concepts définis par le canevas DREAM en détaillant l'établissement d'un canal de communication TCP/IP entre deux composants.

5.2.3 Les outils de gestion de configuration

Les outils de gestion de configuration doivent permettre la prise en charge des différentes étapes du cycle de vie des personnalités construites à l'aide de la bibliothèque de composants : description, configuration, vérification, déploiement, administration. Ces outils s'organisent autour d'un langage de description d'architectures. L'état de l'art dressé au chapitre 3 nous a montré qu'il existe deux catégories d'outils de gestion de configuration : les outils destinés au déploiement et à la configuration des architectures décrites et les outils destinés aux vérifications sur les architectures. Aucun outil disponible aujourd'hui ne prend en charge ces deux aspects.

Dans le cadre de cette thèse, nous proposons une suite d'outils reposant sur le langage de description d'architectures FRACTAL ADL. Comme les autres ADL étudiés au chapitre 3, FRACTAL ADL permet de fournir une vue structurée des applications à composants. Pour ce faire, il permet de décrire, dans une syntaxe XML, l'ensemble des composants d'une application (partie fonctionnelle et contrôleurs), ainsi que leurs relations d'encapsulations et leurs liaisons avec les autres composants. Une usine utilise la description de l'application pour la déployer (éventuellement de façon distribuée).

Dans le chapitre 8, nous proposons trois extensions à FRACTAL ADL permettant respectivement des *reconfigurations structurelles*, des *reconfigurations d'implantation* et des *vérifications de type*.

Concernant les **reconfigurations structurelles**, nous avons développé un module qui permet d'"appliquer" un ADL à l'exécution. Dans cet ADL, sont décrits (1) des composants patrimoniaux qui ont déjà été déployés et (2) des composants qu'il faut déployer et lier aux composants patrimoniaux. Concernant les **reconfigurations d'implantation**, nous avons modifié la structure d'exécution de FRACTAL et son ADL de manière à autoriser des organisations arbitraires de class loaders pour le chargement du code des composants. Ce mécanisme permet d'effectuer des reconfigurations dynamiques des implantations de composants. Enfin, concernant les **vérifications de types**, nous avons développé un système de types pour les composants DREAM

qui a pour rôle d'effectuer certaines vérifications sur les assemblages de composants réalisés. Ce système de type permet, par exemple, de vérifier que les composants reçoivent des messages possédant les chunks appropriés. Notons cependant que ce système de types a quelques limitations qui excluent son utilisation pour certaines configurations de composants.

5.3 Organisation de la seconde partie

La seconde partie de ce document s'organise de la façon suivante : le chapitre 6 décrit le modèle de composants FRACTAL et son langage de description d'architectures. La bibliothèque de composants DREAM est présentée au chapitre 7. Le chapitre 8 décrit les extensions faites à FRACTAL ADL pour la gestion de configuration des architectures construites à l'aide de la bibliothèque DREAM. Enfin, nous concluons cette partie par une évaluation de DREAM dans le chapitre 9. Cette évaluation présente deux personnalités construites à l'aide de DREAM.

Chapitre 6

Le modèle de composants Fractal

Sommaire

Ce chapitre décrit FRACTAL, un modèle de composants généraliste dédié à la construction de systèmes. Nous commençons par une description du modèle. Nous présentons ensuite JULIA, l'implantation Java de référence du modèle. Enfin, nous décrivons FRACTAL ADL, un langage de description d'architectures extensible qui permet de déployer des applications FRACTAL à partir de leur description dans une syntaxe XML.

6.1 Le modèle de composants

Le modèle de composants FRACTAL [BCL+04] est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation. Cela motive les principales caractéristiques du modèle :

- **composants composites** (i.e. composants qui contiennent des sous-composants) pour permettre d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** (i.e. sous-composants de plusieurs composites englobants) pour permettre de modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- **capacités d'introspection** pour permettre d'observer l'exécution d'un système.
- **capacités de (re)configuration** pour permettre de déployer et de configurer dynamiquement un système.

Par ailleurs, FRACTAL est un modèle *extensible* du fait qu'il permet au développeur de personnaliser les capacités de contrôle de chacun des composants de l'application. Il est ainsi possible

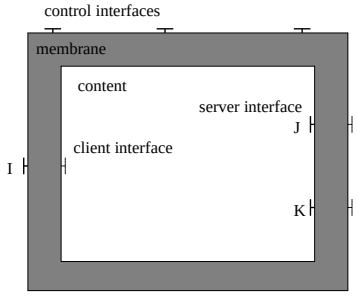

FIG. 6.1 – Exemple de composant FRACTAL.

d'obtenir un continuum dans les capacités réflexives d'un composant allant de l'absence totale de contrôle (boîte noire, objet standard) à des capacités élaborées d'introspection et d'intercession (e.g. accès et manipulation du contenu d'un composant, contrôle de son cycle de vie).

6.1.1 Composants et liaisons

Un composant FRACTAL est une entité d'exécution qui possède une ou plusieurs interfaces. Une *interface* est un point d'accès au composant. Une interface implante un *type d'interface* qui spécifie les opérations supportées par l'interface. Il existe deux catégories d'interfaces : les interfaces *serveurs* — qui sont des points d'accès acceptant des appels de méthodes —, et les interfaces *clients* qui sont des points d'accès émettant des appels de méthodes.

Comme il est représenté sur la figure 6.1, un composant FRACTAL est généralement composé de deux parties : une *membrane* — qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration (dynamique) du composant —, et un *contenu* qui est constitué d'un ensemble fini de *sous-composants*.

Les interfaces d'une membrane sont soit *externes*, soit *internes*. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes sont accessibles par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de *contrôleurs*. Les contrôleurs peuvent être considérés comme des méta-objets. Chaque contrôleur a un rôle particulier : par exemple, certains contrôleurs sont chargés de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). D'autres contrôleurs permettent de contrôler le comportement d'un composant et/ou de ses sous-composants. Un contrôleur peut, par exemple, permettre de suspendre/reprendre l'exécution d'un composant. Les contrôleurs peuvent également jouer le rôle d'intercepteurs. Les *intercepteurs* permettent d'intercepter les appels de méthodes entrant et sortant des interfaces d'un composant. Un exemple classique d'interception est l'ajout de pré- et post-traitements à l'appel.

Le modèle FRACTAL fournit deux mécanismes permettant de définir l'architecture d'une application : l'*imbrication* (à l'aide des composants composites) et la *liaison*. La liaison est ce qui permet aux composants FRACTAL de communiquer. FRACTAL définit deux types de liaisons : primitive et composite. Les liaisons *primitives* sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Une liaison primitive est implantée à l'aide d'un pointeur (en C) ou d'une référence (en Java). Les liaisons *composites* sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (e.g. stub, skeleton) reliés par des liaisons primitives.

Une caractéristique originale du modèle FRACTAL est qu'il permet de construire des compo-

sants *partagés*. Un composant partagé est un composant qui est inclus dans plusieurs composites. De façon paradoxale, les composants partagés sont utiles pour préserver l'encapsulation. En effet, il n'est pas nécessaire à un composant de bas niveau d'exporter une interface au niveau du composite qui l'encapsule pour accéder à une interface d'un composant partagé. De fait, les composants partagés sont particulièrement adaptés à la modélisation des ressources.

6.1.2 Niveaux de contrôle

Le modèle de composants FRACTAL n'impose la présence d'aucun contrôleur dans la membrane d'un composant. Il permet, au contraire, de créer des formes arbitrairement complexes de membranes implantant diverses formes et sémantiques de contrôle et d'interception. La spécification FRACTAL [BCS03] définit un certain nombre de niveaux de contrôle. Au niveau de contrôle le plus bas, un composant FRACTAL est une boîte noire qui ne permet ni introspection ni intercession. Les composants ainsi construits sont comparables aux objets instanciés dans les langages à objets comme Java. L'intérêt de ces composants réside dans le fait qu'ils permettent d'intégrer facilement des logiciels patrimoniaux.

Au niveau de contrôle suivant, un composant FRACTAL fournit une interface `Component`, similaire à l'interface `IUnknown` du modèle COM [Rog97]. Cette interface donne accès aux interfaces externes (clients ou serveurs) du composant. Chaque interface a un nom qui permet de la distinguer des autres interfaces du composant. A ce niveau de contrôle, les composants n'offrent toujours pas de fonctions de contrôle.

Au niveau de contrôle supérieur, un composant FRACTAL possède des interfaces réifiant sa structure interne et permettant de contrôler son exécution. La spécification FRACTAL définit différents contrôleurs :

- le **contrôleur d'attributs** pour configurer les attributs d'un composant.
- le **contrôleur de liaisons** pour créer/rompre une liaison primitive entre deux interfaces de composants.
- le **contrôleur de contenu** pour ajouter/retrancher des sous-composants au contenu d'un composant composite.
- le **contrôleur de cycle de vie** pour contrôler les principales phases comportementales d'un composant. Par exemple, les méthodes de base fournies par un tel contrôleur permettent de démarrer et stopper l'exécution du composant.

6.1.3 Système de types

Le modèle FRACTAL définit un système de types optionnel (les composants du niveau de contrôle le plus bas n'adhèrent pas nécessairement au système de types). Ce système de types autorise la description des opérations supportées par les différentes interfaces d'un composant. Il permet également de préciser le rôle de chacune des interfaces (i.e. client ou serveur), ainsi que sa cardinalité et sa contingence. La contingence d'une interface indique s'il est garanti que les opérations fournies ou requises d'une interface seront présentes ou non à l'exécution :

- les opérations d'une interface *obligatoire* sont toujours présentes. Pour une interface client, cela signifie que l'interface doit être liée pour que le composant s'exécute.
- les opérations d'une interface *optionnelle* ne sont pas nécessairement disponibles. Pour un composant serveur, cela peut signifier que l'interface interne complémentaire n'est pas liée à un sous-composant implantant l'interface. Pour un composant client, cela signifie que le composant peut s'exécuter sans que son interface soit liée.

La cardinalité d'une interface de type T spécifie le nombre d'interfaces de type T que le composant peut avoir. Une cardinalité *singleton* signifie que le composant doit avoir une, et

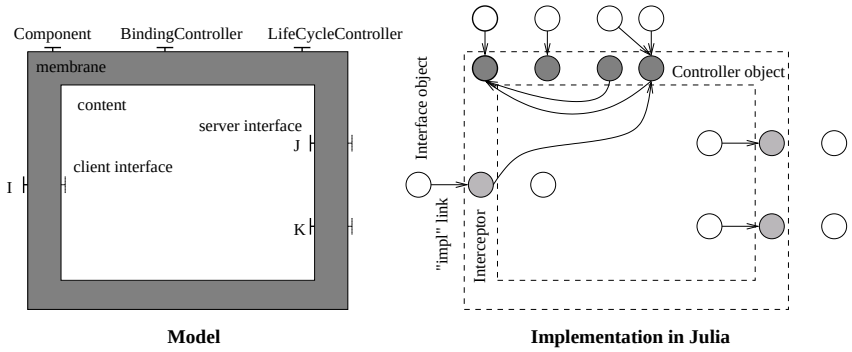

FIG. 6.2 – Un composant FRACTAL et son implantation JULIA.

seulement une, interface de type T. Une cardinalité *collection* signifie que le composant peut avoir un nombre arbitraire d'interfaces du type T. Ces interfaces sont généralement créées de façon paresseuse par le contrôleur de liaisons.

Le système de types permet également de décrire le type d'un composant comme l'ensemble des types de ses interfaces. Notons que le système de types définit une relation de sous-typage qui permet de vérifier des contraintes sur la substituabilité des composants.

6.2 JULIA : une implantation Java du modèle

Il existe différentes implantations du modèle FRACTAL : Think [FSLM02] en C, PLASMA [LH05] en C++, ProActive [BCM03] en Java, etc. Dans cette section, nous décrivons JULIA [jul02], l'implantation de référence du modèle. JULIA est un canevas logiciel écrit en Java qui permet de programmer les membranes des composants et de les déployer programmatiquement ou à l'aide d'un langage de description d'architectures[1].

JULIA fournit un ensemble de contrôleurs que l'utilisateur peut assembler pour créer les intercepteurs et contrôleurs de son choix. Par ailleurs, JULIA fournit des mécanismes d'optimisation qui permettent d'obtenir un continuum allant de configurations entièrement statiques et très efficaces à des configurations dynamiquement reconfigurables et moins performantes. Le développeur d'application peut ainsi choisir l'équilibre performance/dynamicité qu'il requiert. Enfin, il est important de noter que JULIA s'exécute sur toute JVM, y compris les plus contraintes comme la KVM qui ne fournissent pas de class loaders, ni d'API de réflexivité.

6.2.1 Principales structures de données

Un composant FRACTAL est formé de plusieurs objets Java que l'on peut séparer en trois groupes (figure 6.2) :

- Les objets qui implémentent le contenu du composant. Ces objets n'ont pas été représentés sur la figure. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
- Les objets qui implémentent la partie de contrôle du composant (représentés en gris). Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle et des intercepteurs optionnels qui interceptent les appels de méthodes entrants

[1] Le langage de description d'architectures est décrit dans la section 6.3.

et sortants. Les aspects de contrôle n'étant généralement pas indépendants, les contrôleurs et les intercepteurs possèdent généralement des références les uns vers les autres.
- Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.

La mise en place de ces différents objets est effectuée par des fabriques de composants. Celles-ci fournissent une méthode de création qui prend en paramètres la description des parties fonctionnelle et de contrôle du composant.

6.2.2 Contrôleurs

Motivations

FRACTAL étant un modèle de composants extensible, il est nécessaire de pouvoir construire facilement diverses formes de contrôleurs et diverses sémantiques de contrôle pour une même interface de contrôle. Par exemple, si l'on considère l'interface de contrôle de liaisons (`BindingController`), il est nécessaire de fournir différentes implantations de cette interface ; ces implantations diffèrent par les vérifications qu'elles font lors de la création/destruction d'une liaison : interaction avec le contrôleur de cycle de vie pour vérifier qu'un composant est stoppé, vérification que les types d'interface sont compatibles quand un système de types est utilisé, vérification que les composants liés sont parents d'un même composite quand les contrôleurs de contenu sont utilisés, etc.

Différentes solutions sont envisageables pour fournir ces différentes implantations :
- utilisation de l'*héritage de classe* : cette solution n'est pas envisageable car elle conduit à une explosion combinatoire du nombre de classes nécessaires. Supposons que l'on souhaite effectuer des vérifications concernant le système de types, le cycle de vie et le contrôleur de contenu. Il existe $2^3 = 8$ combinaisons possibles de ces différentes vérifications. De fait, pour implanter toutes les combinaisons possibles, il est nécessaire de fournir huit classes, ce qui engendre beaucoup de duplications de code.
- utilisation de la *programmation par aspect* (AOP pour *Aspect Oriented Programming*) [KLM+97] : cette solution résout le problème de l'explosion combinatoire sus-cité ; néanmoins, elle présente deux inconvénients : elle ne peut pas s'appliquer dynamiquement lors de l'exécution des composants et elle requiert le code source des classes[2].
- utilisation de *classes mixin* : une classe mixin est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. La classe mixin peut s'appliquer (c'est à dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe. L'outil ASM [ASM02] permet d'appliquer des classes mixin au chargement des classes, ce qui résout les limitations de la programmation par aspects.

Les classes mixin

Les classes mixin dans JULIA sont des classes abstraites développées avec certaines conventions. En l'occurence, elles ne nécessitent pas l'utilisation d'un compilateur Java modifié ou d'un pré-processeur comme c'est le cas des classes mixins développées à l'aide d'extensions du langage Java. Par exemple la classe mixin JAM [ALZ00] illustrée sur la partie gauche de la figure 6.3 s'écrit en JULIA en pur Java (partie droite).

Le mot clé `inherited` en JAM est équivalent au préfixe `_super_` utilisé dans JULIA. Il permet de spécifier les membres qui doivent être présents dans la classe de base pour que le mixin lui soit

[2]Cette limitation n'existe plus à partir d'AspectJ 1.1 qui n'était pas disponible lors du développement de Julia.

```
mixin A {
   inherited public void m ();
   public int count;
   public void m () {
      ++count;
      super.m();
   }
}
```

```
abstract class A {
   abstract void _super_m ();
   public int count;
   public void m () {
      ++count;
      _super_m();
   }
}
```

FIG. 6.3 – Ecriture d'une classe mixin en JAM et en JULIA.

appliqué. De façon plus précise, le préfixe `_super_` spécifie les méthodes qui sont surchargées par le mixin. Les méthodes qui sont requises mais pas surchargées sont spécifiées à l'aide du préfixe `_this_`.

L'application de la classe mixin `A` à la classe `Base` décrite sur la partie gauche de la figure 6.4 donne la classe `C55d992cb_0` représentée sur la partie droite de la figure 6.4.

```
abstract class Base {
   public void m () {
      System.out.println("m");
    }
}
```

```
public class C55d992cb_0 implements
                                Generated {
   // from Base
   private void m$0 () {
      System.out.println("m");
   }
   // from A
   public int count;
   public void m () {
      ++count;
      m$0();
   }
}
```

FIG. 6.4 – Application d'une classe mixin.

6.2.3 Intercepteurs

JULIA donne la possibilité de développer des *intercepteurs* dont le rôle est d'intercepter les appels de méthode entrant et/ou sortant des interfaces d'un composant. Les intercepteurs doivent implémenter les interfaces interceptées. Il est inconcevable de développer, pour un aspect de contrôle donné, autant d'intercepteurs que ce qu'il y a d'interfaces à intercepter dans l'application. En conséquence, JULIA fournit un outil, appelé *générateur d'intercepteurs*, qui permet de générer (dynamiquement) ces intercepteurs.

Le générateur d'intercepteurs prend en paramètres le nom d'une super-classe, le nom d'une ou plusieurs interfaces applicatives et un ou plusieurs *tisseurs d'aspects*. Les tisseurs d'aspects décrivent le code des intercepteurs générés et leurs interactions avec le code des composants interceptés. Le générateur d'aspect génère une sous-classe de la super-classe qui implante toutes les interfaces fonctionnelles spécifiées et qui, pour chaque méthode interceptée, implante les aspects décrits par les tisseurs d'aspects.

Les tisseurs d'aspects peuvent effectuer des modifications arbitraires des méthodes interceptées. La figure 6.5 représente un exemple de transformation d'une méthode `m()` dont le code est le suivant : `void m () { return delegate.m() }`

```
void m () {
   // pre code A
   try {
      delegate.m();
   } finally {
      // post code A
   }
}
```

FIG. 6.5 – Exemple d'intercepteur.

Notons que JULIA permet de traiter simultanément plusieurs tisseurs d'aspects, ce qui évite d'avoir une multiplication du nombre d'objets intercepteurs nécessaires.

6.2.4 Optimisations

JULIA offre deux mécanismes d'optimisation, intra et inter composants. Le premier mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant une partie de ses objets de contrôle. Pour ce faire, JULIA fournit un outil utilisant ASM [ASM02] et imposant certaines contraintes sur les objets de contrôle fusionnés : par exemple, deux objets fusionnés ne peuvent pas implémenter la même interface.

Le second mécanisme d'optimisation a pour fonction d'optimiser les chaînes de liaison entre composants : il permet de court-circuiter les parties contrôle des composites qui n'ont pas d'intercepteurs. Comme nous l'avons expliqué au paragraphe 6.2.1, chaque interface serveur de composant est représentée par un objet qui contient une référence vers un objet implantant réellement l'interface. Le principe du mécanisme de court-circuitage est représenté sur la figure 6.6 : un composant primitif est relié à un composite exportant l'interface d'un composant primitif qu'il encapsule. En conséquence, il existe deux objets de référencement d'interface (r_1 et r_2). Seuls les appels du primitif sont interceptés (objet i_1). En conséquence, JULIA court-circuite l'objet r_2 : r_1 référence directement i_1.

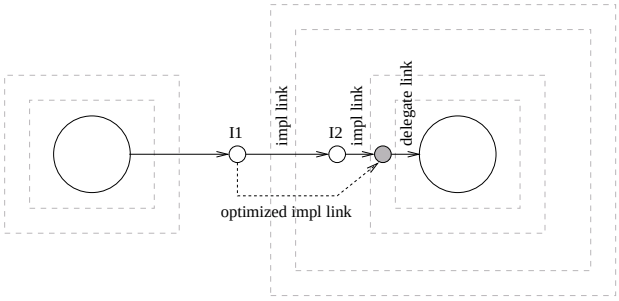

FIG. 6.6 – Optimisation des chaînes de liaison.

6.2.5 Performances

Cette section présente des mesures de performances qui ont été réalisées pour évaluer la surcharge en mémoire et en temps d'exécution induite par l'utilisation de JULIA. Les mesures effectuées comparent la taille d'un composant JULIA à celle d'un objet Java et le temps nécessaire à l'exécution d'une méthode vide sur l'objet et sur le composant. Le tableau 6.1 donne les résultats

pour plusieurs configurations de JULIA. Les mesures ont été effectuées sur un pentium III 1GHz avec une machine virtuelle Java 1.3 (HotSpotVM) s'exécutant sur Linux. La taille d'un objet est 8 octets et la durée d'un appel de méthode vide sur un objet est de 0.014 μs.

	surcharge en mémoire (octets)	surcharge en temps d'exécution (μs)
cycle de vie, sans optimisation	592	0.110
cycle de vie, fusion des contrôleurs	528	0.110
cycle de vie, fusion totale	504	0.092
pas de cycle de vie, pas d'optimisation	496	0.011
pas de cycle de vie, fusion des contrôleurs	440	0.011
pas de cycle de vie, fusion totale	432	0.011

TAB. 6.1 – Surcharge en mémoire et en temps d'exécution induite par l'utilisation de JULIA.

Comme on peut le constater, la fusion des contrôleurs permet de réduire l'empreinte mémoire du composant. Par ailleurs, le tableau montre l'impact de l'utilisation des intercepteurs. En effet, le contrôleur de cycle de vie utilise un intercepteur permettant de compter le nombre d'appels en cours dans le composant. Cet intercepteur utilise un block synchronisé (`synchronized`). La surcharge induite par cet intercepteur est de l'ordre de dix appels de méthode vide ($0.011\mu s$). En revanche, l'utilisation de JULIA sans intercepteurs n'engendre qu'une surcharge comparable à un appel de méthode vide ($0.011\mu s$).

6.3 Fractal ADL : un langage de description d'architectures extensible

FRACTAL fournit un langage de description d'architectures extensible. La motivation d'un ADL extensible est double. D'une part, le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (`LoggerController`) soit ajouté à un composant. Il est nécessaire que l'ADL puisse être étendu facilement pour prendre en compte ce nouveau contrôleur, c'est-à-dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (debug, warning, error, etc.). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL. En effet, nous avons vu dans le chapitre 3 que les ADL étaient aussi bien utilisés pour déployer les architectures décrites que pour effectuer des vérifications sur celles-ci. Comme nous allons le montrer dans le chapitre 8, Fractal ADL permet de construire des outils pour ces différentes utilisations.

Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. Nous présentons ces deux éléments dans les deux sections suivantes. La troisième section décrit le procédé d'extension de l'ADL.

6.3.1 Le langage extensible

Le langage ADL de FRACTAL est basé sur le langage XML. Contrairement aux autres ADL qui fixent l'ensemble des propriétés (implantation, liaisons, attributs, localisation, etc.) qui doivent être décrites pour chaque composant, l'ADL FRACTAL n'impose rien. Il est constitué d'un ensemble (extensible) de modules permettant la description de divers aspects de l'application.

Chaque module — à l'exception du module de base — s'applique à un ou plusieurs autres modules, c'est-à-dire rajoute un ensemble d'éléments et d'attributs XML à ces modules. Le module de base définit l'élément XML qui doit être utilisé pour démarrer la description de tout composant. Cet élément, appelé `definition`, a un attribut obligatoire, appelé `name`, qui spécifie le nom du composant décrit.

Différents types de modules peuvent être définis. Un exemple typique de module est le module *containment* qui s'applique au module de base en permettant d'exprimer des relations de contenance entre composants. Ce module définit un élément XML `component` qui peut être ajouté en sous-élément d'un élément `definition` ou de lui-même pour spécifier les sous-composants d'un composant. Notons que l'élément `component` a un attribut obligatoire `name` qui permet de spécifier le nom du sous-composant. Fractal ADL définit actuellement trois autres modules qui s'appliquent soit au module de base, soit au module `containment` pour spécifier l'architecture de l'application : le module `interface` permet de décrire les interfaces d'un composant ; le module `implementation` permet de décrire l'implantation des composant primitifs (i.e. une classe Java) ; le module `controller` permet la description de la partie contrôle des composants.

Les modules ne servent pas uniquement à décrire les aspects architecturaux de l'application. Par exemple, Fractal ADL fournit des modules permettant d'exprimer des relations de référencement et d'héritage entre descriptions ADL. Le rôle principal de ces modules est de faciliter l'écriture de définition ADL[3]. Le module de référencement s'applique au module `containment` en ajoutant un attribut `definition` à l'élément `component` pour référencer une définition de composant. Le module d'héritage s'applique au module de base. Il permet à une définition d'en étendre une autre (via un attribut `extends`). Notons que l'héritage proposé par Fractal ADL est multiple.

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="Client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="org.objectweb.julia.example.Client"/>
    <controller desc="primitive"/>
  </component>
  <component name="Server" definition="Server"/>
  <binding client="this.r" server="Client.r"/>
  <binding client="Client.s" server="Server.s"/>
  <controller desc="composite"/>
</definition>
```

Fig. 6.7 – Un exemple de définition ADL à l'aide du langage extensible Fractal ADL.

La figure 6.7 donne un exemple de définition réalisée à l'aide de Fractal ADL. Le composant décrit est un composite dont le nom est `HelloWorld`. Ce composite possède une interface serveur, de nom `r` et de signature `java.lang.Runnable`. Par ailleurs, le composite encapsule deux composants : `Client` et `Server`. La définition du composant `Client` est intégrée à celle du composant `HelloWorld` : le composant a deux interfaces (`r` et `s`), sa classe d'implantation est `org.objectweb.julia.example.Client` et il possède une partie de contrôle de type `primitive`[4].

[3] Notons cependant que le module de référencement est nécessaire pour la description des composants partagés.

[4] Les deux mots utilisés pour décrire les parties contrôle des composants (`primitive` et `composite`) sont définis dans un fichier de configuration de Julia qui permet de spécifier l'ensemble des contrôleurs des différents composants.

La définition du composant `Server` n'est pas intégrée à celle du composant `HelloWorld` : le module de référencement est utilisé pour spécifier que la description se trouve dans un fichier nommé `Server.fractal`. Enfin, la description ADL mentionne deux liaisons : entre les interfaces `r` du composite et du `Client` et entre les interfaces `s` du `Client` et du `Server`.

6.3.2 L'usine extensible

L'usine permet de traiter les définitions écrites à l'aide du langage extensible Fractal ADL. Nous commençons par en donner une description générale, avant de présenter plus en détail ses différents constituants.

Description générale

L'usine est constituée d'un ensemble de composants Fractal qui peuvent être assemblés pour traiter les différents modules de Fractal ADL décrits précédemment. Ces composants sont représentés sur la figure 6.8.

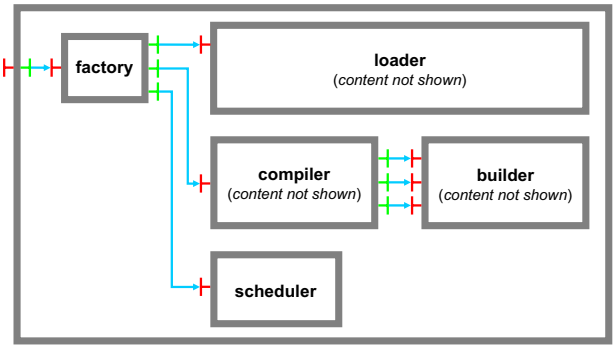

Fig. 6.8 – Architecture de l'usine Fractal ADL.

- le composant *loader* analyse les définitions ADL et contruit un arbre abstrait correspondant (AST pour *Abstract Syntax Tree*). L'AST implante deux API distinctes : une API *générique* similaire à celle de DOM [con] qui permet de naviguer dans l'arbre ; une API *typée* qui varie suivant les modules qui sont utilisés dans Fractal ADL. Par exemple, si le module `interface` est utilisé, l'API typée contient des méthodes permettant de recupérer les informations sur les interfaces de composants (nom, signature, rôle, etc.).
- le composant *compiler* a pour rôle de générer un ensemble de tâches à exécuter à partir d'un AST. Des exemples typiques de tâches sont la création d'un composant, l'établissement d'une liaison, etc. Notons que le *compiler* définit également les dépendances entre les tâches qu'il crée.
- le composant *builder* définit un comportement concret pour les tâches créées par le *compiler*. Par exemple, un comportement concret d'une tâche de création de composant peut être d'instancier le composant à partir d'une classe Java.

Le composant *loader*

Le composant *loader* est un composite encapsulant une chaîne de composants primitifs (figure 6.9). Le composant le plus à droite dans la chaîne (*basic loader*) est responsable de la création des AST à partir de définitions ADL. Les autres composants effectuent des vérifications et des transformations sur les AST. Chaque composant correspond à un module de l'ADL. Par exemple,

le composant *binding loader* vérifie les liaisons déclarées dans l'AST ; le composant *attribute loader* vérifie les attributs, etc. Notons qu'un composite *loader* encapsulant des sous-composants en charge des modules A, B et C sera capable de charger des définitions utilisant les modules B, C et D. Cependant, aucune vérification ne sera faite concernant le module D et le composant en charge du module A sera inutile.

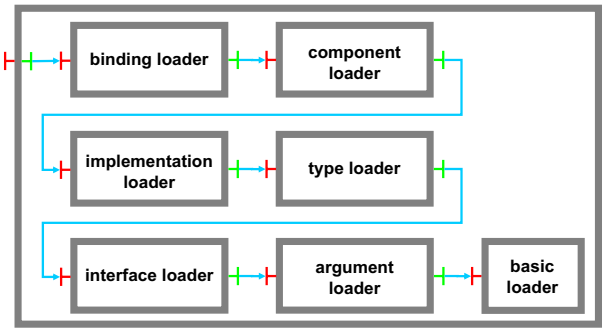

FIG. 6.9 – Architecture du composant *loader*.

Les composants *compiler* et *builder*

Le composant *compiler* est un composite encapsulant un ensemble de composants *compiler* primitifs (figure 6.10). Chaque *compiler* primitif produit des tâches correspondant à un ou plusieurs modules ADL. Par exemple, le composant *binding compiler* produit des tâches de création de liaisons. Les tâches produites par un *compiler* primitif A peuvent dépendre des tâches produites par un *compiler* primitif B. Dans ce cas, il est nécessaire que le *compiler* B soit exécuté avant le compiler *A*. Par exemple, le *compiler* primitif qui produit les tâches de création des composants doit être exécuté avant les *compilers* en charge des liaisons et des attributs. Chaque *compiler* est associé à un composant *builder* qui définit un comportement pour les tâches générées par le *compiler*. Un *builder* est un composant composite qui encapsule plusieurs *builders* primitifs chargés des différentes tâches créées par les *compilers* primitifs. JULIA fournit actuellement quatre composants *builder* qui permettent respectivement de créer des composants avec l'API Java, l'API Fractal ou de produire du code source permettant d'instancier les composants à partir de ces deux API.

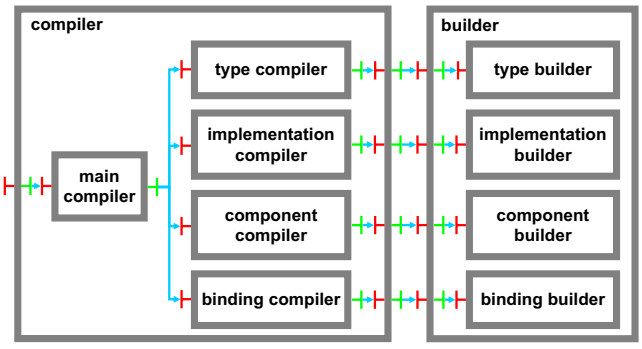

FIG. 6.10 – Architecture des composants *compiler* et *builder*.

6.3.3 Extension de Fractal ADL

Cette section décrit brièvement la démarche à suivre pour étendre Fractal ADL. Supposons que le développeur de l'application ait défini une interface de contrôle `LoggerController` permettant de configurer le nom et le niveau de journalisation de *loggers* associés aux composants (voir figure 6.11). Nous montrons tout d'abord comment le langage ADL peut être étendu pour autoriser la description de ces deux paramètres dans la description ADL de l'application. Nous décrivons ensuite les composants qui doivent être rajoutés à l'usine pour que les loggers soient configurés lors du déploiement de l'application.

```
public interface LoggerController {
   String getLoggerName ();
   void setLoggerName (String logger);
   int getLogLevel ();
   void setLogLevel (int level);
}
```

Fig. 6.11 – L'interface `LoggerController`.

Extension du langage

L'extension du langage consiste à créer un module *logger* qui s'applique aux modules de base `containment` en permettant de rajouter un élément `logger` dont la syntaxe est la suivante :

```
<logger name="logger" level="DEBUG"/>
```

Le développeur du module doit effectuer le travail suivant :
- définition des interfaces qui seront implantées par l'arbre abstrait lorsque le fichier XML aura été analysé par le *basic loader*.
- écriture de "fragments" de DTD spécifiant que le module `logger` permet d'ajouter un élément XML `logger` aux éléments `definition` et `component`.

Extension de l'usine

L'extension de l'usine requiert l'ajout de deux composants : un *compiler* et un *builder*. En effet, le module `logger` ne nécessitant pas de vérifications sémantiques, il n'est pas utile de modifier le composant *loader*. Notons, néanmoins, qu'il est possible de développer un loader vérifiant que les noms et niveaux de journalisation déclarés dans la description ADL ne sont pas nuls.

Le composant *compiler* doit créer des tâches qui configureront le nom et le niveau de journalisation des loggers déclarés dans la description ADL. L'implantation du *compiler* est simple : il crée une tâche pour chaque logger à configurer et ajoute une dépendance de cette tâche vers la tâche de création du composant auquel le logger appartient.

Concernant le composant *builder*, il est nécessaire de réaliser plusieurs implantations permettant de prendre en compte les deux implantations de composants possibles : objets Java standards ou objets Java implantant l'API Fractal. Dans les deux cas, l'implantation du builder est très simple : elle consiste uniquement en la récupération du logger associé au composant et en son initialisation à l'aide des informations contenues dans l'arbre abstrait.

6.4 Conclusion

Dans ce chapitre, nous avons présenté le modèle de composants FRACTAL et son langage de description d'architectures. FRACTAL est un modèle généraliste qui a comme principales caractéristiques la possibilité de créer des architectures hiérarchiques avec partage de composants et la possibilité d'associer à chaque composant un méta-niveau arbitrairement complexe.

Son langage de description d'architectures est extensible, ce qui signifie qu'il est possible d'étendre à la fois la syntaxe XML utilisée pour décrire les architectures et l'usine de traitement de ces descriptions. Nous donnons des exemples de telles extensions dans le chapitre 8.

Chapitre 7

La bibliothèque de composants

Sommaire

Ce chapitre présente la bibliothèque de composants DREAM. Nous décrivons tout d'abord les composants permettant de gérer les deux ressources présentes dans les intergiciels de communication : *activités* et *messages*. Nous présentons ensuite des abstractions et outils fournis par la bibliothèque pour organiser les composants sous forme de protocoles et pour établir des laisons distribuées entre composants via ces protocoles. Enfin, nous décrivons les composants de la bibliothèque. Ces composants encapsulent les fonctions que l'on trouve de façon commune dans les différents intergiciels de communication[1]. Leur spécificité est de manipuler des messages.

7.1 Canevas pour la gestion des activités

Nous avons développé un canevas constitué d'un ensemble de contrôleurs et de composants permettant de gérer des activités au sein de composants FRACTAL. Le but de ce canevas est double : (1) faciliter le travail du développeur d'intergiciels en ôtant le code de gestion des threads Java du code des composants et (2) permettre des organisations flexibles des flots d'exécution au sein d'une application à composants. Ce dernier point consiste à découpler la notion d'*activité* — qui correspond à une tâche que le composant veut exécuter —, de la notion de thread — qui correspond aux flots d'exécution au sein de la machine virtuelle Java. Il est particulièrement

[1] La librairie possède également des composants spécifiques développés pour des personnalités particulières d'intergiciels de communication. Des exemples de tels composants sont présentés dans le chapitre 9.

intéressant de découpler le code des composants de l'organisation des flots d'exécution car cela permet :

- de pouvoir adapter un intergiciel à des environnements d'exécution aux ressources variées. Par exemple, il est possible, en utilisant des ordonnanceurs d'activités appropriés, de n'utiliser qu'un seul thread dans un intergiciel.
- d'optimiser les performances de l'intergiciel en adoptant une structure de flots d'exécution adaptée aux conditions d'exécution. Ainsi, nous présentons deux exemples dans le chapitre 9 d'intergiciels pour lesquels une modification de la structure des flots d'exécution a engendré un gain de performance non négligeable. Dans un cas, ce gain provient du fait que la structure des flots d'exécution est mieux adaptée à la structure des échanges de messages au sein de l'intergiciel : un seul message au plus transitant à tout instant dans l'intergiciel, un seul flot d'exécution est nécessaire. Dans l'autre cas, le gain provient du fait que l'intergiciel est testé en condition de faible charge (i.e. la fréquence des messages traités est faible), ce qui nécessite une organisation des flots d'exécution différente de celle par défaut — qui est adaptée à un fonctionnement en forte charge.

Plusieurs travaux se sont intéressés à la gestion des ressources dans les intergiciels [Gro01, Sel00, CJCP00, CSB99, SLR+03, KYH+01]. Un état de l'art de ces travaux est fait dans [DLBC04]. Le travail le plus proche de celui que nous présentons dans cette section est celui qui a été réalisé dans le cadre du projet OpenORB [BCA+01]. Celui-ci définit un canevas de gestion de ressources permettant d'organiser hiérarchiquement un ensemble de tâches. Les tâches sont associées à un pool de ressources qui supportent leur exécution. L'une des contributions de ce canevas de tâches est la définition d'un langage de configuration des ressources [DLB04] qui permet de spécifier la configuration et la reconfiguration des ressources dans le système. Ce canevas est plus général que celui que nous proposons du fait qu'il prend en compte plusieurs types de ressources (CPU, mémoire, réseau). En revanche, il n'est pas aussi flexible que celui que nous proposons et dans lequel tâches et ressources (i.e. threads) sont entièrement modélisées par des composants FRACTAL communiquant au travers de liaisons.

Cette section est organisée de la façon suivante : nous introduisons le principe de découplage des activités et des threads. Nous détaillons ensuite les composants du canevas. Enfin, nous décrivons son utilisation et en évaluons les performances.

7.1.1 Principe du découplage des activités et des threads

Un composant DREAM peut être *passif* ou *actif*. Un composant actif définit des **tâches** à exécuter. Le code d'une tâche peut faire intervenir, entre autres, des appels sur les interfaces clientes du composant. Au contraire, un composant passif ne définit aucune tâche ; les appels sur ses interfaces clientes sont effectués dans les tâches des composants invoquant ses interfaces serveur. Pour qu'une tâche soit exécutée, il est nécessaire de l'enregistrer auprès d'un composant appelé *gestionnaire d'activités*. Un gestionnaire d'activités contient des tâches et des ordonnanceurs. Les **ordonnanceurs** sont chargés d'associer des tâches de haut niveau à des tâches de bas niveau. Au plus haut niveau, les tâches sont des tâches applicatives (i.e. enregistrées par les composants). Au plus bas niveau, les tâches encapsulent des threads Java. Le canevas autorise des organisations arbitraires d'ordonnanceurs : les ordonnanceurs peuvent être organisés hiérarchiquement en intercalant des tâches intermédiaires, appelées *inter-scheduling tasks*.

Ces concepts sont représentés sur la figure 7.1. Les composants A et B ont enregistré trois tâches qui sont ordonnancées par deux ordonnanceurs organisés hiérarchiquement. Le résultat obtenu est le suivant : l'ordonnanceur `Periodic Scheduler` utilise deux tâches de plus bas niveau

(i.e. encapsulant des threads Java) pour assurer l'exécution de la tâche `B` à une période T_B[2] et l'exécution de la tâche `IS task` à une période T_{IS_task}. L'exécution de cette dernière tâche déclenche l'ordonnancement en séquence des tâches `A1` et `A2` par l'ordonnanceur `FIFO Scheduler`.

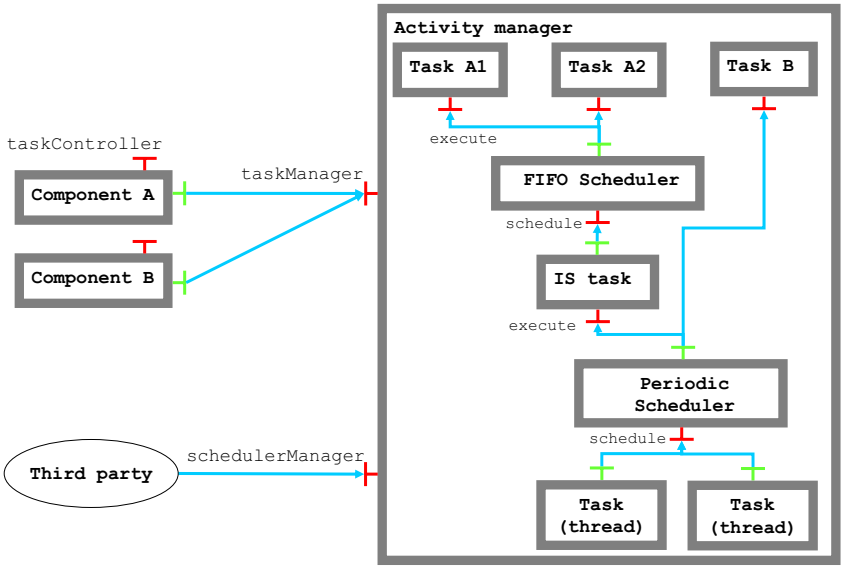

FIG. 7.1 – Exemple de gestionnaire d'activités.

7.1.2 Les composants du canevas

Les ordonnanceurs

Les ordonnanceurs sont des composants qui ont une interface serveur `Scheduler` et une interface client de collection `Task`. Le rôle d'un ordonnanceur est d'associer des tâches de haut niveau (auxquelles son interface client de collection `Task` est liée) à des tâches de bas niveau (qui sont liées à son interface serveur `Scheduler`).

L'interface `Scheduler` (figure 7.2) définit une méthode `schedule` qui est appelée par les tâches de bas niveau pour déclencher l'ordonnancement de l'exécution (d'un ensemble) des tâches de haut niveau. Cette méthode a un paramètre `params` qui permet de spécifier les paramètres d'ordonnancement (e.g. le nombre de tâches de haut niveau à exécuter). Elle retourne un objet représentant le résultat de l'ordonnancement (e.g. le nombre de tâches de haut niveau exécutées).

```
public interface Scheduler {
   Object schedule(Object params) throws InterruptedException,
                                         StoppedSchedulerException;
}
```

FIG. 7.2 – L'interface `Scheduler`.

L'interface `Task` (figure 7.3) définit une méthode `execute` qui est utilisée par l'ordonnanceur pour exécuter des tâches de haut niveau. Cette méthode a un paramètre `params` qui permet de

[2]La période est spécifiée à l'enregistrement de la tâche B.

spécifier les paramètres d'exécution de la tâche. Par ailleurs, la méthode `execute` retourne un objet représentant le résultat de l'exécution de la tâche.

```
public interface Task {
    Object execute(Object params) throws InterruptedException;
    void interrupted();
    void registered(Object controlItf);
    void unregistered();
}
```

FIG. 7.3 – L'interface `Task`.

Il existe divers types d'ordonnanceurs, parmi lesquels :
- l'**ordonnanceur de transfert** (*forwarder scheduler*) est le plus simple des ordonnanceurs : il est lié à une seule tâche de haut niveau qu'il exécute chaque fois que sa méthode `schedule` est invoquée.
- l'**ordonnanceur FIFO** (*FIFO scheduler*) exécute les tâches de haut niveau auxquelles son interface client de collection `Task` est liée dans l'ordre dans lequel les tâches ont été liées.
- l'**ordonnanceur périodique** (*periodic scheduler*) exécute les différentes tâches de haut niveau auxquelles son interface client de collection `Task` est liée ; les exécutions sont faites de façon périodique. La période peut être spécifiée à l'enregistrement de la tâche ou en retour de chacune de ses exécutions.
- l'**ordonnanceur aléatoire** (*random scheduler*) exécute les tâches de haut niveau auxquelles son interface client de collection `Task` est liée dans un ordre aléatoire.

Les tâches

Les tâches sont des composants avec une interface serveur `Task` (figure 7.3). Outre la méthode `execute` qui correspond au code de la tâche, cette interface définit trois méthodes de *callback* qui sont appelées lorsque la tâche est (des-)enregistrée ou interrompue. Nous distinguons :
- les **tâches de plus haut niveau** qui sont les tâches définies par les composants de l'application. Leur méthode `execute` contient du code fonctionnel et suit certaines conventions qui dépendent de l'ordonnanceur par lequel la tâche est exécutée. Par exemple, dans le cas d'un ordonnanceur de transfert, FIFO ou aléatoire, cette méthode doit retourner 0 pour indiquer que la tâche doit être de nouveau exécutée par l'ordonnanceur, et -1 le cas échéant. En revanche, si cette tâche est destinée à être exécutée par un ordonnanceur périodique, elle peut retourner les deux valeurs précédentes ou un nombre positif pour spécifier à l'ordonnanceur l'intervalle de temps (en ns) qui doit s'écouler avant la prochaine exécution de la tâche. Notons qu'il n'y a aucune limitation, a priori, sur le nombre de threads qui peuvent exécuter une tâche simultanément. Il est donc de la responsabilité du développeur de la tâche de synchroniser les accès à des espaces mémoire partagés[3].
- les **tâches de plus bas niveau** qui encapsulent des threads Java. Ces tâches ont une interface cliente `Schedule`. Leur méthode `run`[4] est une boucle qui appelle leur méthode

[3] Il est cependant possible de ne pas utiliser de code de synchronisation si le développeur utilise des ordonnanceurs garantissant qu'un seul thread exécutera la tâche à tout moment, ou s'il ajoute un intercepteur au composant représentant la tâche afin de limiter (à 1) le nombre des appels à l'interface `Task`.

[4] La méthode `run` est la méthode implantée par les threads Java. Elle est définie dans l'interface `java.lang.Runnable`.

`execute`. Le code de cette dernière consiste simplement en un appel de la méthode `schedule` de leur interface cliente.
- les **tâches intermédiaires** qui sont créées par les ordonnanceurs pour permettre l'ordonnancement de leur exécution par des ordonnanceurs de plus bas niveau. Ces tâches permettent d'organiser hiérarchiquement les ordonnanceurs.

7.1.3 Utilisation des activités

Pour qu'un composant soit actif, il est nécessaire qu'il possède un contrôleur implantant l'interface `TaskController` (figure 7.4). Ce contrôleur maintient une liste des tâches du composant auquel il appartient. Pour enregistrer une tâche, le développeur de composant crée la tâche (un objet Java implantant l'interface `Task`) et l'enregistre auprès de son contrôleur via la méthode `addTask`. L'enregistrement de la tâche auprès du gestionnaire d'activité et son retrait sont effectués de façon automatique par le contrôleur de cycle de vie lors du démarrage et de l'arrêt du composant. Notons que la hiérarchie d'ordonnanceurs est mise en place lors du démarrage du gestionnaire d'activités en parsant une description faite à l'aide de Fractal ADL. Les ordonnanceurs et les tâches étant des composants Fractal communiquant par l'intermédiaire de liaisons, il est possible de modifier dynamiquement la hiérarchie d'ordonnanceurs en utilisant l'API de contrôle de Julia.

```
public interface TaskController {
    void addTask(Task task, Map params) throws IllegalTaskException;
    void removeTask(Task task) throws NoSuchTaskException,
                                      IllegalTaskException;
    Object getTaskControl(Task task) throws NoSuchTaskException,
                                            IllegalTaskException;
    Task[] getTasks();
}
```

Fig. 7.4 – The `TaskController` interface.

7.1.4 Performances

Nous avons effectué des mesures de performances afin d'évaluer la surcharge en temps d'exécution induite par l'utilisation du canevas de gestion des activités.

La première expérience vise à mesurer le surcoût lié à la création d'une tâche. Nous comparons les temps d'exécution d'une application créant 5000 threads et d'une application créant et enregistrant 5000 tâches. Chaque tâche est ordonnancée par un ordonnanceur de transfert. Le tableau 7.1 présente les résultats obtenus sur un pentium IV 2GHz avec 1Go de mémoire, exécutant une JDK 1.5 sur linux 2.6.13. Nous constatons une surcharge en temps d'exécution de l'ordre de 10%. Ce surcoût est tout à fait raisonnable : il est, en effet, de l'ordre de 16μs par tâche créée, ce qui est tout à fait négligeable par rapport à la durée de vie moyenne d'une tâche.

La seconde expérience réalisée vise à mesurer le surcoût engendré par l'ordonnancement des tâches. Nous comparons les temps d'exécution :
- d'une application dans laquelle un thread exécute 5000000 fois une méthode synchronisée inversant l'état d'un booléen.
- d'une application qui exécute 5000000 fois une tâche qui exécute une méthode synchronisée inversant l'état d'un booléen. A chaque exécution de la tâche, celle-ci est ordonnancée par

Expérience	Temps moyen (s)
Création de 5000 threads	0.72
Création et enregistrement de 5000 tâches	0.80

TAB. 7.1 – Comparaison du temps de création d'un thread et du temps de création et d'enregistrement d'une tâche.

un ordonnanceur de transfert.

Le tableau 7.2 présente les résultats obtenus sur un pentium IV 2GHz avec 1Go de mémoire, exécutant une JDK 1.5 sur linux 2.6.13. Nous constatons une surcharge en temps d'exécution de l'ordre de 6%. Ce surcoût est tout à fait raisonnable, compte-tenu du fait que la méthode exécutée n'a quasiment aucun code. Comme le prouvent les expériences présentées dans le chapitre 9, ce surcoût disparaît totalement dans les cas réels. En effet, il est négligeable face à des opérations de sérialisation de messages, de transferts réseaux, etc.

Expérience	Temps moyen (s)
Thread exécutant 5000000 appels	1.68
5000000 exécutions d'une tâche qui exécute un appel	1.79

TAB. 7.2 – Comparaison du temps d'exécution d'une application exécutant un thread et d'une application exécutant une tâche.

7.2 Les messages DREAM et leur gestion

DREAM étant destiné à la construction d'intergiciels de communication, il n'est pas surprenant que la majeure partie des composants de la bibliothèque échangent des messages. Le choix effectué par la plupart des intergiciels de communication développés en Java est de laisser la gestion du cycle de vie des messages à la machine virtuelle Java. En d'autres termes, chaque fois que l'intergiciel nécessite un message, il crée un objet Java représentant le message ; lorsque le message n'est plus utilisé, il est collecté par le ramasse-miettes de la machine virtuelle. Nous n'avons pas retenu ce mode de fonctionnement dans DREAM, car il engendre des pertes de performances du fait de deux facteurs :

- la création d'objet est coûteuse. Ce coût tend à diminuer avec les versions récentes de la machine virtuelle Java ; il demeure néanmoins non négligeable.
- le ramasse-miettes consomme du temps CPU et son utilisation fréquente peut engendrer des effets pervers ; par exemple, on peut constater des pertes importantes de paquets UDP ou IP-multicast dans les périodes d'activité du ramasse-miettes. Ceci vient du fait que les tampons du système d'exploitation utilisés pour stocker les paquets UDP et IP-multicast ont une taille limitée. Si le tampon est plein et que des paquets arrivent, le système d'exploitation détruit des paquets.

Afin d'améliorer les performances, nous avons décidé de rendre explicite le cycle de vie des messages DREAM et de déléguer leur gestion à des composants spécifiques, appelés *gestionnaires de messages*. Dans la suite de cette section, nous présentons les abstractions fournies par le canevas DREAM pour gérer les messages.

7.2.1 Les interfaces d'entrée et de sortie de messages

Le canevas DREAM définit deux interfaces permettant aux composants d'échanger des messages. Ces deux interfaces, appelées `Push` et `Pull` sont représentées sur la figure 7.5.

```
public interface Push {
   void push(Message message) throws PushException;
}

public interface Pull {
   Message pull() throws PullException;
}
```

FIG. 7.5 – Les interfaces `Push` et `Pull`.

Les interfaces `Push` et `Pull` peuvent se comporter comme entrée (*input*) ou sortie (*output*) de messages selon qu'elles ont un rôle client ou serveur. Comme nous l'avons représenté sur la figure 7.6 (a), les messages sont toujours transmis des sorties vers les entrées. Dans le *mode push*, l'échange de message est initié par la sortie qui est une interface client `Push` liée au composant auquel le message est destiné (figure 7.6 (b)). Dans le *mode pull*, l'échange de message est initié par l'entrée qui est une interface client `Pull` liée au composant émetteur du message (figure 7.6 (c)).

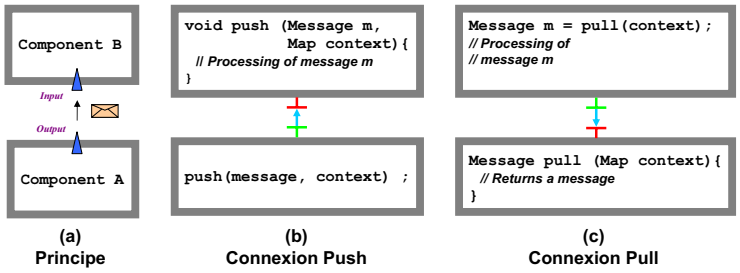

FIG. 7.6 – Les interfaces d'entrée et de sortie de messages.

7.2.2 Les messages DREAM

Contrairement aux intergiciels monolithiques implantant une API déterminée, DREAM ne possède pas un format de message fixe. En effet, les informations véhiculées par un message dépendent de l'intergiciel construit à l'aide de la bibliothèque de composants. Par exemple, si l'intergiciel utilise des sockets TCP pour le transport, le message devra contenir un numéro de port et une adresse IP. En revanche, si les échanges reposent sur IP-multicast, les messages devront posséder une adresse multicast. Par ailleurs, les intergiciels DREAM étant dynamiquement configurables, le format des messages peut être amené à changer en cours d'exécution. Supposons qu'il soit nécessaire d'imposer dynamiquement un ordonnancement causal des messages échangés, il faut alors modifier le format de ces messages afin qu'ils contiennent une horloge matricielle. En conséquence, les messages DREAM doivent avoir une structure flexible autorisant des contenus de messages arbitrairement complexes.

La structure que nous avons retenue est la suivante : les messages DREAM sont des objets Java encapsulant des *chunks* et des sous-messages. Chaque chunk est également un objet Java

implantant des accesseurs (*getter*) et des mutateurs (*setter*). Par exemple, un message qui doit être causalement ordonné doit posséder un chunk `CausalChunk` qui définit des méthodes pour consulter/modifier une horloge matricielle. Par ailleurs, chaque chunk doit implanter l'interface `Chunk` (figure 7.7) qui définit des méthodes pour créer une instance de chunk de la même classe et pour transférer l'état d'un chunk dans un autre chunk. Ces méthodes sont utilisées par les gestionnaires de messages pour permettre une duplication par valeur des messages.

```
public interface Chunk {
   Chunk createChunk ();
   void transferStateTo (Chunk newInstance);
}
```

FIG. 7.7 – L'interface `Chunk`.

Un message peut également encapsuler des sous-messages. Les messages composés de plusieurs messages sont appelés *messages agrégés*. Le but des messages agrégés est de pouvoir transmettre simultanément sur le réseau un ensemble de messages sémantiquement corrélés. Prenons, par exemple, le cas d'un intergiciel LEWYS[5] permettant d'envoyer des informations de supervision sur le CPU, la mémoire et les statistiques réseau. Il est intéressant de pouvoir envoyer un message agrégé contenant trois sous-messages possèdant chacun un chunk de supervision. Seul le message agrégé sera porteur des chunks nécessaires à l'émission du message (adresse du destinataire, ordonnancement, etc.).

Chaque message implémente l'interface `Message` (figure 7.8) qui donne accès aux chunks et sous-messages du message. Cette interface permet également d'ajouter/retrancher des chunks et sous-messages. Notons que les chunks et les sous-messages sont traités de façons différentes : les messages sont des contextes de noms pour les chunks qu'ils encapsulent. En revanche, les sous-messages ne sont pas associés à des noms. Ce choix résulte du fait qu'il est nécessaire de pouvoir construire des messages agrégés de façon efficace. Cela ne serait pas possible s'il fallait générer des noms différents pour chacun des sous-messages. Néanmoins, il est possible d'associer des noms aux messages en leur ajoutant, par exemple, un chunk `NameChunk` contenant un objet `String`.

7.2.3 Les gestionnaires de messages

Les messages sont gérés par des composants partagés, appelés gestionnaires de messages (*message managers*). Ces composants implémentent l'interface serveur `MessageManager` (figure 7.9) qui permet de créer, détruire et dupliquer des messages. Le but des gestionnaires de messages est double :

- améliorer les performances en utilisant des pools de messages afin de réduire le nombre d'allocations d'objets Java et le travail effectué par le ramasse-miettes Java.
- gérer les ressources mémoires consommées par les intergiciels de communication. Les gestionnaires de messages peuvent, par exemple, limiter le nombre de messages présents dans une architecture donnée. Cette capacité est notamment intéressante pour les intergiciels de communication destinés à être déployés sur des équipements ayant des ressouces mémoires restreintes.

[5] LEWYS est un canevas à composants dédié à la construction de systèmes de supervision. Il est présenté au chapitre 10.

```
public interface Message {
    // Chunk management
    Chunk getChunk(String name);
    Chunk addChunk(String name, Chunk chunk);
    Chunk removeChunk(String name);
    Iterator getChunkNamesIterator();

    // Submessage management
    Iterator getSubMessagesIterator();
    void addSubMessage(Message message);
    boolean removeSubMessage(Message message);
}
```

FIG. 7.8 – L'interface `Message`.

```
public interface MessageManager {
    Message createMessage();
    void deleteMessage(Message message);
    Message duplicateMessage(Message message, boolean clone);
}
```

FIG. 7.9 – L'interface `MessageManager`.

7.3 Outils pour l'établissement de liaisons

Le canevas DREAM définit des abstractions et fournit des outils pour établir des liaisons à l'aide des composants de la bibliothèque. Une liaison permet à un ensemble de composants d'interagir. La liaison est donc une fonction de base de tout intergiciel. Elle couvre une large gamme de situations, en raison de la variété des sémantiques de communication qui traduit différents besoins des applications, et de la variété des contextes dans lesquelles elle est mise en œuvre : communication point à point ou diffusion, flots multimédia, systèmes client/serveur, etc. Une telle variété nécessite l'usage de mécanismes génériques et modulaires pouvant être adaptés ou étendus pour répondre à des besoins changeants.

Le canevas d'établissement de liaisons est inspiré des travaux qui ont été faits dans l'intergiciel Jonathan [DHTS99]. En effet, celui-ci repose sur l'utilisation du patron de conception `export-bind` [KS05]. En revanche, contrairement à Jonathan, sa mise en œuvre est entièrement faite à l'aide de composants et fait intervenir la structure de partage fournie par FRACTAL, ce qui offre des possibilités d'administration dynamique des liaisons.

Cette section s'organise ainsi : nous décrivons tout d'abord le patron `export-bind`. Nous présentons ensuite sa mise en œuvre dans DREAM. Enfin, nous présentons l'exemple de l'établissement d'une liaison TCP/IP entre deux composants.

7.3.1 Le patron Export/Bind

Principe

Le patron `export-bind` fournit un mécanisme générique pour créer et gérer des liaisons entre composants dans un système réparti, en vue de permettre la communication entre ces composants,

avec les contraintes suivantes :

- Les composants existent dans des systèmes hétérogènes, avec différentes conventions de représentations et différentes sémantiques de communication.
- Le patron doit être applicable à plusieurs niveaux (système d'exploitation, réseau, application), et à des formes diverses d'organisation des composants (hiérarchique, graphe, etc.).
- La communication peut traverser plusieurs espaces d'adressage, sur plusieurs machines en réseau, et peut englober plusieurs niveaux de hiérarchie de mémoire.

Le principe de fonctionnement du patron `export-bind` est le suivant : un composant *serveur* souhaitant rendre accessible un service aux autres composants, exporte l'interface correspondante. Le résultat de cette opération d'export est la création d'un identifiant unique. Cet identifiant peut être utilisé par un composant *client* souhaitant accéder au service exporté par le serveur. Pour ce faire, le composant appelle la méthode `bind` en donnant en paramètre l'identifiant du service auquel il veut accéder. Le résultat de cette opération est la création d'une liaison entre le client et le serveur.

Exemples

Deux exemples classiques d'utilisation du patron `export-bind` sont :

- *l'invocation de méthode à distance :* l'opération `export` consiste à créer un skeleton donnant accès à la méthode exportée. L'opération `bind` retourne un stub qui peut être utilisé pour invoquer la méthode à distance.
- *la création/souscription à un topic de type publication/abonnement :* la création du topic est réalisée par l'opération `export`. La souscription au topic est réalisée par la méthode `bind`.

7.3.2 Implantation du patron export-bind

Principe

Le patron `export-bind` est mis en œuvre par des composants **protocoles** qui implantent une interface `Protocol` définissant deux méthodes : `export` et `bind`. Ces composants sont généralement organisés en pile ou en graphe, chaque protocole utilisant des services fournis par les protocoles de niveaux inférieurs dans la pile (ou dans le graphe).

Les composants protocoles gèrent des composants **usines de sessions** — créés par la méthode `export` — et des **sessions** — créées par la méthode `bind` et par les usines de sessions. Une session représente un canal de communication défini par deux interfaces, `IncomingPush` et `OutgoingPush`, représentées sur la figure 7.10. L'interface `IncomingPush` définit la méthode `incomingPush` que les sessions de niveau inférieur utilisent pour transmettre des messages à la session. La méthode `incomingClosed` sert à notifier à la session qu'une session de niveau inférieur a été fermée de façon inattendue. L'interface `OutgoingPush` définit la méthode `outgoingPush` qui permet aux sessions de niveau supérieur de transmettre des messages à la session. La méthode `outgoingClose` permet de fermer la session.

Architecture

L'architecture des composants protocole est représentée sur la figure 7.11. Cette figure représente une pile de deux protocoles. Chaque composant protocole est implanté par un composite constitué de trois sous-composants :

- le composant `SessionFactoryContainer` encapsule les composants usines de sessions (`SessionFactory`).

```
public interface IncomingPush {
    void incomingPush(Message m);
    void incomingClosed(Exception e);
}

public interface OutgoingPush {
    void outgoingPush(Message m);
    void outgoingClose();
}
```

FIG. 7.10 – Les interfaces `IncomingPush` et `OutgoingPush`.

– le composant `SessionContainer` encapsule les composants sessions (`Session`).
– le composant (primitif) `ProtocolImplementation` implante l'interface `Protocol`.

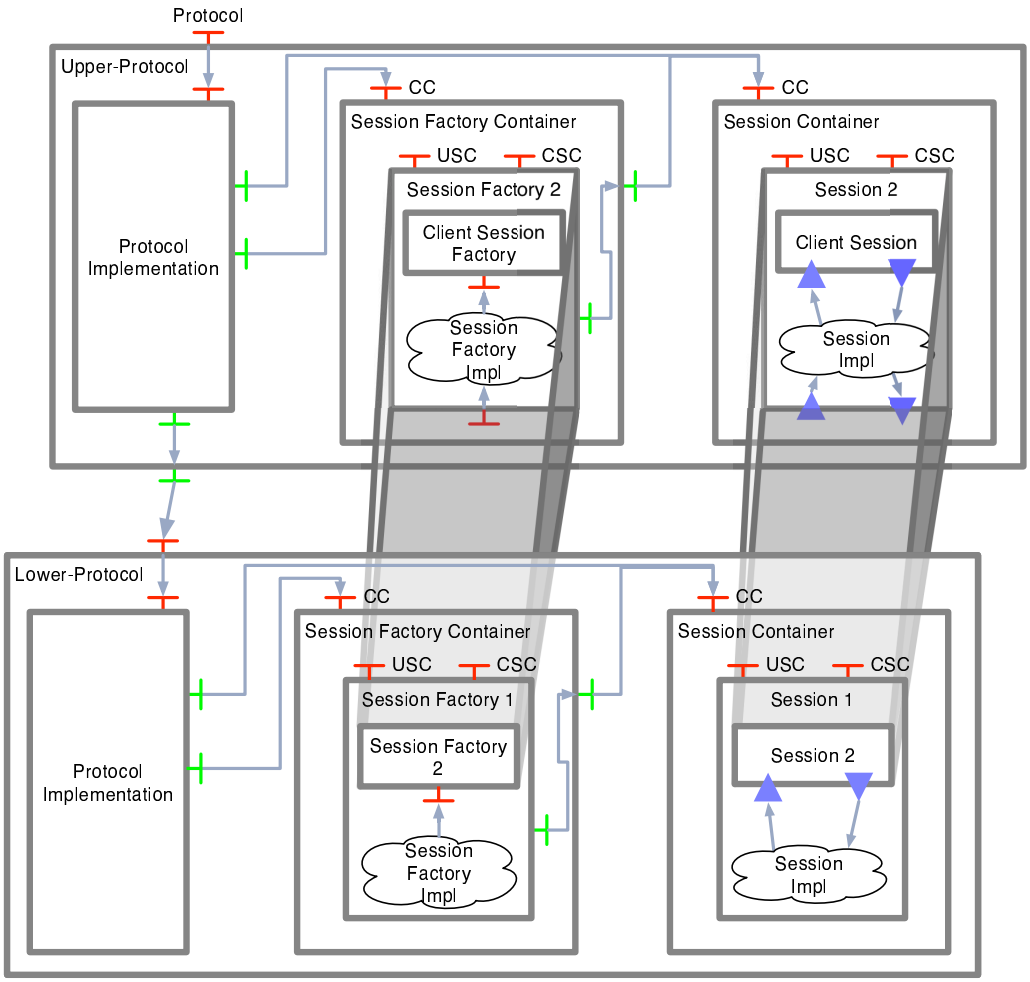

FIG. 7.11 – Architecture d'une pile de deux protocoles.

Chaque composant `SessionFactory` est un composite composé de deux parties : une implan-

tation — représentée par le nuage — et un ou plusieurs composites[6] qui sont les `SessionFactory` exportées par les protocoles de plus haut niveau dans le graphe. En conséquence, chaque composant `SessionFactory` est partagé par le `SessionFactoryContainer` du protocole qui l'a créé et par un composite `SessionFactory` du protocole auprès duquel ce composant `SessionFactory` a été exporté.

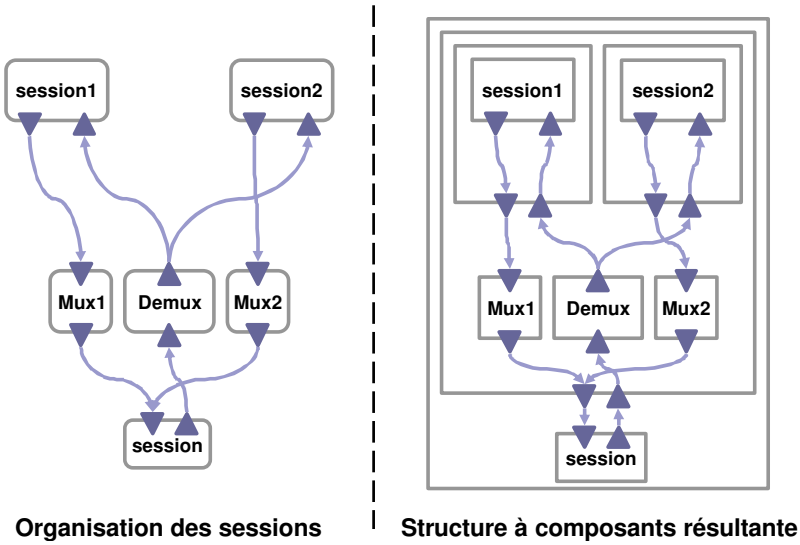

FIG. 7.12 – Structure récursive des composants `SessionFactory`.

La figure 7.12 représente l'organisation des `SessionFactory` dans le cas d'un assemblage de protocoles organisés en un graphe de forme Y. Pour des raisons de lisibilité, nous n'avons pas représenté les composants `SessionFactoryContainer`. La structure obtenue est totalement récursive : à chaque niveau, le composite `SessionFactory` contient son implantation et les composites représentant les `SessionFactory` qu'il exporte. Ainsi, le composite le plus englobant représente la `SessionFactory` du protocole de plus bas niveau dans le graphe[7].

La structure des composants `Session` est semblable à celle des composants `SessionFactory`. Nous retrouvons la même structure récursive des `Session`.

L'intérêt de la structure avec composants partagés réside dans le fait que l'architecture logicielle rend explicite l'organisation des différentes sessions. Ainsi, les structures en Y, ou Y inversé des graphes de protocoles sont modélisés par l'assemblage de composants représentant les sessions. Il est ainsi trivial de trouver dynamiquement l'ensemble des sessions formant une liaison afin de l'administrer (i.e. changer un des protocoles utilisés, détruire la liaison, etc.).

7.3.3 Exemple du protocole TCP/IP

Interfaces

Dans cette section nous illustrons l'architecture décrite dans la section précédente à l'aide du protocole TCP/IP.

L'interface `Protocol` implantée par le protocole TCP/IP est représentée sur la figure 7.13.

[6] Par souci de clarté, nous n'en avons représenté qu'un sur la figure 7.11.

[7] S'il existe plusieurs protocoles de plus bas niveau dans le graphe de protocoles (graphe en Y inversé), la structure résultante fait intervenir du partage au niveau de l'ensemble des composites de plus haut niveau.

```
public interface Protocol {
   ExportId export(ChannelFactory channelFactory);
   OutgoingPush bind(ExportId exportId, IncomingPush toClientPush);
}
```

FIG. 7.13 – L'interface `Protocol`.

La méthode `export` permet d'exporter une interface de type `ChannelFactory`, représentant une usine de canaux de communication. Cette méthode crée un composant `SessionFactory` encapsulant une `ServerSocket`. Elle retourne un identifiant de l'interface exportée contenant l'adresse IP et le numéro de port d'écoute de la `ServerSocket`.

La méthode `bind` permet d'instancier un canal de communication vers une interface précédemment exportée. Cette méthode retourne une interface `OutgoingPush` qui doit être utilisée par le client pour émettre les messages sur le canal. Par ailleurs, le paramètre `toClientPush` spécifie l'interface par laquelle le client (le composant qui appelle la méthode `bind`) souhaite recevoir les messages émis par le serveur (le composant qui a appelé la méthode `export`). Dans le cas du protocole TCP/IP, la méthode `bind` a pour résultat l'ouverture d'une connexion TCP vers la `ServerSocket` identifiée par le paramètre `exportId`.

L'interface `ChannelFactory` (figure 7.14) exportée à l'aide de la méthode `export` définit une méthode `instantiate` qui est appelée lorsqu'une connexion est acceptée sur la `ServerSocket`, c'est-à-dire lorsqu'un client appelle la méthode `bind`. Le paramètre `toClientPush` spécifie l'interface sur laquelle peuvent être émis les messages à destination du client qui s'est connecté. Par ailleurs, la méthode `instantiate` retourne une interface utilisée pour transmettre les messages en provenance du client.

```
public interface ChannelFactory {
   IncomingPush instantiate(OutgoingPush toClientPush);
}
```

FIG. 7.14 – L'interface `ChannelFactory`.

Diagramme de séquence

La figure 7.15 présente un exemple de séquence d'événements dans l'établissement d'un canal de communication TCP/IP entre un client et un serveur.

1. Le protocol `ServeurProtocol` crée un composant `ChannelFactory` implantant l'interface `ChannelFactory` et l'exporte auprès du protocole TCP/IP. Celui-ci crée un composant `SessionFactory` contenant une tâche dont le rôle est d'attendre des demandes de connexion TCP/IP sur une `ServerSocket`. Par ailleurs, le composant `ChannelFactory` est ajouté comme sous-composant du composant `SessionFactory` et il est lié à son composant implantation. La méthode `export` retourne un identifiant composé de l'adresse IP et du numéro de port d'écoute de la `ServerSocket`.

2. Pour se lier à la session exportée, le composant `Client` doit préalablement créer une session `ClientSession`. Le client peut ensuite demander au protocole TCP/IP l'établissement d'une liaison vers le serveur précédemment exporté. Pour ce faire, il donne en paramètres l'iden-

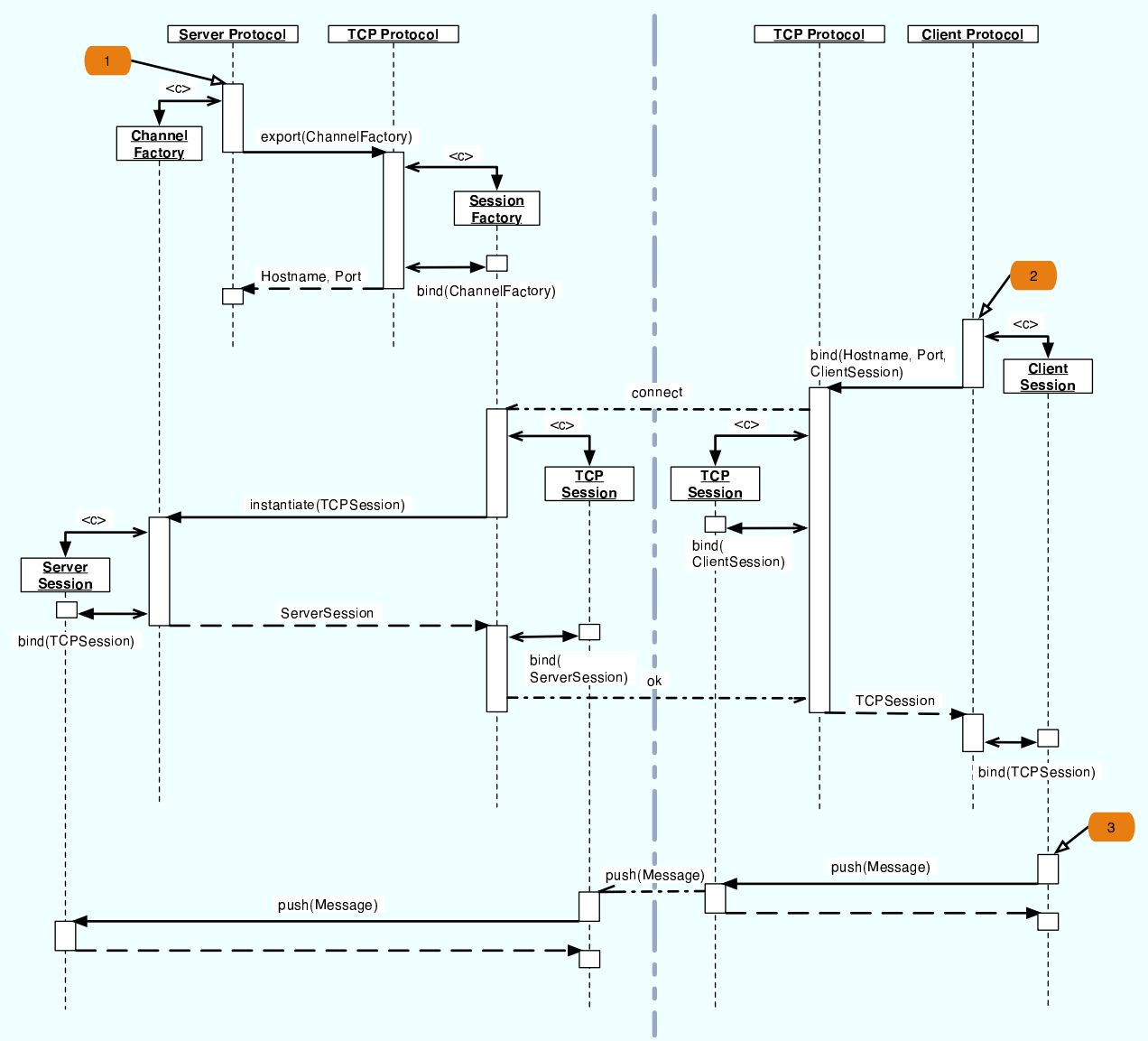

FIG. 7.15 – Séquence d'événements pour l'établissement de sessions pour le protocole TCP/IP.

tifiant (IP, port) créé par l'export du composant `ChannelFactory` et l'interface `IncomingPush` du composant `Client session` (interface par laquelle il souhaite recevoir d'éventuels messages en provenance du server). Le protocol TCP/IP ouvre une socket et crée un composant `TCPSession` qui l'encapsule.

Du côté serveur, l'ouverture d'une nouvelle connexion TCP/IP déclenche la création d'une nouvelle session `TCPSession` encapsulant une socket. La `SessionFactory` signale ensuite au composant `ChannelFactory` l'établissement d'une nouvelle session. Pour ce faire, elle appelle la méthode `instantiate` en passant en paramètre l'interface `OutgoingPush` du composant `TCPSession`. Il en résulte la création d'une session au niveau du composant `Server` (`ServerSession`) et sa liaison bidirectionnelle avec la session `TCPSession`. La `SessionFactory` envoie un message via la connexion TCP/IP afin de notifier au client que l'instanciation du canal de communication côté serveur s'est terminée correctement.

Sur le site client, la reception de ce message permet à la méthode `bind` de se terminer en retournant l'interface `OutgoingPush` du composant `TCPSession`. Ainsi, le composant `ClientProtocol` peut lier de façon bidirectionnelle la session `ClientSession` préalablement créée.

7.4 Les composants de la bibliothèque DREAM

Cette section débute par une présentation générale de la bibliothèque de composants DREAM. Nous détaillons ensuite l'architecture de deux composants, afin d'illustrer la variété des composants qui sont proposés, ainsi que pour montrer les possibilités de configuration qu'offre chacun des composants.

7.4.1 Présentation générale

La bibliothèque possède des composants encapsulant les diverses fonctions trouvées dans les intergiciels de communication. Cette section dresse une liste non exhaustive de ces composants. Ces composants sont décrits plus en détail dans [LQS05].

Les files de messages servent à stocker les messages. Elles ont une entrée — qui est utilisée par les autres composants pour stocker des messages —, et une sortie — que la file utilise pour délivrer les messages. La bibliothèque DREAM fournit plusieurs files qui diffèrent par :

- les modes de connexion de l'entrée et de la sortie (push vs. pull) ;
- la manière dont les messages sont triés (FIFO, LIFO, numéro de séquence, etc.) ;
- le comportement de la file dans les différents états : file pleine (bloque vs. détruit des messages), file vide, etc.

Les files font l'objet d'une étude détaillée dans la section 7.4.2.

Les transformateurs sont des composants avec une entrée et une sortie. Chaque message reçu sur l'entrée est transformé, puis délivré sur la sortie. Les transformateurs peuvent être en mode push ou en mode pull. Un exemple typique de transformation consiste à rajouter une adresse IP à un message.

Les pompes sont des composants avec une entrée pull et une sortie push. Les pompes ont une activité qui consiste à récupérer un message sur l'entrée et à le délivrer sur la sortie. Cette activité étant sans état, il est possible de l'exécuter simultanément par plusieurs threads.

Les routeurs ont une entrée et plusieurs sorties. Le rôle d'un routeur est de router les messages reçus en entrée sur une ou plusieurs sorties. DREAM fournit plusieurs routeurs qui se distinguent par :

- le mode de connexion de l'entrée et des sorties ;
- la technique de duplication (par référence ou par valeur) lorsque le message doit être routé sur plusieurs sorties ;
- l'algorithme de routage.

Les agrégateurs/désagrégateurs Les agrégateurs sont des composants avec une ou plusieurs entrées et une sortie. Leur rôle est de délivrer sur la sortie un agrégat des messages reçus en entrée. Les désagrégateurs implémentent le comportement inverse des agrégateurs.

Les sérialiseurs/désérialiseurs permettent de (dé)sérialiser les messages. Un message sérialisé est un message qui ne contient qu'un chunk, appelé `ByteArrayChunk`, qui contient un tableau d'octets.

Les composants de diffusion permettent de diffuser un message à un ensemble de destinations. Pour ce faire, ils créent, à partir du message à diffuser, plusieurs messages contenant une destination appropriée. DREAM fournit diverses implantations : unicast "naïf" à l'ensemble des destinations, utilisation d'arbres couvrants, utilisation de chaînes de diffusion (i.e. arbres d'arité égale à 1).

Les composants de protocole sont en charge de faire respecter un protocole par n processus. Ils ont une entrée/sortie pour les messages entrants, et une entrée/sortie pour les messages sortants. La bibliothèque DREAM contient les protocoles suivants :

- *fragmentation* : ce composant permet de fragmenter des messages en plusieurs messages d'une taille spécifiée par un contrôleur d'attributs. Ce protocole est utile en cas d'utilisation de canaux ne fragmentant pas les messages de façon native (e.g. UDP).
- *ordonnancement FIFO* : ce composant estampille les messages avec un numéro de séquence afin d'assurer leur ordonnancement FIFO.
- *ordonnancement causal* : ce composant possède une horloge matricielle qu'il utilise pour estampiller les messages sortants, et pour garantir une délivrance causale des messages entrants.
- *acquittement négatif (NACK)* : ce composant implante un acquittement négatif des messages, c'est-à-dire qu'il notifie aux processus émetteurs de messages les numéros de séquence de messages qui n'ont pas été reçus afin que ceux-ci les retransmettent. Ce protocole permet de fiabiliser des échanges sur canaux non fiables (e.g. UDP, IP Multicast).
- *diffusion avec ordonnancement total et uniforme* : ce composant permet à n processus de diffuser des messages de façon fiable avec ordonnancement total et uniforme. Une description détaillée de ce composant est effectuée au chapitre 11.
- *diffusion probabiliste* : ce composant permet à n processus de diffuser des messages de façon non fiable à l'aide d'algorithmes probabilistes. Ce composant est présenté dans la section 7.4.3.

Les canaux permettent l'échange de messages entre différents espaces d'adressages. Un canal est composé de deux composants : un canal sortant (*channel out*) — qui permet d'envoyer des messages vers un autre espace d'adressage —, et un canal entrant (*channel in*) — qui permet de recevoir des messages en provenance d'autres espaces d'adressage. DREAM fournit actuellement des canaux permettant l'envoi des messages à l'aide des protocoles TCP, UDP et IP Multicast.

7.4.2 Les files de messages

Cette section décrit l'architecture des files de messages. Nous commençons par décrire l'architecture générale des files, puis étudions les divers composants qui les constituent.

Architecture

L'architecture des files de messages DREAM est représentée sur la figure 7.16. Une file est un composite (`Queue`) qui possède une entrée et une sortie — qui peuvent être en mode *push ou pull* — et qui encapsule trois composants :

- le composant `IncomingHandler` traite les messages arrivant, c'est-à-dire les messages qui doivent être stockés dans la file. Son rôle est principalement de traiter les cas de débordement de la capacité de la file par différents moyens : blocage, suppression de messages, levée d'exception, etc.
- le composant `Buffer` permet le stockage des messages. Les buffer peuvent implanter diverses politiques d'ordonnancement : FIFO, LIFO, avec numéro de séquence, etc.
- le composant `OutgoingHandler` est responsable de la délivrance des messages stockés dans le buffer. Son rôle est principalement de traiter les demandes lorsque la file est vide : blocage, retour de pointeur `null`, levée d'exception, etc.

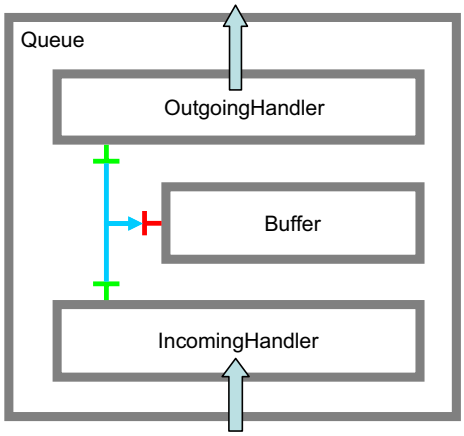

FIG. 7.16 – Architecture des files de messages.

Le composant `Buffer`

Le composant `Buffer` est responsable du stockage des messages. Chaque composant `Buffer` a une capacité (éventuellement infinie) qui correspond au nombre de messages qu'il est capable de stocker. Cette capacité est dynamiquement modifiable à l'aide d'un contrôleur d'attributs. Un buffer implante deux interfaces représentées sur la figure 7.17. L'interface `BufferAdd` est utilisée par le composant `IncomingHandler` pour stocker des messages. La méthode `BufferRemove` permet au composant `OutgoingHandler` d'accéder aux messages stockés. Les méthodes `add`, `get` et `remove` sont bloquantes. Les méthodes `tryAdd`, `tryGet` et `tryRemove` ne le sont pas. Par exemple, si la capacité maximale du `Buffer` a été atteinte, la méthode `add` bloquera jusqu'à ce qu'un message soit supprimé ; en revanche, la méthode `tryAdd` retournera immédiatement le booléen `false` signifiant que le message n'a pas été ajouté.

```
public interface BufferAdd {
   void add(Message message) throws InterruptedException;
   boolean tryAdd(Message message);
}

public interface BufferRemove {
   Message get() throws InterruptedException;
   Message tryGet();
   Message remove() throws InterruptedException;
   Message tryRemove();
   Message removeAll();
}
```

FIG. 7.17 – Les interfaces `BufferAdd` et `BufferRemove`.

La bibliothèque DREAM fournit plusieurs implantations du composant `Buffer`, parmi lesquelles :

- le `Buffer` *FIFO* retourne les messages dans l'ordre dans lequel ils ont été ajoutés.
- le `Buffer` *LIFO* retourne les messages dans l'ordre inverse de celui dans lequel ils ont été ajoutés.
- le `Buffer` *filtrant* retourne les messages dans l'ordre dans lequel un composant `Filtre` les renvoie. Ce composant est utilisé par le `Buffer` à chaque appel des méthodes `get` et `remove` pour déterminer le message à retourner.
- le `Buffer` *triant* trie les messages à l'aide d'une clé qui est générée par un composant `ComparableKeyManager` pour chaque message stocké. Les messages sont retournés par le buffer par ordre de clé ascendante.

Le composant `IncomingHandler`

Le composant `IncomingHandler` possède une interface `Push` sur laquelle il reçoit les messages en entrée de la file. Il utilise l'interface `BufferAdd` du composant `Buffer` pour stocker ces messages. Les implantations actuellement fournies par la bibliothèque DREAM de ce composant sont les suivantes :

- le composant `IncomingHandler` *bloquant* utilise la méthode bloquante `add` pour ajouter les messages. Si la capacité du `Buffer` a été atteinte, le flot d'exécution du composant demandant le stockage du message est bloqué.
- le composant `IncomingHandler` *supprimant* utilise la méthode non bloquante `tryAdd` pour ajouter les messages. Si un message ne peut être stocké, le composant choisit soit de supprimer le message à stocker, soit de supprimer un message du `Buffer` à l'aide de l'interface `BufferRemove`. Le choix de l'une ou l'autre des stratégies est effectué à l'aide d'un contrôleur d'attributs.
- le composant `IncomingHandler` *à exception* utilise la méthode non bloquante `tryAdd` pour ajouter les messages. Si un message ne peut être stocké, le composant `IncomingHandler` lève une exception.

Le composant `OutgoingHandler`

Le composant `OutgoingHandler` possède une interface `Pull` qui est utilisée par les autres composants pour récupérer des messages stockés dans la file. Le composant `OutgoingHandler`

obtient ces messages à l'aide de l'interface `BufferRemove` du composant `Buffer`. La bibliothèque DREAM fournit les implantations suivantes :

- le composant `OutgoingHandler` *bloquant* utilise la méthode bloquante `remove` pour récupérer les messages du `Buffer`. Si le `Buffer` est vide, le flot d'exécution du composant demandant le message est bloqué.
- le composant `OutgoingHandler` *non bloquant* utilise la méthode non bloquante `tryRemove` pour récupérer les messages du `Buffer`. Si le `Buffer` est vide, le pointeur `null` est retourné.
- le composant `OutgoingHandler` *à exception* utilise la méthode non bloquante `tryRemove` pour récupérer les messages du `Buffer`. Si le `Buffer` est vide, une exception est levée.
- le composant `OutgoingHandler` *agrégeant* utilise la méthode non bloquante `removeAll` pour récupérer tous les messages du `Buffer` sous forme d'un message agrégé.

Différentes personnalités de files

Les différents composants présentés dans les sections précédentes peuvent être assemblés afin de construire différentes personnalités de files. Il est par exemple possible de construire une file triant les messages à l'aide d'une clé, bloquante lorsqu'elle est pleine et retournant un pointeur `null` lorsqu'elle est vide. Il suffit de modifier le composant `IncomingHandler` pour obtenir une file qui lève une exception lorsqu'elle est pleine. Par ailleurs, il est possible d'ajouter des composants à l'architecture générique présentée pour obtenir certains comportements. Par exemple, il est aisé de construire une file ayant son entrée et sa sortie en mode `Push` : pour cela, il suffit de lier la sortie `Pull` du composant `OutgoingHandler` à un composant `Pompe`. Un tel composant permet de découpler les flots d'exécution des deux composants reliés par la file.

7.4.3 Le composant protocolaire LPBCast

Cette section présente un exemple de composant implantant un protocole[8]. Ce protocole, appelé *LPBcast* (*Lightweight Probabilistic Broadcast*), permet d'effectuer des *diffusions probabilistes* de messages entre un ensemble de processus [EGH+01]. Par opposition aux protocoles de diffusion non probabilistes, dans *lpbcast*, chaque processus ne connaît qu'un sous-ensemble des autres processus et ne stocke qu'un sous-ensemble des messages diffusés (pour une retransmission ultérieure éventuelle). Cette propriété garantit le passage à l'échelle du protocole. Chaque processus diffuse les messages au sous-ensemble des processus qu'il connaît. Plus la connaissance des processus est uniformément répartie au sein des différents processus, plus la probabilité qu'un message soit diffusé à l'ensemble des processus est grande.

La figure 7.18 représente l'implantation de *lpbcast* réalisée à l'aide de DREAM. Comme nous l'avons présenté dans la section 7.4.1, les composants de protocole possèdent une entrée/sortie pour les messages entrant, et une entrée/sortie pour les messages sortant. Les quatre composants au milieu du composite représentent les structures de données utilisées dans le protocole : `View` contient la liste des processus connus par le processus auquel ce composant appartient ; `Subs` (resp. `UnSubs`) contient la liste des souscripteurs[9] (resp. dé-souscritpeurs) ; `EventIds` est la liste des identifiants de messages reçus. La file de message (`MessageQueue`) est une file FIFO bloquante qui stocke les messages à diffuser. Ces messages, soit sont produits localement (via l'interface `outgoing-in-push`), soit ont déjà été reçus et vont être retransmis pour que d'autres processus les reçoivent.

[8] En toute rigueur, le composant présenté correspond à l'implantation d'une session dans le sens présenté dans la section 7.3 sur l'établissement de liaisons.

[9] Les souscripteurs sont les processus qui veulent s'abonner au protocole de diffusion.

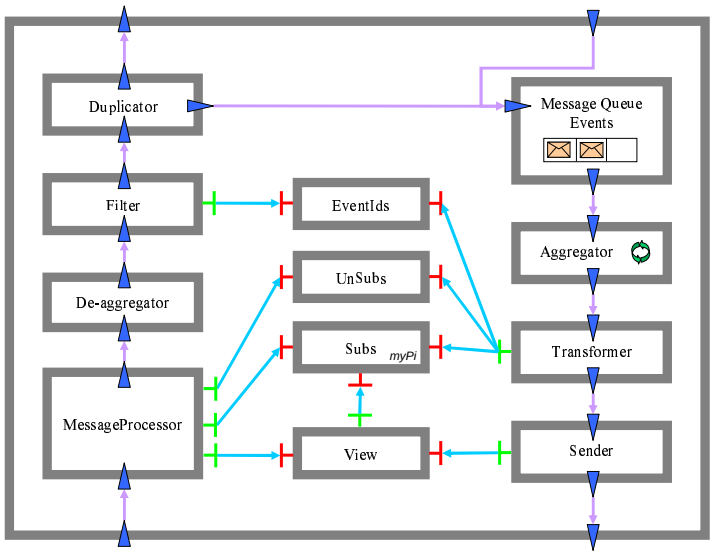

FIG. 7.18 – Implémentation du composant protocolaire LPBcast.

Le principe du protocole est le suivant : toutes les T ms, le composant `Aggregator` construit un message agrégé contenant l'ensemble des messages stockés dans la file. Ce message est ensuite transformé par le composant `Transformer` qui ajoute des chunks contenant les listes maintenues par les composants `EventIds`, `Subs` et `UnSubs`. Le rôle du composant `Sender` est de choisir un sous-ensemble des processus connus auxquels le message va être diffusé.

Les messages arrivant sont traités de la façon suivante : un composant `MessageProcessor` utilise le message reçu pour mettre à jour les composants `View`, `Subs` et `UnSubs`. Le message est ensuite dé-agrégé par le composant `DeAggregator` qui transmet le message au composant `Filter` dont le rôle est de déterminer si le message a déjà été délivré localement. S'il ne l'a pas été, il est transmis au composant `Duplicator` qui délivrera le message à l'aide de l'interface `incoming-out-push` et qui placera une copie du message dans la file des messages à diffuser. Notons que les listes maintenues par les composants `View`, `Subs`, `UnSubs` et `EventIds` sont tronquées lorsqu'elle atteignent une certaine taille.

L'implantation de ce protocole à l'aide de DREAM est intéressante du fait qu'il est aisé de changer (dynamiquement) l'implantation des composants qui maintiennent les structures de données (`View`, `Subs`, `UnSubs` et `EventIds`). Cela permet de construire facilement les différentes variantes du protocole décrites dans [JGKvS04]. Par ailleurs, la structure à composants a facilité l'intégration des fonctions de fiabilité décrites dans [EGH+03].

7.5 Conclusion

Dans ce chapitre, nous avons présenté la bibliothèque de composants DREAM. Nous avons tout d'abord décrit une extension de FRACTAL pour la gestion des activités au sein des composants. Cette extension est un canevas qui permet de déployer des organisations arbitrairement complexes de tâches et d'ordonnanceurs. Nous avons ensuite présenté les abstractions et les composants permettant de gérer les messages échangés entre les composants de la bibliothèque DREAM. Nous avons également décrit un ensemble d'abstractions et d'outils pour l'établissement de liaisons entre des composants répartis. Ces outils mettent en œuvre, à l'aide d'une structure à composants complexe, le patron de conception `export-bind`, développé dans le cadre

du projet Jonathan [DHTS99]. Enfin, nous avons conclu ce chapitre par une présentation de la bibliothèque de composants DREAM. Nous avons tout particulièrement détaillé l'architecture de deux composants : les files de messages et un composant implantant un protocole de diffusion probabiliste.

Remerciements : les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Matthieu Leclercq, ingénieur expert au sein du projet SARDES.

Chapitre 8

Gestion de configuration

Sommaire

Ce chapitre décrit les outils de gestion de configuration fournis par le canevas DREAM. La gestion de configuration regroupe plusieurs fonctions couvrant diverses étapes du cycle de vie d'une application : description, configuration, vérification, déploiement, reconfiguration, etc. DREAM utilise le langage de description d'architectures extensible FRACTAL ADL[1] pour effectuer la description, la configuration et le déploiement des architectures construites à l'aide de la bibliothèque. Dans ce chapitre, nous présentons trois outils qui sont des extensions de FRACTAL ADL permettant d'effectuer :

- des **reconfigurations de structure** sur une architecture en cours d'exécution. Cet outil autorise notamment des ajouts de composants et de liaisons au sein d'une architecture déployée.
- des **reconfigurations d'implantation** : cet outil permet de mettre à jour le code des composants.
- des **vérifications de types** sur les assemblages décrits dans l'ADL. Cet outil sert, par exemple, à vérifier avant le déploiement que chaque composant recevra des messages ayant la structure attendue (i.e. les chunks appropriés).

8.1 Reconfiguration de structure

Nous avons étendu le langage de description d'architectures FRACTAL ADL de manière à autoriser des modifications de structure des applications en cours d'exécution. Ces modifications de structure consistent en l'ajout de composants au sein de composites existants et en

[1] FRACTAL ADL est présenté dans le chapitre 6.

l'établissement de liaison entre les composants ajoutés et les composants patrimoniaux. Nous commençons cette section par un exemple illustrant l'utilisation de l'outil que nous proposons. Nous en présentons ensuite la mise en œuvre.

8.1.1 Exemple

La figure 8.1 représente un exemple de reconfiguration de structure. Le composite `HelloWorld` contient initialement un `client` lié à un `duplicator` qui délègue les appels à deux composants, `server 1` et `server 2`. Le reconfiguration de structure consiste à ajouter un troisième composant, `server 3`, auquel le `duplicator` délèguera également les appels.

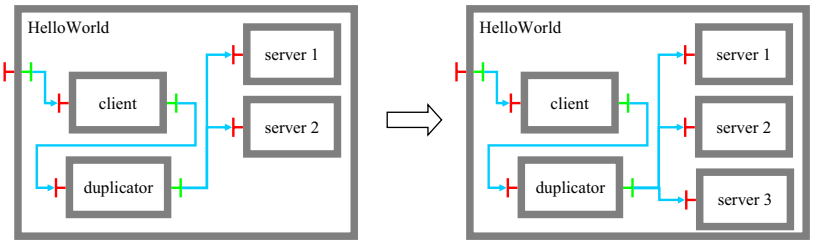

FIG. 8.1 – Exemple de reconfiguration de structure.

Une telle reconfiguration est possible programmatiquement : il est nécessaire de stopper le composite `HelloWorld`. Ce dernier ne recevra, ainsi, plus d'appels sur ses interfaces serveurs et n'en émettra plus sur ses interfaces clientes. Il faut ensuite créer le composant `server 3`, l'ajouter dans le composite `HelloWorld` à l'aide de son contrôleur de contenu (`ContentController`), puis effectuer la liaison entre le `duplicator` et le `server 3`. Enfin, il faut redémarrer le composite `HelloWorld`.

L'outil que nous proposons est une extension de FRACTAL ADL qui permet d'effectuer l'ensemble de ces tâches à partir d'une description ADL. Ainsi, concernant l'exemple de la figure 8.1, l'administrateur de l'application doit uniquement écrire la description ADL représentée sur la figure 8.2 et utiliser l'usine pour l'"appliquer" au composant `HelloWorld`.

```
<definition name="AddServer" argument="num">
  <component name="duplicator">
    <interface name="s" role="client" cardinality="collection"
               signature="Service"/>
    <legacy contingency="mandatory"/>
  </component>
  <component name="server${num}" definition="Server"/>
  <binding client="duplicator.s${num}" server="server${num}.s"/>
</definition>
```

FIG. 8.2 – Description ADL de la reconfiguration de structure.

Cette description ADL spécifie qu'il existe un composant `duplicator` possédant une interface client de collection de nom `s` et de signature `Service`. L'élément `legacy` indique que ce composant est patrimonial. Par ailleurs, la description spécifie qu'il est nécessaire de créer un composant de nom `server${num}`[2]. Enfin, il est indiqué qu'il faut effectuer une liaison entre le composant

[2] `${num}` signifie la valeur de l'argument `num` qui est donné lorsque la description est parsée par l'usine ADL. Cela permet d'utiliser la même description ADL pour ajouter un troisième, quatrième, cinquième, ... composant `server`.

`duplicator` et le composant `server${num}`.

La section suivante décrit les modifications qui ont été faites à l'usine pour prendre en compte ces descriptions ADL de reconfigurations de structure.

8.1.2 Mise en œuvre

La prise en compte des ADL de reconfigurations de structure a nécessité l'extension des trois composants principaux de l'usine : *loader*, *compiler* et *builder*.

L'extension du composant *loader* consiste en l'ajout d'un nouveau loader, *legacy loader*, en tête de la chaîne des *loaders*. Ce composant effectue en parallèle un parcours de l'arbre abstrait et de la structure à reconfigurer. Il vérifie que les composants patrimoniaux existent et sont compatibles avec leur description ADL : leur type doit être un sous-type du type décrit dans l'ADL.

Le composant *compiler* a également été modifié : nous avons ajouté un *legacy compiler* dont le rôle est de créer des tâches d'instanciation "fictives" pour les composants patrimoniaux. Le rôle de ces tâches est de fournir, aux tâches qui en dépendent, les instances de composants patrimoniaux au lieu de les créer.

Enfin, il a été nécessaire de modifier le composant *builder*. Le builder responsable de l'ajout de sous-composants dans un composite a été modifié pour vérifier que le composant à ajouter n'est pas déjà présent. Par ailleurs, le builder en charge d'établir les liaisons entre composants doit rompre la liaison si elle a déjà été établie.

8.2 Reconfiguration d'implantation

La plupart des modèles de composants implantés en Java ne permettent pas d'effectuer des reconfigurations d'implantation. Cette limitation est due aux contraintes imposées par le mécanisme de chargement de classes de Java (*class loader*). Il est, par exemple, impossible de charger deux versions d'une même classe, ou encore de décharger une classe. Deux possibilités s'offrent aux développeurs Java pour fournir des mécanismes de reconfiguration d'implantation :

- *modification du code des classes chargées* pour garantir que deux versions d'une même classe portent des noms différents. Cette approche est utilisée dans JPloy [LvdH04] et SOFA [HT03] : ces deux modèles utilisent des manipulations de *bytecode* pour garantir que les différentes versions d'une même classe sont chargées avec des noms différents. Ces noms sont générés à partir de fichiers de descriptions (comparables à des descriptions ADL) qui spécifient les versions des classes utilisées. L'avantage de cette approche est qu'elle n'engendre aucun surcoût à l'exécution. En revanche, elle rend impossible l'utilisation de certaines méthodes du langage Java. Il n'est, par exemple, pas possible d'utiliser certaines méthodes de construction de classe par réflexion (e.g. `Class.forName(String name)`), étant donné que celle-ci se base sur le nom de la classe à créer — nom qui peut avoir été modifié lors du chargement.
- *utilisation de class loaders différents* pour charger les différentes classes. Cette solution est utilisée par les plates-formes de services telles OSGi [Ope03] et par les serveurs d'applications J2EE [j2e02] (IBM WebSphere [Web03], BEA WebLogic [Web04], JOnAS [JOn04], etc.). La plate-forme de services OSGi permet de déployer des applications Java empaquetées sous forme de *bundles*. Un bundle contient des fichiers jar et des méta-données sur ces fichiers jar (e.g. version). Le rôle d'une plate-forme OSGi est de gérer le cycle de vie des bundles. OSGi impose une contrainte qui rend son utilisation impossible dans notre contexte : il ne permet pas de faire coexister deux versions d'une même classe. Concernant

les serveurs d'applications J2EE, l'utilisation de plusieurs class loaders est faite dans le but de pouvoir isoler les différentes applications déployées sur un serveur. Les class loaders sont organisés en arbre, chaque class loader ayant un unique parent auquel il peut éventuellement déléguer le chargement de classes. Cette technique s'avère suffisante pour les applications ciblées, mais elle est trop restrictive dans un modèle de composants flexible comme FRACTAL.

La solution que nous proposons est d'utiliser une organisation arbitraire de class loaders : cette organisation est construite en se basant sur les frontières de composants et les besoins de reconfigurabilité exprimés par le développeur d'applications à l'aide de FRACTAL ADL. Les class loaders sont créés et administrés à l'aide de *Module loader* [Hal04], un système permettant de faire coopérer un ensemble de class loaders.

Cette section s'organise de la façon suivante : nous présentons tout d'abord les concepts de notre solution. Nous détaillons ensuite sa mise en œuvre, avant de conclure par une évaluation de la surcharge en temps d'exécution induite par son utilisation.

8.2.1 Versionnement et reconfiguration d'implantation dans les applications FRACTAL

Les contraintes du class loader Java

Le class loader Java [SB98] permet de charger dynamiquement des classes Java au sein d'une machine virtuelle. Chaque class loader charge les classes à partir d'une source définie : système de fichiers, réseau, etc. Le rôle d'un class loader est de créer des objets `Class` à partir de fichiers `.class`. Pour ce faire, chaque class loader implémente une méthode `loadClass(String name)` qui a pour but de charger une classe. Cette méthode utilise la méthode `findLoadedClass(String name)` pour vérifier que la classe n'a pas déjà été chargée, auquel cas elle est stockée dans un cache. Si tel n'est pas le cas, `loadClass` utilise `defineClass` pour construire un objet `Class` à partir d'un tableau d'octets. Notons qu'un class loader peut également déléguer le chargement d'une classe à un autre class loader. Ce mécanisme est majoritairement utilisé avec des hiérarchies en arbre de class loaders : chaque class loader a un père auquel il peut déléguer le chargement.

Comme il est décrit dans [SB98], une classe est définie par un couple $< C, L_d >$, dans lequel C est le nom de la classe et L_d est le class loader qui l'a chargée. En conséquence, la même classe chargée par deux class loaders différents est considérée comme deux classes différentes : les classes $< C, L_d >$ et $< C, L'_d >$ sont incompatibles, quand bien même elles correspondent au même bytecode. L'utilisation d'une classe à la place de l'autre lève une exception `ClassCastException`.

Les class loaders ne permettent pas de décharger des classes. En conséquence, les nouvelles versions d'une classe doivent toujours être chargées par un nouveau class loader. Pour éviter les `ClassCastException`, il peut être nécessaire de recharger transitivement une partie de l'application. Par exemple, supposons qu'une classe C_1 référence une classe C_2 qui référence elle même une classe C_3. Chaque fois qu'une nouvelle version de C_3 est chargée, il est également nécessaire de recharger C_1 et C_2. En revanche, pour éviter qu'il soit nécessaire de recharger C_3 à chaque nouveau chargement de C_1[3], il faut utiliser le mécanisme de délégation de chargement de classes. Ce mécanisme permet d'assurer que le class loader chargeant une nouvelle version de C_1 chargera C_3 avec le class loader qui l'avait précédemment chargée.

[3]Nous supposons que C_3 ne référence pas C_1.

Organisation des class loaders

Une application FRACTAL est formée d'un ensemble de composants. Chaque composant est constitué d'un ensemble de classes (décrites au chapitre 6). La question à laquelle nous devons répondre est : à quels class loaders faut-il déléguer le chargement de chacune des classes ?

Selon Szyperski [Szy02], "un composant est déployable indépendamment et il est sujet à composition par une tierce partie". Une approche naïve pour rendre les composants deployables indépendamment consiste à charger chaque composant dans un class loader indépendant. Cette approche permet de faire coexister plusieurs versions d'un même composant ; elle permet également de recharger des composants. Cependant elle n'est pas correcte car les composants ne sont pas composables par une tierce partie. En effet, toute tentative de liaison entre les interfaces de deux composants résulterait en une `ClassCastException`.

Il apparaît donc logique de séparer le chargement des *classes d'interfaces* du composant (fonctionnelles et de contrôle) — destinées à être utilisées par les autres composants —, de celui des *classes d'implémentation* — destinées à l'usage "privé" du composant. Supposons que l'on ait deux composants C_1 et C_2 qui communiquent via une interface `Push` qui définit une méthode `void push(Message m)`. Les classes `Push` et `Message` doivent être chargées par un class loader commun aux deux composants tandis que les classes d'implantation des composants peuvent être chargées par des class loaders indépendants.

Cette solution n'est néanmoins pas suffisante. Supposons que la signature de la méthode `push` soit différente : `void push(Object o)`. Il est nécessaire que toutes les classes des objets transitant via la méthode `push` soient chargées dans un class loader commun à C_1 et C_2. Ces classes, appelées *classes partagées*, sont un sous-ensemble des classes d'implémentation des composants.

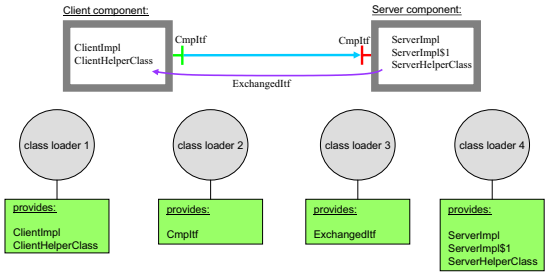

FIG. 8.3 – Organisation des class loaders.

La figure 8.3 donne un exemple d'organisation de class loaders pour une application client/-serveur faisant intervenir deux composants. L'implantation du composant client est réalisée à l'aide de deux classes (`ClientImpl` et `ClientHelperClass`) qui sont chargées par le class loader 1. Le serveur est implanté par trois classes qui sont chargées par le class loader 4. Par ailleurs, les composants communiquent via l'interface `CmpItf` qui est chargée par le class loader 2. Enfin, les composants s'échangent une classe partagée (`ExchangedItf`) qui est chargée par le class loader 3.

Reconfigurations possibles

La solution présentée dans la section précédente permet de réaliser des reconfigurations flexibles de l'implantation des composants avec un impact minimum sur les autres composants en cours d'exécution. Ce dernier point est crucial pour la maintenance sur le long terme des applications à composants. Notons que lorsqu'un composant a été mis à jour, les class loaders

qui avaient été utilisés pour charger la version remplacée du composant peuvent être supprimés de la mémoire par le ramasse-miettes Java (*garbage collector*).

Par ailleurs, notre solution permet également de mettre à jour les interfaces de composants. Néanmoins, du fait que ces interfaces sont référencées par les classes d'implantation des composants, il est nécessaire de recharger l'ensemble des composants liés via cette interface au composant mis à jour. Ce principe est représenté sur la figure 8.4 : l'interface `I` est utilisée pour lier les composants C_1 et C_2, et les composants C_3 et C_4. La mise à jour de l'interface `I` pour le composant C_4 requiert la mise à jour des implantations des composants C_3 et C_4. En revanche, elle ne nécessite pas de mise à jour des composants C_1 et C_2. Ceci est dû au fait que l'interface `J` crée une frontière de reconfiguration. Notons que, de façon similaire aux classes d'implantation, dès lors qu'une interface n'est plus utilisée, le class loader qui l'a chargée peut être supprimé de la mémoire.

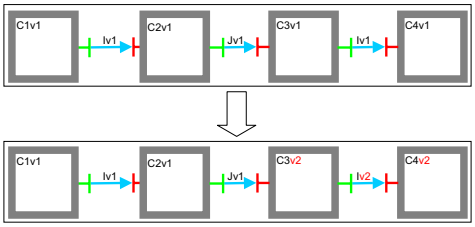

FIG. 8.4 – Reconfiguration d'interface.

8.2.2 Mise en œuvre

Nous avons effectué la mise en œuvre de la proposition formulée dans la section précédente à l'aide de *Module loader*, un système permettant de créer et gérer des organisations arbitraires de class loaders. Nous commençons par une brève description de *Module loader* ; nous décrivons ensuite son utilisation dans JULIA, puis nous montrons comment l'ADL de JULIA a été étendu pour permettre la spécification de versions d'interfaces, du code partagé, et l'instanciation des différents class loaders.

Module loader

Module loader [Hal04] est un canevas permettant de construire des organisations arbitraires de class loaders, respectant une logique de chargement définie par le développeur. Les principaux concepts de *Module loader* sont représentés sur la figure 8.5.

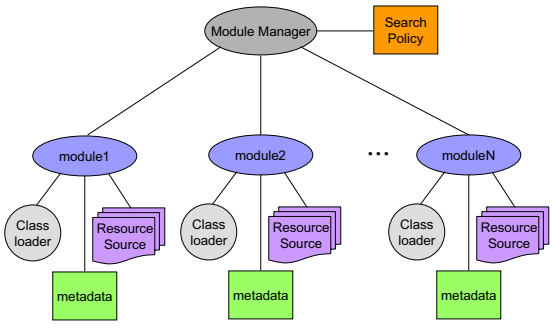

FIG. 8.5 – Architecture de Module loader.

- Le **module** est une unité logique de groupement de ressources (i.e classes). Chaque module est associé à un class loader. Par ailleurs, chaque module peut définir des méta-données.
- Un **module manager** gère un ensemble de modules. Il envoie des notifications quand des modules sont ajoutés ou retranchés.
- Une **resource source** est une source à partir de laquelle un module peut charger des classes. Cette source peut être un fichier, une URL, une base de données, etc. Les concepteurs de modules peuvent définir leurs propres sources.
- La **search policy** est le "cerveau" du mécanisme de chargement de classes. Il définit la façon dont un module peut localiser (et donc charger) une classe. La *search policy* adapte son comportement en fonction des événements qu'elle reçoit du *module manager*. Un exemple typique de *search policy* est l'*import search policy* : les modules annoncent (via des méta-données) les ressources qu'ils exportent (i.e. fournissent) aux autres modules, et les ressources qu'ils importent (i.e. requièrent) des autres modules. Un module ne peut être créé que si les ressources qu'il importe sont présentes.

Utilisation de Module loader dans Julia

L'utilisation de *Module loader* dans Julia nécessite (1) de définir les modules appropriés, et (2) de développer une *search policy*. Deux types de modules sont utilisés dans Julia. Le premier type, appelé *loading module* correspond aux class loaders présentés dans la section 8.2.1. Les *loading modules* utilisent leur class loader pour charger des classes "versionnées". Les méta-données du module sont organisées sous forme d'une table de hachage indiquant les couples (classe, version) que le module est capable de charger. Le second type de module est appelé *info module*. Un *info module* est associé à un composant ; son rôle est d'indiquer les classes (et leur version) nécessaires au composant. Le class loader d'un *info module* ne charge aucune classe : il délègue le chargement des classes au *loading module* fournissant la version de classe requise. Cela permet d'assurer que deux composants communiquant par une interface `I` la chargeront à l'aide du même class loader. La *search policy* connaît les différents *loading* et *info modules*. Elle est utilisée par les *info modules* pour trouver les *loading module* correspondant aux versions de classes recherchées.

La figure 8.6 présente l'organisation des *loading* et *info modules* dans le cadre de l'application décrite sur la figure 8.3. Chaque composant est associé à un *info module* qui délègue le chargement des classes à des *loading modules*. Les *loading modules* correspondent exactement aux quatre class loaders représentés sur la figure 8.3.

Quand un composant est créé, il est nécessaire que le class loader associé à son *info module* devienne le class loader du flot d'exécution. Il est également nécessaire que lorsque les composants communiquent, le flot d'exécution des requêtes (i.e le *thread*) soit associé aux class loaders appropriés. Par exemple, si un appel "traverse" quatre composants, il est nécessaire que lors du transit de l'appel dans chacun des composants, le class loader du flot d'exécution soit mis à jour : pour chaque composant, il doit être positionné sur l'*info module* du composant. Pour ce faire, nous avons développé un intercepteur dont le rôle est d'intercepter toutes les requêtes pour modifier le class loader du flot d'exécution. Ce mécanisme d'interception engendre un léger surcoût qui est évalué dans la section 8.2.3.

Extension de l'ADL Fractal

Les modules peuvent être créés programmatiquement. Néanmoins, c'est une tâche fastidieuse et sujette à erreurs : le développeur doit créer les *info* et *loading modules* appropriés, remplir leur méta-données ; enfin, il doit positionner le class loader adapté quand les composants sont créés.

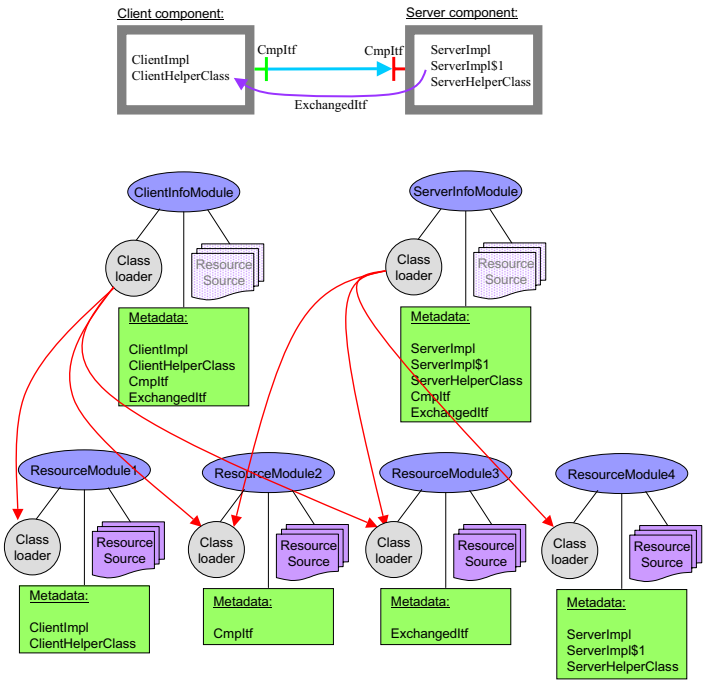

FIG. 8.6 – Organisation des modules.

Pour faciliter ce travail, nous avons développé une extension de l'ADL FRACTAL qui permet de spécifier les versions de composants et d'interfaces utilisées par l'application. Ce module permet également de spécifier les *classes partagées* de chacun des composants. La figure 8.7 donne l'ADL de l'application représentée sur la figure 8.3. L'attribut `version` a été ajouté aux définitions d'interfaces et de contenu ; il permet de spécifier la version de classe à utiliser. Par ailleurs, l'élément `file` permet de spécifier les classes requises par le composant. Enfin, l'élément `shared` est utilisé pour spécifier les classes partagées, i.e. les classes qui sont échangées entre les composants via leurs interfaces.

Nous avons également étendu l'usine ADL pour qu'elle utilise les informations de versionnement afin de créer les *loading* et *info modules* appropriés. Dans sa version actuelle, l'usine n'offre que deux granularités concernant le découpage en class loaders de l'application : un seul class loader — ce qui correspond au comportement par défaut de JULIA et ne permet pas de reconfiguration dynamique —, et la granularité définie dans les sections précédentes.

8.2.3 Performances

Nous avons effectué des tests pour évaluer la surcharge en temps d'exécution induite par notre proposition.

Le tableau 8.1 présente le temps d'exécution de certaines opérations en Java et celui nécessaire à l'interception d'un appel de méthode JULIA et à un changement de class loader. Les mesures ont été effectuées sur un pentium IV 2.2GHz exécutant une JDK1.2.2_06 sur un système d'exploitation Linux. Le tableau montre que le coût d'une interception et d'un changement de class loader est à peu près similaire à celui de la création d'un objet de la classe `String`. Il est en revanche six fois plus coûteux que l'exécution d'une méthode vide et quatre fois moins coûteux que la création d'un objet de la classe `File`.

La surcharge en temps d'exécution induite par notre solution dépend du code des composants

```
<definition name="ClientServerApp" version="1.0">
  <component name="Client">
    <interface name="c" role="client" signature="CmpItf" version="1.0"/>
    <content class="ClientImpl" version="1.0"/>
    <file name="ClientHelperClass"/>
    <shared name="ExchangedItf" version="1.0"/>
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="CmpItf" version="1.0"/>
    <content class="ServerImpl" version="2.0"/>
    <file name="ServerImpl1"/>
    <file name="ServerHelperClass"/>
    <shared name="ExchangedItf" version="1.0"/>
  </component>
  <binding client="Client.c" server="Server.s"/>
</definition>
```

FIG. 8.7 – Extension de l'ADL FRACTAL pour le chargement de code.

Opération	Temps d'exécution [ns]
Incrémentation d'un entier	9
Invocation d'une méthode vide	12
Création d'un objet de la classe `String`	76
Création d'un objet de la classe `File`	262
Interception d'un appel JULIA et changement de class loader	**71**

TAB. 8.1 – Coût de certaines opérations Java et coût de l'interception d'un appel de méthode JULIA pour effectuer un changement de class loader.

dont les appels sont interceptés. Nous avons évalué cette surcharge sur une implantation simpliste d'un serveur HTTP à environ 3%.

8.3 Validation d'architectures

Cette section présente un outil permettant la validation des architectures construites à l'aide de la bibliothèque DREAM. Cet outil repose sur l'utilisation d'un système de types adapté à partir de travaux réalisés dans le cadre du langage ML. Nous commençons par présenter ce système de types. Nous donnons ensuite un exemple d'utilisation de cet outil pour valider une pile de protocoles. Enfin, nous exposons la limitation principale de cet outil.

8.3.1 Problématique

Un intergiciel construit à l'aide de DREAM est composé d'un ensemble de composants qui s'échangent des messages. Chaque composant effectue des traitements sur les messages ; ces traitements consistent principalement en l'ajout, la suppression et la modification de chunks. Dans le système de types de base fourni par JULIA, tous les messages ont le même type : l'interface Java `Message` définie au chapitre 7. En conséquence, il n'est pas possible d'effectuer des vérifications de types portant sur les chunks qu'un message doit/peut posséder pour être traité par un composant. En effet, les seules vérifications de types possibles portent sur les types Java des interfaces utilisées par les composants pour communiquer (e.g. `Push` et `Pull`). De fait, un assemblage de

composants perçu comme correct dans le système de types de JULIA peut ne pas s'exécuter du fait que les composants reçoivent des messages ne possédant pas les chunks appropriés.

La figure 8.8 donne un exemple d'architecture erronnée : le composant `readTS` attend des messages possédant un chunk dont le nom est *ts*, tandis que le composant `addTS` attend des messages n'ayant pas de chunk dont le nom est *ts*. Les deux composants recevant exactement les mêmes messages (dupliqués par le composant `duplicator`), chaque message reçu provoquera une levée d'exception de la part d'un des deux composants.

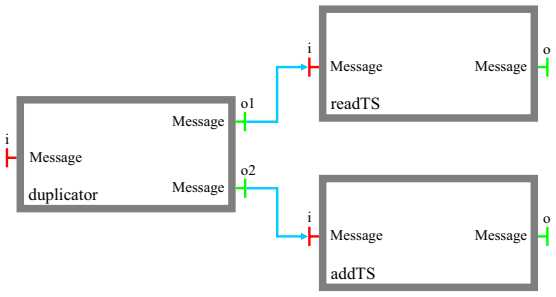

FIG. 8.8 – Exemple d'architecture incorrecte.

8.3.2 Un système de types pour le canevas DREAM

Dans cette section, nous proposons un système de types polymorphe permettant de décrire la structure des messages (en termes de chunks). Ce système de types est une adaptation de travaux qui ont été réalisés dans le cadre des *enregistrements extensibles* [Rém93a, Rém93b]. Nous décrivons ce système de types de façon intuitive en décrivant le typage des messages, le typage des composants et en montrant comment le système de types permet de détecter que l'architecture de l'exemple 8.8 est incorrecte.

Types de messages

Les *enregistrements* sont des structures de données utilisées dans plusieurs langages de programmation. Un enregistrement est un ensemble fini d'associations entre des *noms* et des *valeurs*. Dans [Rém93a, Rém93b], Rémy décrit des extensions du langage ML permettant d'effectuer les opérations de base sur les enregistrements : ajout/retrait d'une association nom-valeur et concaténation d'enregistrements. Il définit également un système de types statique qui garantit que les programmes ne produiront pas d'erreurs telles que l'accès à des noms d'enregistrements inexistants. Les messages DREAM peuvent être considérés comme des enregistrements associant des noms à des chunks. Les composants DREAM peuvent, eux, être considérés comme des fonctions polymorphes. Le polymorphisme est important pour les deux raisons suivantes : d'une part, un composant peut être utilisé dans différents contextes avec différents types. Par ailleurs, le polymorphisme autorise la mise en relation des types des interfaces clientes et serveurs d'un composant, ce qui permet des descriptions plus précises du comportement des composants.

Le type d'un message est composé d'une **liste de couples** et d'une **information supplémentaire**. Le premier élément de chaque couple est un *nom*[4] ; le second élément peut prendre les formes suivantes :

– `abs` signifie que le message ne contient pas de chunk associé à *nom*.

[4]Tous les noms de la liste doivent être distincts.

- `pre`(τ) signifie que le message contient un chunk de type[5] τ associé à *nom*. Notons que τ peut être une *variable de type*, ce qui signifie qu'elle représente un type arbitraire de chunk.
- une *variable de champ* dont la valeur peut être `abs` où `pre`(τ).

L'information supplémentaire concerne tous les noms qui ne sont pas présents dans la liste de couples. Elle peut prendre les formes suivantes :

- `abs` signifie que le type du message ne contient pas d'autres couples que ceux décrits dans la liste.
- une *variable de rangée* qui peut être `abs` ou toute liste de couples.

$$\mu_1 = \{ipc : \texttt{pre}(\texttt{IPChunk}); mc : \texttt{pre}(\texttt{MonitoringChunk}); \texttt{abs}\}$$
$$\mu_2 = \{ipc : \texttt{pre}(\texttt{IPChunk}); mc : \texttt{pre}(\texttt{MonitoringChunk}); c : \texttt{abs}; \texttt{abs}\}$$
$$\mu_3 = \{a : \texttt{pre}(X); \texttt{abs}\}$$
$$\mu_4 = \{a : Y; \texttt{abs}\}$$
$$\mu_5 = \{ipc : \texttt{pre}(\texttt{IPChunk}); Z\}$$

FIG. 8.9 – Exemples de types de messages.

La figure 8.9 donne des exemples de types de messages. Un message de type μ_1 contient exactement deux chunks de types `IPChunk` et `MonitoringChunk`, respectivement associés aux noms *ipc* et *mc*. Le type μ_2 est identique au type μ_1. Le type μ_3 utilise une *variable de type* X, ce qui signifie qu'un message de type μ_3 doit contenir un chunk dont le nom est a, mais dont le type n'est pas spécifié. Le type μ_4 fait intervenir une *variable de champ* Y qui peut valoir `abs` ou `pre`(X), X étant une variable de type. Enfin, le type μ_5 fait intervenir une *variable de rangée* Z qui représente soit `abs`, soit toute liste de couples.

Types de composants

Le type d'un composant est représenté par un type de fonction polymorphe. Nous ne typons que les interfaces d'entrée et de sortie de messages (`Pull` et `Push`). La figure 8.10 donne des exemples de composants et leurs types associés. Le composant *duplicator* a une entrée i et deux sorties o_1 et o_2 : il reçoit des messages d'un type quelconque X sur son entrée et les duplique sur ses deux sorties. Le composant add_{ipc} ajoute un chunk `IPChunk` (associé au nom *ipc*) aux messages qu'il reçoit ; ces messages ne doivent pas posséder initialement de chunk associé au nom *ipc*. Le composant $remove_{ipc}$ enlève le chunk associé au nom *ipc* si celui-ci est présent dans le message. Le composant *serializator* reçoit des messages d'un type quelconque X. Pour chaque message, il renvoie un nouveau message contenant un chunk de nom *bac*[6] qui contient la forme sérialisée du message reçu. Le type de ce chunk est créé à l'aide d'un constructeur de type, noté `ser`. Le composant *deserializator* implante le comportement dual du composant *serializator*.

Vérification de types

La figure 8.11 reprend l'exemple de la figure 8.8. Les ports des composants sont typés à l'aide du système de types décrit dans les paragraphes précédents. L'architecture représentée sur la figure est correctement typée si, et seulement si, le système d'équations suivant a une solution :

[5] Le type d'un chunk est la classe Java d'implantation du chunk.
[6] "bac" est l'acronyme de *byte array chunk*.

$$
\begin{aligned}
\mathit{duplicator} &: \forall X.\{i : \{X\}\} \rightarrow \{o_1 : \{X\}; o_2 : \{X\}\} \\
\mathit{add}_{ipc} &: \forall X.\{i : \{ipc : \texttt{abs}; X\}\} \rightarrow \{o : \{ipc : \texttt{pre}(\texttt{IPChunk}); X\}\} \\
\mathit{remove}_{ipc} &: \forall X, Y.\{i : \{ipc : Y; X\}\} \rightarrow \{o : \{ipc : \texttt{abs}; X\}\} \\
\mathit{serializator} &: \forall X.\{i : \{X\}\} \rightarrow \{o : \{bac : \texttt{ser}(\{X\}); \texttt{abs}\}\} \\
\mathit{deserializator} &: \forall X.\{i : \{bac : \texttt{ser}(\{X\}); \texttt{abs}\}\} \rightarrow \{o : \{X\}\}
\end{aligned}
$$

FIG. 8.10 – Exemples de types de composants.

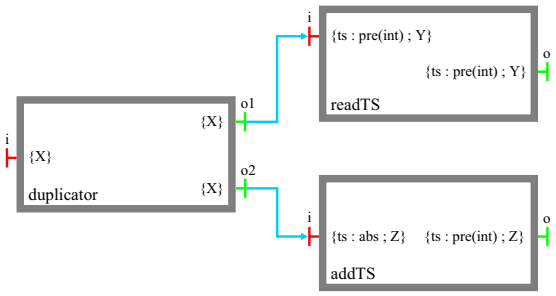

FIG. 8.11 – Typage de l'architecture représentée sur la figure 8.8.

$$
\begin{aligned}
\{X\} &= \{ts : \texttt{pre}(\texttt{TSChunk}); Y\} \\
\{X\} &= \{ts : \texttt{abs}; Z\}
\end{aligned}
$$

Ce système n'a pas de solution. En conséquense, l'architecture décrite est incorrecte.

8.3.3 Exemple d'utilisation

Le figure 8.12 (a) représente une pile de protocoles réalisée à l'aide de DREAM. Voici une description des composants de la pile de gauche[7] :

- Le composant *producer* au sommet de la pile génère des messages contenant un unique chunk de type `TestChunk` et de nom *tc*. Le type de ce composant est :

$$\mathit{producer} : \{\} \rightarrow \{o : \{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\}\}$$

- Le composant *serializer* retourne des messages possédant un chunk unique *sc* qui est la forme sérialisée des messages reçus sur l'entrée *i*. Son type est :

$$\mathit{serializer} : \forall X.\{i : \{X\}\} \rightarrow \{o : \{sc : \texttt{ser}(\{X\}); \texttt{abs}\}\}$$

- Le composant *addIP* rajoute un chunk de type `IPChunk` de nom *ipc* aux messages qui ne contiennent pas un tel chunk. Son type est :

$$\mathit{addIP} : \forall X.\{i : \{ipc : \texttt{abs}; X\}\} \rightarrow \{o : \{ipc : \texttt{preIPChunk}; X\}\}$$

- Le composant *ChannelOut* envoie les messages sur le réseau. Il attend des messages possédant au moins un chunk nommé *ipc* de type `IPChunk`.

[7] La pile de droite effectue les actions symétriques.

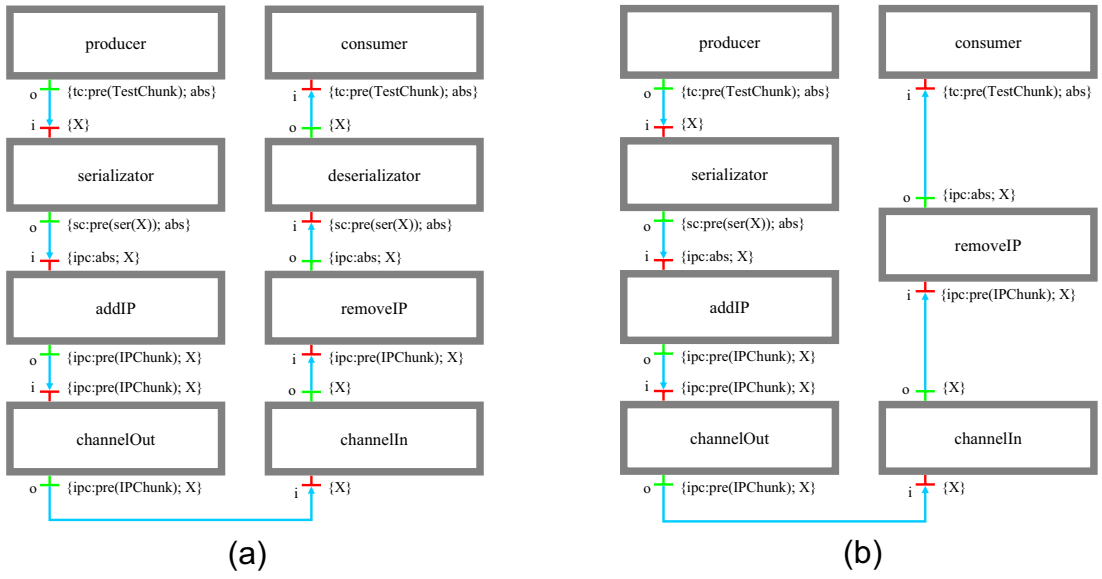

FIG. 8.12 – Un exemple de pile de protocoles.

La figure 8.12 (b) présente une architecture incorrecte : le composant *deserializer* est manquant. Nous déduisons des connexions entre composants les équations suivantes :

$$\{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} = \{U\} \tag{8.1}$$

$$\{sc : \texttt{pre}(\texttt{ser}U); \texttt{abs}\} = \{ipc : \texttt{abs}; Z\} \tag{8.2}$$

$$\{ipc : \texttt{pre}(\texttt{IPChunk}); T\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} \tag{8.3}$$

$$\{ipc : \texttt{pre}(\texttt{IPChunk}); Z\} = \{Y\} \tag{8.4}$$

$$\{Y\} = \{ipc : \texttt{pre}(\texttt{IPChunk}); X\} \tag{8.5}$$

$$\{ipc : \texttt{abs}; X\} = \{tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}\} \tag{8.6}$$

De 8.6, nous déduisons :

$$X = \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

Puis, à partir de 8.5, nous avons :

$$Y = ipc : \texttt{pre}(\texttt{IPChunk}); tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

.

Il découle alors de 8.4 et 8.3 que :

$$T = Z = tc : \texttt{pre}(\texttt{TestChunk}); \texttt{abs}$$

Par ailleurs, nous déduisons de 8.2 que :

$$Z = sc : \texttt{pre}(\texttt{ser}U); \texttt{abs}$$

Les deux dernières équations sont contradictoires. En conséquence, l'architecture décrite est incorrecte.

8.3.4 Limitation

Le système de types présenté a une limitation : il est trop restrictif pour typer les composants DREAM qui exhibent différents comportements en fonction de la présence d'un certain chunk dans un message. Considérons par exemple un composant `routeur` qui route les messages reçus sur son entrée, sur deux sorties différentes en fonction de la présence d'un chunk de nom a dans les messages. Nous voudrions pouvoir lui attribuer un type de cette forme :

$$\begin{aligned}\texttt{route} : \forall X.\{i : \{a : \texttt{pre}(\texttt{A}); X\}\} \rightarrow \{o_1 : \{X\}; o_2 : \{\texttt{abs}\}\} \\ \wedge \{i : \{a : \texttt{abs}; X\}\} \rightarrow \{o_1 : \{\texttt{abs}\}; o_2 : \{X\}\}\end{aligned}$$

Ce type utilise un opérateur $\wedge$ pour spécifier les deux comportements possibles du composant. Or le système de types présenté ne possède pas un tel opérateur. Nous travaillons actuellement sur une extension du système de types pour prendre en compte cet opérateur.

8.4 Conclusion

Dans ce chapitre, nous avons présenté les outils de gestion de configuration fournis par le canevas DREAM. Ces outils permettent de prendre en charge deux étapes du cycle de vie des composants : leur vérification et leur reconfiguration.

Concernant les reconfigurations, nous avons décrit deux outils : le premier permet d'effectuer des reconfigurations de structure (i.e. ajout de composants et de liaisons, etc.) ; le second prend en charge les reconfigurations d'implantation (i.e. modification du code des composants en cours d'exécution). Concernant les vérifications de structure à composants, nous avons présenté un outil dont le rôle est d'effectuer des vérifications de type sur les assemblages décrits dans l'ADL. Cet outil permet de vérifier, avant le déploiement, que chaque composant recevra des messages ayant la structure attendue.

Remerciements : l'outil de reconfiguration de structure a été réalisé en collaboration avec Matthieu Leclercq, ingénieur expert au sein du projet SARDES. L'outil de reconfiguration d'implantation a été réalisé en collaboration avec Jakub Kornaś, lors de son stage de Master au sein du projet SARDES. Jakub était alors étudiant à l'Université des Sciences et Technologies de Cracovie (AGH). Enfin, l'outil de verification de types a été réalisé avec Philippe Bidinger et Alan Schmitt, respectivement doctorant et chargé de recherches au sein du projet SARDES.

Chapitre 9

Evaluation

Sommaire

Ce chapitre présente une évaluation du canevas DREAM à travers deux expériences que nous avons réalisées : une implantation du canevas SEDA et la ré-ingénierie de la plate-forme ScalAgent. SEDA est un canevas dédié à la construction de services Internet ; la plate-forme ScalAgent est un intergiciel de communication fournissant un modèle d'exécution événement/réaction. Pour chacune des expériences, nous présentons une évaluation des performances et des bénéfices apportés par l'utilisation de DREAM.

9.1 Implémentation du canevas SEDA

9.1.1 Présentation de SEDA

SEDA (*Staged Event-Driven Architecture*) [WCB01] est un canevas logiciel dédié à la construction de services Internet. Le but de SEDA est de construire des services qui supportent des pics de charges. Pour ce faire, les services sont construits comme des réseaux d'étages de traitement d'événements. Comme nous l'avons représenté sur la figure 9.1, un étage est constitué d'une file de messages entrants, d'un traitant d'événements et d'un pool de threads. Chaque thread a

un rôle similaire : il prélève un groupe de n messages de la file d'entrée et invoque le traitant d'événements avec les messages prélevés. Le traitant d'événements peut produire des messages à destination des files de messages d'autres étages.

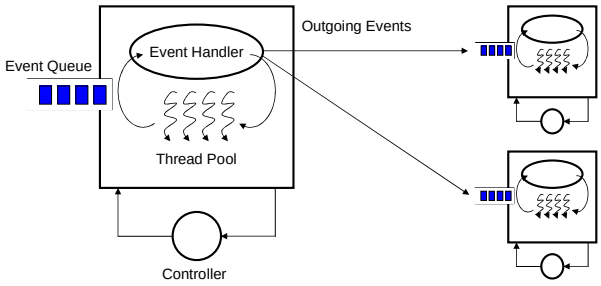

FIG. 9.1 – Architecture d'un étage SEDA.

L'une des originalités de SEDA est d'utiliser des mécanismes de contrôle de ressources afin de garantir que chaque étage reste dans un certain régime opératoire. Ces mécanismes sont mis en œuvre par deux contrôleurs associés à chaque étage :

- le *contrôleur de pool de threads* ajuste le nombre de threads contenus dans le pool. Il observe périodiquement le nombre de messages dans la file d'entrée de l'étage et ajoute un nouveau thread à chaque fois que ce nombre augmente d'une valeur δ déterminée, et ce, jusqu'à ce que le nombre maximal de threads par étage soit atteint.
- le *contrôleur de prélèvement des messages* ajuste le nombre de messages qui sont prélevés dans la file d'entrée par chacun des threads du pool. Le but de ce contrôleur est d'obtenir le meilleur compromis possible entre une augmentation du débit (mesuré en événements par seconde) — du fait que des traitements agrégés sur les messages peuvent être réalisés — et une augmentation du temps de réponse.

9.1.2 Mise en œuvre de SEDA à l'aide de DREAM

Nous avons réalisé une implantation du canevas SEDA à l'aide de DREAM. L'architecture d'un étage est représentée sur la figure 9.2[1]. Un étage est implanté par un composant composite qui encapsule trois composants : une file de messages FIFO qui stocke les messages destinés à l'étage, une pompe qui utilise son entrée `Pull` pour prélever des groupes de messages de la file et qui fournit ces messages à un composant de traitement des messages. La pompe est le seul composant actif de l'étage[2]. La tâche définie par la pompe est exécutée par un pool de threads hébergé par le gestionnaire d'activités (comme décrit dans le chapitre 7). Les contrôleurs de pool de threads et de prélèvement des messages sont implantés par des intercepteurs (représentés par des cercles rouges sur la figure). Le contrôleur de pool de threads intercepte les messages entrant dans la file de messages. Il est lié au gestionnaire d'activités afin de pouvoir modifier le nombre de threads exécutant la tâche de la pompe. Le contrôleur de prélèvement des messages intercepte les messages sortant de l'étage afin d'effectuer des mesures de débit, et modifie en conséquence le nombre de messages prélevés par chacun des threads de la pompe. Notons que les groupes de messages prélevés par la pompe sont encapsulés dans des messages agrégés.

[1] Les différentes flèches représentent des liaisons FRACTAL. Elles sont distinguées sur la figure par souci de clarté.

[2] Il aurait été possible d'architecturer un étage sans pompe dans lequel le composant actif aurait été le traitant d'événements. Néanmoins, cela empêchait de réaliser certaines des expériences décrites dans les sections suivantes.

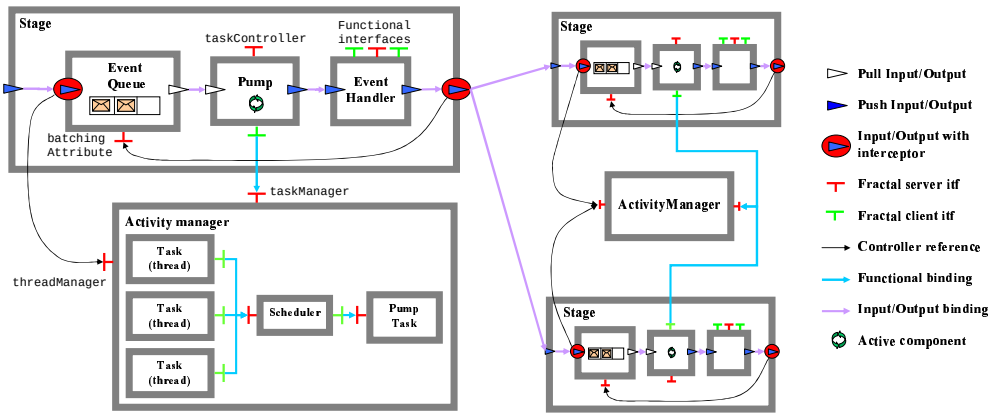

FIG. 9.2 – Architecture d'un étage SEDA réalisée avec DREAM.

9.1.3 Gain en configurabilité

Le premier bénéfice apporté par l'utilisation de DREAM est qu'un traitant d'événements peut avoir un contrôle direct sur les threads qui l'exécutent. Pour ce faire, il suffit de le lier au gestionnaire d'activités qui héberge la tâche définie par la pompe. Dans l'implantation SEDA, un étage n'a aucun moyen de connaître et de contrôler le nombre de threads qui l'exécutent. Disposer d'une telle capacité permet d'améliorer les performances : un traitant d'événements peut éviter l'utilisation de primitives de synchronisation, s'il sait qu'un seul thread est en train de l'exécuter. Il peut même interagir avec le gestionnaire d'activité pour exiger d'être exécuté par un seul thread pendant une certaine période de temps.

Un autre bénéfice apporté par l'implantation DREAM est que plusieurs étages peuvent partager le même gestionnaire d'activités. De fait, il est aisé d'appliquer une politique d'ordonnancement entre différents étages. Ceci peut être utilisé, par exemple, pour contrôler l'accès à de la mémoire partagée. Dans l'implantation existante de SEDA, les étages doivent faire un usage explicite des primitives de synchronisation (mutex, sémaphores, etc.). Dans l'implantation DREAM, il est possible d'utiliser un ordonnanceur qui garantit que deux étages s'exécuteront toujours en exclusion mutuelle. Un étage peut également affecter la politique d'ordonnancement d'un autre étage.

Un troisième bénéfice de l'implantation DREAM est que chaque élément d'un étage (file d'entrée, pompe, traitant d'événements) est implanté par un composant FRACTAL. En conséquence, il est possible de connaître, à l'exécution, la structure des différents étages et les interconnexions entre étages. Ainsi, si les étages ont été déployés avec les contrôleurs appropriés, il est possible de reconfigurer dynamiquement le réseau d'étages. Par ailleurs, contrairement à l'implantation existante de SEDA qui impose l'utilisation d'une file de messages FIFO pour tous les étages, il est aisé, à l'aide de l'implantation DREAM, d'utiliser d'autres types de files. Cela permet d'ordonnancer les messages reçus par un traitant d'événements, ce qui n'est pas possible avec SEDA à moins de le coder explicitement dans le code du traitant d'événements.

Enfin, l'implantation DREAM permet de modifier (dynamiquement) l'organisation des flots d'exécution au sein du réseau d'étages. Dans SEDA, chaque étage possède son propre flot d'exécution, ce qui découple son exécution des autres étages. Comme il est montré expérimentalement dans la section suivante, il peut être intéressant de changer ce comportement, en cours d'exécution, pour réduire la latence induite par les changements de contexte liés aux changements de threads entre chacun des étages. A l'aide de DREAM, il est trivial de changer une interaction asynchrone entre deux étages en une interaction synchrone : il suffit d'enlever la file d'entrée et

la pompe. L'interface d'entrée de l'étage est directement liée à l'interface d'entrée du traitant d'événements. En conséquence, ce dernier s'exécute dans le flot d'exécution des étages produisant des événements dont il est destinataire.

Notons que les deux premières améliorations décrites dans cette section étaient mentionnées comme nécessitant des travaux futurs dans [Wel02].

9.1.4 Mesures de performances

Nous avons effectué des mesures de performances pour comparer l'implantation de SEDA réalisée à l'aide de DREAM à l'implantation existante. Le premier test que nous avons réalisé consiste en un réseau de trois étages organisés en anneau. Chaque étage transmet les événements qu'il reçoit à son successeur dans l'anneau. Nous mesurons le temps moyen nécessaire à un message pour parcourir l'anneau. Chaque étage est exécuté par un pool contenant un seul thread. Les expériences ont été réalisées sur un Athlon XP 1.5GHz avec 512Mo de mémoire, exécutant une JDK 1.4 sur linux 2.4.20.

Intergiciel	Temps moyen (ms)
SEDA	0.415
DREAM (non-reconf.)	0.410
DREAM (reconf.)	0.420

TAB. 9.1 – Performances de l'implantation de SEDA effectuée avec DREAM.

Le tableau 9.1 montre le temps moyen nécessaire à un message pour parcourir l'anneau. Nous comparons deux implantations réalisées avec DREAM à l'implantation existante. La première implantation réalisée avec DREAM n'est pas dynamiquement reconfigurable. La seconde l'est : chaque composant possède des contrôleurs et des intercepteurs permettant de l'arrêter de façon fiable. Comme nous pouvons le voir, le temps moyen nécessaire à un message pour parcourir l'anneau est sensiblement le même pour les trois implantations. Il est néanmoins légèrement supérieur dans la version reconfigurable ($\approx 2.5\%$), ce qui s'explique par les interceptions réalisées.

Une autre expérimentation que nous avons réalisée est de comparer les performances d'Haboob[3], un serveur HTTP implanté à l'aide du canevas SEDA. Le serveur est mis en œuvre par 10 étages responsables de la transmission des messages (à l'aide de sockets asynchrones), des entrées/sorties disque, du protocole HTTP (réception, mise en mémoire cache des données, parsing, émission, etc.).

Intergiciel	Temps moyen(ms)
SEDA	1,31
DREAM (non reconf.) 1 thread par étage	1,34
DREAM (reconf.) 1 thread par étage	1,38
DREAM (non reconf.) 1 thread par serveur	1,19
DREAM (reconf.) 1 thread par serveur	1,23

TAB. 9.2 – Temps moyen de traitement d'une requête HTTP.

Le tableau 9.2 montre le temps moyen nécessaire au traitement d'une requête HTTP. Notons que les expériences que nous présentons ont été réalisées dans le but de comparer les performances

[3]Une description détaillée d'Haboob peut être trouvée dans [WCB01].

des deux implantations, et non dans le but d'évaluer les contrôleurs de gestion de ressources présents dans SEDA. En conséquence, nous n'essayons pas de surcharger le serveur. Néanmoins, nous avons réalisé d'autres expériences qui nous ont permis de vérifier que les contrôleurs de gestion de ressources se comportaient de la même façon dans les implantations SEDA et DREAM. Les lecteurs intéressés par le comportement d'Haboob en forte charge sont invités à lire [WCB01, Wel02].

Nous comparons l'implantation existante de Haboob avec deux implantations réalisées à l'aide de DREAM. La première associe un pool de threads par étage ; ce pool contient un seul thread. Les performances de SEDA sont légèrement meilleures ($\approx$ 3 à 5%). Cela s'explique par le fait que l'architecture de Haboob a été modifiée : dans l'implantation existante, des références vers les étages sont propagées en même temps que les événements. Ceci, empêchant toute reconfiguration dynamique ultérieure, est contraire au principe d'architecture de DREAM. En conséquence, des liaisons supplémentaires entre étages sont nécessaires, ce qui explique la légère perte de performances.

L'autre expérience que nous avons réalisée consiste à modifier l'organisation des flots d'exécution au sein du serveur HTTP. Comme nous l'avons expliqué dans la section précédente, il est aisé, à l'aide de DREAM, de changer une interaction asynchrone entre deux étages en une interaction synchrone. Nous avons ôté les files d'entrée et les pompes des différents étages ; il en résulte une architecture dans laquelle il n'y a qu'un thread qui exécute l'ensemble des étages (le thread qui reçoit les requêtes sur le réseau). Le tableau 9.2 montre que les performances de cette version mono-threadée sont meilleures ($\approx$ 12%) que celles de la version avec un thread par étage. Notons, par ailleurs, que le gain est le même pour les versions reconfigurables et non reconfigurables. Ce gain s'explique facilement par le fait que le serveur n'étant pas en forte charge, un seul thread par requête est nécessaire à tout moment. Ce résultat montre qu'il est intéressant de pouvoir changer dynamiquement l'organisation des flots d'exécution au sein d'un réseau d'étages.

9.2 Ré-ingénierie de JORAM

Cette section présente la ré-ingénierie de JORAM [jor05a] effectuée à l'aide de DREAM. JORAM s'exécute sur la plate-forme ScalAgent, un intergiciel permettant de construire des applications à base d'*agents* s'exécutant suivant un modèle "événement → réaction". Nous montrons que l'implantation DREAM de la plate-forme ScalAgent permet d'obtenir un gain en configurabilité (dynamique) important — et ce, sans perte de performances.

9.2.1 Une brève présentation de la plate-forme ScalAgent

Le modèle d'agents

La plate-forme ScalAgent [QBB+04] permet le déploiement et l'exécution d'agents. Les agents sont des objets réactifs qui se comportent conformément au modèle "événement → réaction" [Agh86]. Un événement est une transition d'état à laquelle un ou plusieurs agents vont réagir. Il est représenté par un message typé, appelé notification. L'envoi de notifications représente le seul moyen de communication et de synchronisation entre agents. Les agents sont persistants : l'état d'un agent n'est pas perdu lorsque la machine sur laquelle il s'exécute s'arrête ou tombe en panne. Par ailleurs, la réaction des agents est atomique : cette propriété assure qu'une réaction est soit complètement exécutée, soit annulée.

L'infrastructure d'exécution

La création, l'exécution et les communications des agents sont prises en charge par un intergiciel de communication. Celui-ci est constitué d'un ensemble de serveurs d'agents organisés en bus. Comme on peut le voir sur la figure 9.3, chaque serveur d'agents est constitué de trois éléments architecturaux : l'*engine* est en charge de la création et de l'exécution des agents. Il effectue, en boucle, un ensemble d'instructions consistant à prendre un message dans le *conduit* et faire réagir l'agent destinataire. Il garantit la persistance des agents et l'atomicité de leurs réactions. Le conduit route les messages de l'engine vers les *networks*. Les networks assurent la transmission fiable et causalement ordonnée des messages.

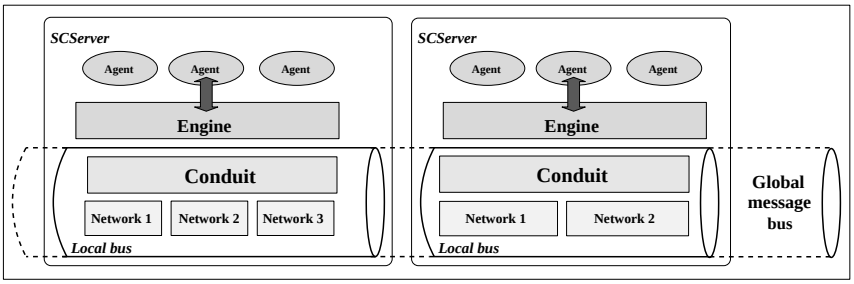

FIG. 9.3 – Deux serveurs d'agents interconnectés.

9.2.2 Ré-ingénierie de l'intergiciel ScalAgent avec DREAM

Nous avons implémenté l'intergiciel ScalAgent à l'aide de DREAM (figure 9.4). Ses principales structures (networks, engine et conduit) ont été préservées de manière à faciliter la comparaison. L'engine est un composite qui comprend deux parties. La première partie traite des messages DREAM. Elles est constituée d'une file de messages — qui stocke les messages entrants — et d'un composite (`AtomicityProtocol`) qui garantit l'atomicité de la réaction des agents. Le routeur qu'il encapsule route les messages en fonction de l'identifiant de la transaction qui les a produits. La seconde partie correspond au composite `Repository`. Celui-ci est en charge de la création et de l'exécution des agents. Les composants `NotifToMessage` et `MessageToNotif` permettent de transformer les notifications échangées par les agents en messages DREAM et vice versa.

Par ailleurs, nous avons représenté deux networks typiques. Les deux sont des composites qui encapsulent un canal entrant (`TCPChannelIn`), un canal sortant (`TCPChannelOut`), et un transformateur (`DestinationResolver`) en charge de mettre l'adresse IP et le numéro de port du destinataire du message. Le network 1 encapsule deux composants supplémentaires : le `CausalSorter` garantit l'ordonnancement causal des messages échangés. La file de messages permet de découpler les flots d'exécution de l'engine et du network.

Le conduit est implémenté par un routeur. La logique de routage est basée sur l'identifiant de l'engine auquel le message est destiné.

9.2.3 Gain en configurabilité

L'implémentation DREAM de la plate-forme ScalAgent apporte de nombreux bénéfices que nous listons ci-dessous.

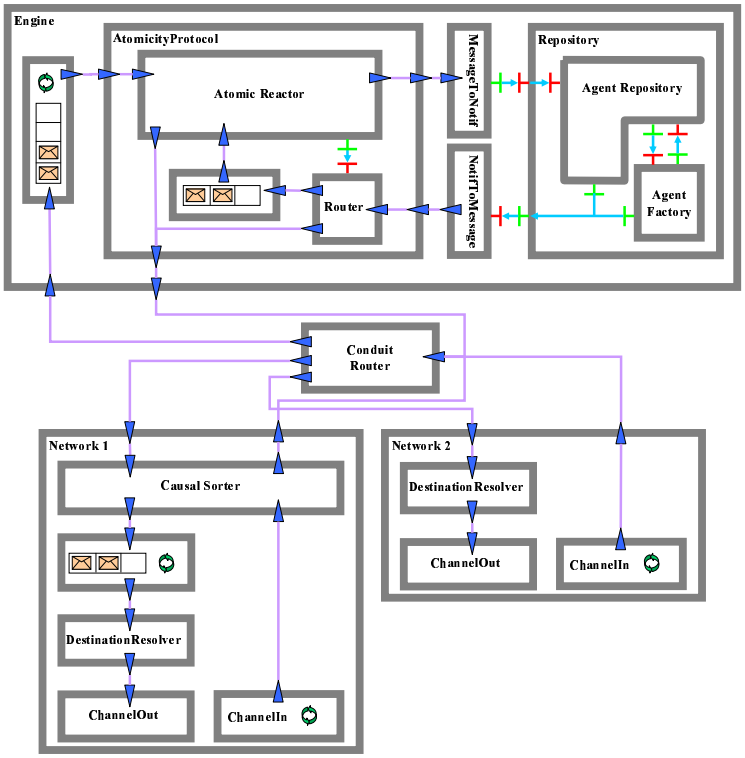

FIG. 9.4 – Architecture DREAM d'un serveur d'agents.

Modification des propriétés non fonctionnelles

Un des apports de l'implémentation DREAM est qu'il est aisé de changer les propriétés non fonctionnelles fournies par l'intergiciel de communication. Par exemple, supprimer l'atomicité des réactions des agents ne requiert qu'une modification de la description ADL de l'intergiciel. Cela peut également être fait programmatiquement en cours d'exécution. En revanche, pour enlever cette propriété de l'intergiciel ScalAgent, il est nécessaire d'en modifier le code source.

Parallélisation de l'exécution des agents

Le fait que le conduit soit implémenté par un routeur permet de faire coexister plusieurs engines dans le même serveur d'agents. Cette particularité a deux avantages : d'une part elle permet la parallélisation de l'exécution des agents (au sein d'un serveur d'agents, les exécutions des agents sont sérialisées). D'autre part, elle permet de fournir différentes propriétés non fonctionnelles à des agents s'exécutant dans le même serveur d'agents. Il est, par exemple, possible de faire coexister des agents persistants, et d'autres non persistants.

Modification du nombre de composants actifs

Un autre bénéfice de l'implémentation DREAM réside dans la possibilité de changer le nombre de composants actifs s'exécutant dans un serveur d'agents. L'architecture que nous avons présentée sur la figure 9.4 fait intervenir trois composants actifs[4] pour un serveur d'agents avec un seul network. Cette architecture reflète l'implémentation originale de la plate-forme qui uti-

[4]Les composants actifs sont signalés par une double flèche verte.

lise trois threads différents. A l'aide de DREAM, il est possible d'obtenir une implémentation mono-threadée en supprimant les files de messages de l'engine et du network.

Adaptation de la plate-forme à des environnements contraints

Une dernière expérimentation que nous avons effectuée, a été de construire un serveur d'agents dédié aux équipements mobiles avec ressources limitées. Ces équipements présentent deux caractéristiques : ils sont sujets à des déconnexions temporaires, et ils ont une capacité mémoire limitée. La figure 9.5 montre une nouvelle architecture du serveur d'agents qui prend en compte ces deux caractéristiques.

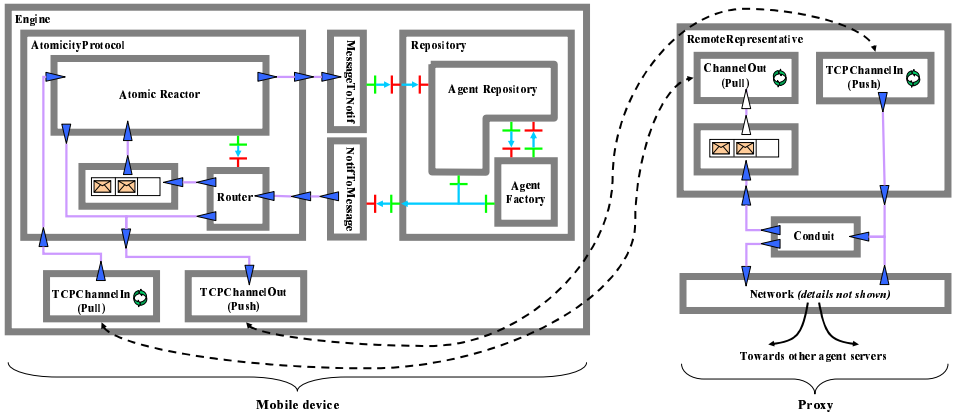

FIG. 9.5 – Architecture d'un serveur d'agents pour équipements aux ressources limitées.

Le serveur d'agents (sur la partie gauche de la figure) est un composant composite. Ce composite est un engine dans lequel la file de messages a été remplacée par un canal entrant, et dont la sortie est un canal sortant. Les messages destinés au serveur d'agents sont stockés sur un autre équipement (sur la partie droite de la figure). Cet équipement fait office de proxy pour l'équipement mobile. Il héberge un serveur d'agents avec un engine particulier qui a deux fonctions : (1) il stocke les messages à destination de l'équipement mobile ; ces messages sont ensuite récupérés par l'équipement (canal entrant en mode pull). (2) il est en charge de retransmettre les messages émis par l'équipement mobile.

Cette architecture préserve les fonctions et les caractéristiques de la plate-forme à agents tout en économisant les ressources : le serveur d'agents est mono-threadé ; les messages sont récupérés en mode pull, il n'y a pas de `CausalSorter`, ni de `DestinationResolver`. De plus, elle permet à l'équipement mobile d'être temporairement déconnecté car les messages sont stockés sur un proxy.

9.2.4 Comparaison des performances

Des mesures de performances ont été effectuées pour comparer l'efficacité de la même application s'exécutant sur la plate-forme ScalAgent et sur son implémentation à l'aide de DREAM. L'application testée fait intervenir quatre serveurs d'agents. Chaque serveur héberge un agent. Les agents sont organisés en anneau virtuel. Ils échangent des messages le long de cet anneau. Chaque agent se comporte de la façon suivante : il retransmet les messages qu'il reçoit à son successeur sur l'anneau. Nous avons effectué deux séries de tests : des messages vides et des messages d'une taille de 1 Ko. Les expériences ont été réalisées sur quatre PC dual-Xeon 1.8

GHz avec 1Go de mémoire, et connectés par un réseau Ethernet Gigabit. Par ailleurs, chacun des PC fonctionnait sous Linux 2.4.20.

Intergiciel	Nombre de tours		Empreinte mémoire
	0kB	1kB	(kB)
ScalAgent	325	255	4×1447
Dream (non-reconfigurable)	329	260	4×1580
Dream (reconfigurable)	318	250	4×1587

TAB. 9.3 – Performances de l'implémentation ScalAgent vs. DREAM.

Le tableau 9.3 montre le nombre moyen de tours d'anneau effectués par chaque message en une seconde, ainsi que l'empreinte mémoire de chaque implémentation. Nous avons comparé l'implémentation ScalAgent avec deux implémentations réalisées avec DREAM.

La première implémentation réalisée avec DREAM n'est pas dynamiquement reconfigurable. Comme nous pouvons le voir, le nombre de tours est identique à celui obtenu avec l'implémentation ScalAgent. En ce qui concerne l'empreinte mémoire, l'implémentation DREAM requiert 9% de mémoire en plus, ce qui peut s'expliquer par la taille des structures requises par FRACTAL (70 Ko), et le fait que chaque composant a plusieurs contrôleurs. Notons que cette surcharge mémoire n'est pas significative pour des PC modernes.

La seconde implémentation réalisée avec DREAM est dynamiquement reconfigurable. En particulier, chaque composant possède des contrôleurs et des intercepteurs permettant de l'arrêter de façon fiable. Cette implémentation est légèrement plus lente que la version non reconfigurable, et ne requiert que 7 Ko de plus que la version non reconfigurable. Notons que les performances de DREAM sont proportionnellement meilleures avec les messages de 1 Ko qu'avec ceux de 0 Ko. Ceci peut s'expliquer par le fait que, dans le premier cas, plus de temps est passé dans les transmissions des messages, ce qui réduit l'impact du coût des intercepteurs.

Intergiciel	Nombre de tours		Empreinte mémoire
	0 kB	1 kB	(kB)
ScalAgent	182	150	4×1447
Dream (4 serveurs d'agents)	188	153	4×1580
Dream (2 serveurs d'agents)	222	181	2×1687
Dream (1 serveur d'agents)	6597	6445	1×1900

TAB. 9.4 – Impact du nombre d'engines par serveur d'agents.

Nous avons également évalué le gain apporté par la possibilité de mettre plusieurs engines dans un même serveur d'agents. Nous avons comparé quatre architectures : l'architecture ScalAgent, une configuration DREAM équivalente (un seul engine par serveur d'agents), une configuration DREAM avec deux serveurs d'agents qui contiennent deux engines chacun, et une configuration DREAM avec un seul serveur d'agents qui contient quatre engines. Contrairement à l'expérimentation précédente, tous les serveurs d'agents sont hébergés sur le même PC. Le tableau 9.4 montre que l'architecture faisant intervenir deux engines par serveur d'agents est 18% plus performante que l'architecture traditionnelle, et réduit l'empreinte mémoire de 47%. L'amélioration des performances peut s'expliquer par le fait que la taille des horloges matricielles ajoutée sur chaque message est n^2, n étant le nombre de serveurs d'agents. De fait, limiter le nombre de

serveurs d'agents réduit la taille des matrices envoyées. Par ailleurs, on peut constater que la configuration faisant intervenir quatre engines dans un seul serveur d'agents est 29 fois plus rapide que la configuration traditionnelle. Ce résultat qui peut paraître surprenant s'explique par le fait que les communications entre agents ne transitent plus par les composants network. En effet le routeur les route directement vers l'engine approprié.

Intergiciel	Nombre de tours		Empreinte mémoire (kB)
	0 kB	1 kB	
Dream (3 threads)	329	260	4×1580
Dream (2 threads)	346	268	4×1516
Dream (1 thread)	370	279	4×1452

TAB. 9.5 – Impact du nombre de composants actifs.

Le tableau 9.5 présente les résultats d'une expérience visant à mesurer l'impact du nombre de composants actifs. Nous comparons trois architectures construites à l'aide de DREAM qui diffèrent par le nombre de composants actifs qu'elles encapsulent. L'architecture faisant intervenir deux threads a été obtenue en supprimant la file de messages du network. L'architecture mono-threadée a été réalisée en enlevant également la file de messages de l'engine. L'application testée est la même que dans la première expérience. Nous pouvons constater que, dans ce cas particulier, la réduction du nombre de composants actifs améliore les performances (+ 3 à 5 % pour l'architecture avec deux threads, et +7 à 12 % pour l'architecture mono-threadée). Ces résultats peuvent être expliqués par le fait que les agents sont organisés en anneau. En conséquence, chaque serveur ne possède qu'un message à un instant donné ; un seul thread est donc nécessaire. Notons que la diminution du nombre de composants actifs engendre également une diminution de l'empreinte mémoire. Ceci est dû au fait que les serveurs d'agents encapsulent moins de composants, et que la JVM alloue moins de mémoire pour les threads.

9.3 Conclusion

Dans ce chapitre, nous avons présenté deux expérimentations réalisées avec DREAM : une implantation du canevas SEDA pour la construction de services Internet hautes performances et une ré-ingénierie de la plate-forme ScalAgent.

Ces deux expérimentations nous ont permis de montrer :

- que DREAM permet de construire diverses personnalités d'intergiciels implantant différents paradigmes de communication.
- qu'il est possible d'associer différentes propriétés non fonctionnelles à chacune des personnalités et qu'il est possible d'adapter leurs capacités d'administration aux besoins du développeur. Ainsi est-il aussi bien possible de créer des architectures non reconfigurables ayant d'excellentes performances, que des architectures totalement reconfigurables et légèrement moins performantes.
- enfin, que l'utilisation de DREAM facilite l'adaptation d'une personnalité à divers équipements. Nous avons notamment illustré cette capacité par la description d'une version de la personnalité JORAM pour équipements mobiles aux ressources restreintes.

Remerciements : les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Matthieu Leclercq, ingénieur expert au sein du projet SARDES.

Troisième partie

Élements d'infrastructures logicielles pour l'administration autonome de systèmes

Chapitre 10

LeWYS : un canevas logiciel pour la construction de systèmes d'observation

Sommaire

Comme nous l'avons présenté dans l'introduction, la construction de systèmes autonomes nécessite la mise en place de boucles de commande composées de composants de supervision dont le rôle est d'observer l'état de fonctionnement du système, de composants d'analyse, responsables des décisions de reconfiguration, et de composants de reconfiguration qui opèrent les changements [BBC+04].

Dans ce chapitre, nous exposons les travaux que nous avons effectués concernant l'observation de systèmes distribués. Nous présentons LeWYS, un canevas logiciel à composants permettant de construire des systèmes d'observation [CELQ04, CELQ05, CLQ05]. Nous commençons ce chapitre par l'exposé des motivations à l'origine de la construction d'un tel canevas. Nous détaillons ensuite les divers éléments du canevas. Enfin, nous procédons à son évaluation.

10.1 Motivations

Les développeurs de systèmes autonomes ont besoin de fonctions d'observation permettant de connaître l'état de fonctionnement d'un système en cours d'exécution. L'état de fonctionnement du système regroupe l'état des ressources du système (consommation CPU, état du réseau, consommation mémoire, etc) et l'état de l'application. Actuellement, les développeurs de systèmes autonomes sont souvent contraints de développer des outils de supervision ad hoc pour collecter ces informations. En effet, chaque système a ses propres besoins et il existe rarement une application de supervision qui réponde à ces besoins. Il manque un canevas générique permettant de construire des applications de supervision adaptées aux différents besoins. Pour que ce canevas soit utile, il doit avoir les caractéristiques suivantes :

- *adaptation à différentes échelles de systèmes :* des systèmes centralisés à des systèmes distribués à grande échelle.
- *possibilité de remonter divers indicateurs :* outre les indicateurs "standards" (i.e. CPU, mémoire, réseau, disques, etc.), une application de supervision doit laisser la possibilité à l'administrateur de développer de façon simple ses propres collecteurs.
- *possibilité de construire des chaînes arbitrairement complexes de traitement et de propagation des indicateurs collectés :* il est nécessaire de pouvoir filtrer l'information, composer plusieurs événements, corréler certains indicateurs, etc. Par ailleurs, il est indispensable de pouvoir insérer ces traitements à divers endroits en fonction de l'architecture du système observé et du traitement effectué : par exemple, le filtrage des données s'effectue au plus près de la source, alors que la corrélation multi-sites peut se faire sur différents noeuds.
- *possibilité de configurer dynamiquement les divers paramètres de l'application :* il doit être possible de changer les indicateurs collectés, les traitements effectués sur ces indicateurs, les sites sur lesquels ils sont acheminés, etc.
- *non-intrusivité :* il est impératif de minimiser l'impact de l'application de supervision sur le système observé.

Dans ce chapitre, nous présentons LeWYS[1], un canevas logiciel à composants pour la construction de systèmes de supervision qui répond aux besoins précédents. LeWYS fournit une bibliothèque de composants Fractal qui peuvent être assemblés pour construire des applications de supervision adaptées aux besoins de l'administrateur. Le traitement et l'acheminement des données sont effectués par des canaux à événements dynamiques construits à l'aide de Dream. LeWYS permet de construire des applications de supervision aux caractéristiques très variées : d'une simple console locale affichant périodiquement la consommation CPU, à des systèmes de supervision de Grille utilisant un intergiciel à publications/abonnements pour le transfert des données.

Le reste de ce chapitre est structuré de la façon suivante : nous commençons tout d'abord par une présentation générale du canevas LeWYS dans la section 10.2. Puis les sections 10.3 et 10.4 sont consacrées à la description des composants du canevas. Une évaluation tant qualitative que quantitative est effectuée dans la section 10.5. Enfin, la section 10.6 présente les travaux connexes, avant notre conclusion qui figure en section 10.7.

10.2 Présentation générale

LeWYS est un canevas à composants dédié à la construction de systèmes d'observations. Un exemple typique de système développé avec LeWYS est représenté sur la figure 10.1. L'applica-

[1] LeWYS est un projet open-source développé au sein du consortium ObjectWeb. Il est téléchargeable à l'URL suivante : `http ://lewys.objectweb.org`

tion observée est déployée sur trois sites. Sur chacun de ces sites s'exécute un composant, appelé **pompe**, dont le rôle est de générer les informations de supervision pour le site. Ces différentes informations sont fournies par des **sondes**. Les informations générées au niveau de chaque site sont véhiculées auprès d'**observateurs** à l'aide de **canaux à événements** dynamiques construits en utilisant DREAM. LEWYS n'impose aucune contrainte sur la nature des systèmes observés. Si les seuls indicateurs nécessaires sont les ressource physiques (CPU, disque, mémoire, etc.), LEWYS n'impose aucune modification aux applications s'exécutant sur les sites observés. En revanche, s'il est nécessaire de collecter des données propres à une application, il faut développer des sondes appropriées, collaborant avec l'application pour fournir les données requises.

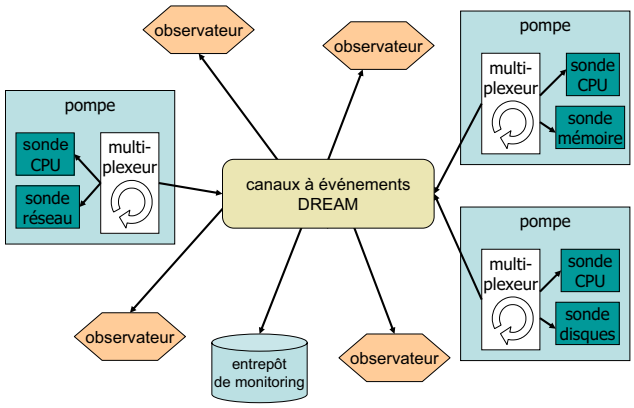

FIG. 10.1 – Exemple d'application de supervision construite avec LEWYS.

L'objectif de LEWYS n'est pas de fournir des implantations d'observateurs ; néanmoins, LEWYS propose un observateur particulier, appelé **entrepôt de monitoring**[2], qui est un composant permettant le stockage (dans une base de données) de l'ensemble des informations de supervision générées et le traitement hors-ligne de ces données (affichage et corrélation).

Les sections suivantes décrivent les différents composants du canevas LEWYS : la section 10.3 est consacrée à la pompe ; les différentes sondes sont présentées dans la section 10.4.

10.3 Le composant pompe

Une pompe est déployée sur chacun des sites sur lesquels l'application à observer s'exécute. Le rôle d'une pompe est de générer périodiquement des informations de supervision et de les transmettre aux observateurs intéressés. L'architecture de la pompe est représentée sur la figure 10.2.

Le composant principal de la pompe est le **multiplexeur**. Ce composant possède deux interfaces clients de collection : la première interface `Pull` est reliée à un ensemble de composants sondes. Le multiplexeur utilise la méthode `pull()` pour récupérer les messages produits par les sondes. La deuxième interface `Push` est reliée à un ensemble de canaux de communications. Le multiplexeur utilise la méthode `push(Message m)` pour transmettre un message à l'un des canaux.

Le fonctionnement du multiplexeur est le suivant : il possède une activité qui, périodiquement, "pull" un (ou plusieurs) message(s) sur une (ou plusieurs) de ses interfaces `Pull`, et "push" ce(s) message(s) sur une (ou plusieurs) de ses interfaces `Push`. Nous avons fait le choix de mettre l'activité dans le multiplexeur et non dans les sondes afin de limiter l'intrusivité de la pompe. Le

[2]Ce composant n'est pas décrit dans le reste de ce chapitre.

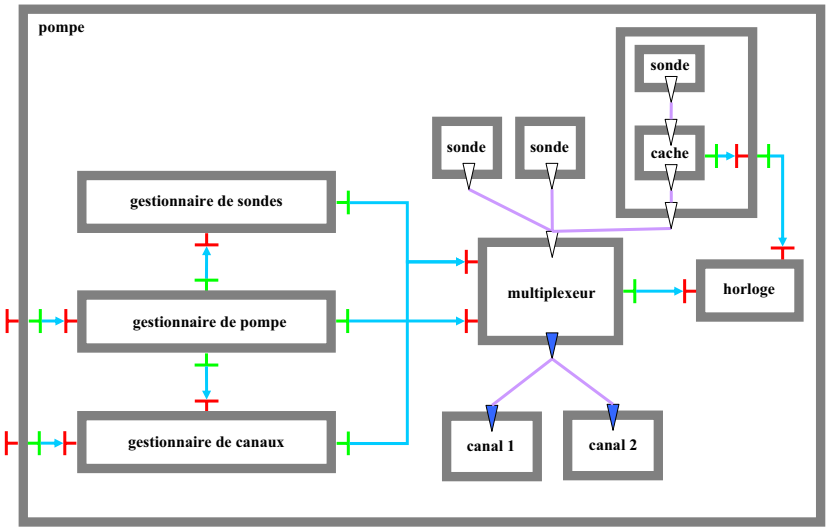

FIG. 10.2 – Architecture de la pompe.

multiplexeur estampille les messages qu'il envoie à l'aide du composant **horloge**. Notons que ce composant peut également être utilisé par des composants **caches** dans le cas où les sondes sont coûteuses. Le rôle d'un composant cache est de conserver la dernière valeur collectée pendant un laps de temps configurable.

Les sondes sont déployées par un composant appelé **gestionnaire de sondes**. Celui-ci implémente une interface serveur `ProbeManager` dont le rôle est de déployer une sonde et de la lier au multiplexeur.

Les canaux sont enregistrés en utilisant le **gestionnaire de canaux**. Contrairement aux sondes qui sont créées par le gestionnaire de sondes, le rôle du gestionnaire de canaux consiste uniquement à insérer un composant canal donné dans le composite **pompe** et à lier ce composant au multiplexeur.

Enfin, le **gestionnaire de pompe** permet aux utilisateurs d'interagir avec le multiplexeur : celui-ci implante une interface qui permet de souscrire aux informations de supervision d'une ou plusieurs sondes et de spécifier un ensemble de canaux auxquels ces informations doivent être envoyées. Il est également possible de spécifier la fréquence à laquelle les informations doivent être collectées. Par exemple, l'utilisateur peut demander à ce que les données concernant le CPU et la mémoire soient envoyées toutes les $200ms$ aux canaux 1 et 3.

10.4 Les composants sondes

LEWYS fournit différentes sondes permettant de réifier un vaste spectre de données provenant du système d'exploitation et des applications en cours d'exécution [CELQ04, CELQ05]. Il est, de plus, facile de développer ses propres sondes. La seule exigence concernant les sondes est qu'elles implantent l'interface `Pull` qui est utilisée par le multiplexeur pour collecter les données. Dans la suite de cette section, nous allons décrire les différentes sondes fournies dans LEWYS.

10.4.1 Les sondes du système d'exploitation

Les sondes Linux

Linux, comme d'autres systèmes d'exploitation UNIX, réifie l'état du noyau à travers un système de fichiers virtuel, généralement monté sous l'entrée `/proc`. [SH02] décrit plusieurs optimisations permettant de collecter de façon efficace des données depuis ce système de fichiers. Toutes les sondes Linux de LeWYS ont le même comportement : lors de son déploiement, une sonde analyse le fichier approprié afin de déterminer l'ensemble des *ressources* disponibles et de calculer des numéros d'indices correspondants. Ceci permet ensuite d'appliquer plusieurs optimisations comme (1) garder le même descripteur de fichier pour toutes les lectures, (2) collecter les informations en un seul bloc (les valeurs sont générées pour chaque lecture) et (3) accéder aux données à l'aide des indices pré-calculés, évitant ainsi des conversions intermédiaires.

Suivant ce modèle, LeWYS fournit les sondes nécessaires pour observer la consommation CPU, l'utilisation de la mémoire (totale, libre, disponible, partagée, en cache, etc.), l'utilisation des disques durs (nombre de lectures et d'écritures, nombre d'opérations fusionnées, débit des opérations de lecture/écriture, etc.), les performances du réseau (nombre de connexions actives, débits entrant/sortant, taux de pertes de paquets, etc.) et l'état du noyau (nombre de processus, nombre d'interruptions, etc.).

Les sondes Windows

Windows fournit une infrastructure d'observation permettant aux applications d'accéder aux performances du système sous-jacent. Ces informations concernent (1) les composants matériels tels que la mémoire et le processeur, (2) les composants systèmes tels que les processus et les interruptions et (3) les composants applicatifs utilisant les services Windows pour exposer leurs performances. Quel que soit le type du composant observé, les données sont structurées en trois niveaux hiérarchiques : *objet*, *instance* et *compteur* :

- un *objet* désigne une ressource observable présente dans le système. Un objet peut donc concerner les trois types de ressources précédents.
- une *instance* représente un sous-ensemble d'un objet. Par exemple, un objet "réseau" peut avoir plusieurs instances relatives aux différentes interfaces réseau présentes dans le système.
- un objet (ou une instance) définit plusieurs *compteurs* reflétant les métriques mesurées de cet objet. Une instance réseau, par exemple, définit des compteurs relatifs au débit de transmission, débit de réception, nombre de paquets perdus, nombre de connexions actives, etc.

Comme le montre la figure 10.3, plusieurs composants se chargent de mesurer les performances des différents composants observables. Ces données sont accessibles aux applications à travers la base de registres (sous la clé `HKEY_PERFORMANCE_DATA`). Les données ne sont pas stockées dans le registre lui-même, mais l'accès à cette clé provoque leur collecte à partir des composants appropriés. Nous avons développé une librairie native qui permet aux sondes LeWYS de récupérer ces données de manière efficace.

10.4.2 Les sondes applicatives

LeWYS fournit des sondes dont le but est de collecter des informations applicatives dans les environnements Java. Ces sondes sont basées sur JMX (*Java Mananagement Extensions*), un ensemble de spécifications pour l'administration des applications Java. Comme nous l'avons représenté sur la figure 10.4, l'architecture JMX fait intervenir les trois niveaux suivants :

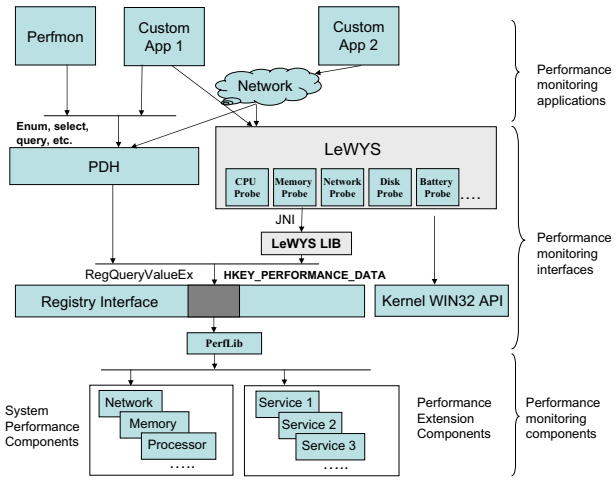

FIG. 10.3 – Les structures Windows d'accès aux métriques du système.

- le niveau *instrumentation* est composé d'un ensemble de *MBeans* déployés au sein de l'application administrée. Les MBeans permettent de collecter des informations sur l'application et d'effectuer des opérations d'administration.
- le niveau *serveur de MBeans* permet de gérer les MBeans déployés.
- le niveau *services distribués* est composé d'un ensemble de clients JMX et de leurs canaux de communication vers le serveur de MBeans.

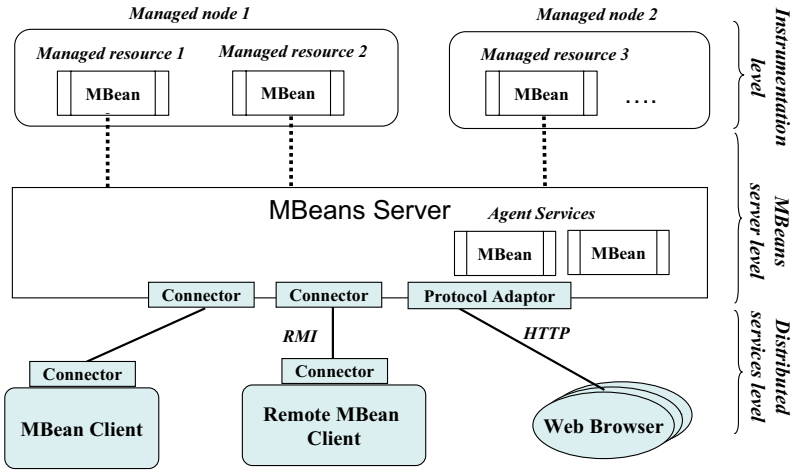

FIG. 10.4 – L'architecture JMX.

Les sondes JMX de LEWYS sont des clients JMX. Chaque sonde est associée à un serveur de MBeans. Durant son initialisation, elle s'informe sur la liste des MBeans enregistrés dans le serveur, ainsi que sur leurs attributs. La communication avec le serveur est basée sur RMI. En conséquence, les sondes peuvent soit être déployées localement sur la même machine que le serveur, soit être déployées sur une machine distante.

10.5 Evaluation

Dans cette section, nous présentons une évaluation qualitative et quantitative de LeWYS.

10.5.1 Evaluation qualitative

Reprenons les différentes caractéristiques que nous mentionnons dans l'introduction comme nécessaires pour qu'une application de supervision soit utilisable dans de nombreux contextes :

- *adaptation à différentes échelles de systèmes* : en permettant de déployer dans la pompe des canaux de communication arbitrairement complexes, LeWYS permet de construire des applications de supervision adaptées à différentes tailles de systèmes. Par exemple, il est très simple de construire une console "locale" de supervision qui affiche périodiquement un certain nombre d'indicateurs. Pour ce faire, il suffit d'enregistrer comme canal un composant qui ne fait qu'afficher les messages qu'il reçoit. Le système d'observation ainsi construit a uniquement pour but de donner périodiquement des informations sur la machine sur laquelle il est déployé. Il est également possible de construire des systèmes de propagation d'événements à très grande échelle en utilisant, par exemple, le protocole de diffusion probabiliste décrit au chapitre 7.
- *possibilité de remonter divers indicateurs* : l'architecture à base de composants de LeWYS a été conçue de façon à ce qu'il soit aisé de déployer des sondes spécifiques à un système donné. Il est uniquement nécessaire de développer les classes des sondes et de les enregistrer dans un fichier de configuration du gestionnaire de sondes qui sera ainsi capable de les déployer.
- *possibilité de construire des chaînes arbitrairement complexes de traitement et de propagation de l'information collectée* : l'utilisation de canaux à événements Dream permet de construire des chaînes de propagation des données de supervision arbitrairement complexes. En effet, il est aussi bien possible de construire des systèmes de dissémination simplistes (sockets TCP/IP) que des systèmes de traitement de l'information basés sur des règles "événement → réaction" en utilisant, par exemple, la plate-forme ScalAgent présentée au chapitre 9.
- *possibilité de configurer dynamiquement les divers paramètres de l'application de supervision* : l'utilisation du modèle de composants Fractal permet de modifier dynamiquement l'architecture de l'application de supervision. Des exemples de reconfigurations dynamiques sont l'ajout d'une sonde, la modification de la fréquence de collecte d'une sonde, l'ajout d'un composant *cache* pour limiter l'intrusivité de la sonde, l'ajout d'un canal de dissémination des données, etc.
- *non-intrusivité* : l'architecture de la pompe a été conçue de manière à minimiser les ressources consommées par l'application de supervision : la seule activité de la pompe se trouve dans le multiplexeur. Par ailleurs, des caches permettent de limiter l'intrusivité des sondes les plus coûteuses.

10.5.2 Evaluation quantitative

Sondes Linux

Le tableau 10.1 présente les performances des différentes sondes Linux : pour chacune d'elle, nous évaluons le temps moyen de collecte des informations de supervision. Les mesures ont été effectuées sur des Pentium IV 2 Ghz, avec 512 Mo de mémoire, un disque IDE de 40 GO avec 6 partitions. La machine utilise un noyau Linux 2.4.20 et la JVM Sun 1.4.2.

Sonde	Temps moyen de collecte
CPU	23.4 μs
Mémoire	40.7 μs
Disques	31.5 μs
Réseau	27.8 μs
Noyau	23.0 μs

TAB. 10.1 – Performances des sondes Linux.

Le coût dominant dans chaque mesure est celui de l'accès au système de fichier `/proc`. La sonde mémoire a des performances moindres du fait qu'elle requiert des accès à deux fichiers `/proc` différents. Néanmoins, le coût reste négligeable et prouve qu'il est possible de construire des applications de supervision peu intrusives.

Sondes Windows

Le tableau 10.2 présente les temps moyen de collecte des sondes Windows. Ces résultats ont été obtenus sur un Pentium IV 2 Ghz, avec 512 Mo de mémoire, un disque IDE de 40 GO avec 6 partitions. La machine utilise Windows 2000 et la JVM Sun 1.4.2.

Sonde	Temps moyen de collecte
CPU	0.72 ms
Mémoire	0.35 ms
Disques	1.9 ms
Réseau	12 ms
Noyau	1.2 ms

TAB. 10.2 – Performances des sondes Windows.

Le temps moyen de collecte des informations de supervision est largement supérieur au temps nécessaire pour les sondes Linux. Ainsi, la sonde la plus efficace nécessite 0.35 ms pour collecter l'ensemble des indicateurs qu'elle réifie. Les mauvaises performances des sondes Windows s'expliquent par deux facteurs principaux :

- elles effectuent des appels JNI afin d'accéder au fichier DLL contenant le code d'interfaçage avec la base de registres.
- elles nécessitent des accès à la base de registres du système.

Par ailleurs, on constate une forte variation entre les performances des différentes sondes (de 0.35 ms à 12,2 ms) qui s'explique par les différences de quantités de mémoire nécessaires à l'exécution de chacune des sondes. Par exemple, les très mauvaises performances de la sonde réseau s'expliquent par le fait qu'elle génère plus de 1ko de données à chaque appel.

10.6 Travaux connexes

Plusieurs travaux ont été consacrés aux systèmes de supervision, notamment dans le contexte des grappes de machines. Parmi eux, on distingue deux types de travaux : ceux qui se focalisent

sur la supervision des applications [net02, pab99, rem02], et ceux qui fournissent des plates-formes de supervision plus générales. Notre étude de l'état de l'art s'intéresse à cette deuxième catégorie.

PHOENIX [SBF02] est un système dédié à l'observation de grappes de machines. Il permet d'observer aussi bien des ressources physiques que des applications. Pour ces dernières, il propose une bibliothèque contenant un ensemble de primitives d'observation permettant de remplacer les appels systèmes effectués par les applications. Cette librairie doit être liée aux applications observées. Par ailleurs, PHOENIX définit un langage permettant d'exprimer des contraintes sur les granularités auxquelles les informations de supervision doivent être produites. Enfin, PHOENIX fournit deux outils permettant respectivement d'afficher et d'analysr les informations de supervision générées et d'effectuer un équilibrage de charge entre les machines de la grappe observée. La principale limitation de PHOENIX provient du fait qu'il utilise un système de transport des événements basé sur des primitives de multicast. De fait, il impose que les machines observées soient localisées sur un même LAN.

Ganglia [gan02] est un système dédié à l'observation de grappes et de grilles de machines. Il repose sur l'utilisation de deux types de démons systèmes : *gmond* et *gmetad* communiquant via un protocole d'annonce/écoute en multicast. Un gmond s'exécute sur chaque nœud observé et se charge de collecter un ensemble d'informations sur le système. Celles-ci sont envoyées périodiquement à un gmetad qui se charge de leur collecte et leur traitement. Outre le fait qu'il offre plus de fonctions, LeWYS tire profit de l'architecture Dream en offrant plus de flexibilité dans le placement et la construction des traitements des données observées (filtrage, agrégation, etc.). Par ailleurs, l'emploi d'un protocole d'annonce/écoute en multicast rend Ganglia peu apte à l'observation de systèmes à grande échelle.

L'architecture JAMM [TCG+00] définit une plate-forme à agents automatisant le déploiement de sondes d'observations et la collecte des événements correspondants. Il est possible de construire un système analogue avec LeWYS en utilisant la plate-forme ScalAgent décrite dans le chapitre 7. L'avantage de l'utilisation de Dream est qu'il est, par exemple, aisé de modifier dynamiquement les propriétés d'exécution des agents (ordonnancement des messages, atomicité de leurs réactions, persistance, etc.).

WatchTower [KSD02] est un système de supervision spécifique aux environnements Windows. Il est basé sur PDH (*Performance Data Helper*), une interface de programmation qui masque la complexité de la base de registres de Windows. Bien que cette interface donne accès à tous les indicateurs de performances, elle est coûteuse car elle requiert un accès à la base de registres pour chaque valeur collectée. Dans LeWYS, ceci est optimisé du fait qu'un seul accès est nécessaire pour récupérer les différents indicateurs de performance d'une sonde. Par ailleurs, le canevas LeWYS étant implanté en Java, il est indépendant de la plate-forme d'exécution : il est uniquement nécessaire de développer les sondes adéquates.

Enfin, citons NWS (*Network Weather Service*) [WSH99], un système de supervision qui permet de faire des prévisions sur les performances à court terme du réseau. NWS utilise différents composants écrits en C et utilise des sockets TCP/IP pour la communication des données. L'accent a été mis sur la minimisation de l'intrusivité du système. LeWYS est un système de supervision plus général et de plus haut niveau : la possibilité de s'abonner à différents canaux d'événements permet notamment de construire des chaînes de traitement de l'information complexes, ce qui n'est pas le but de NWS. Il est néanmoins possible de fournir les mêmes fonctions que NWS. Pour ce faire, il suffit de construire des canaux à événements Dream utilisant de simples sockets TCP/IP.

10.7 Conclusion

Dans ce chapitre, nous avons présenté LeWYS, un canevas logiciel à composants dédié à la construction d'applications de supervision. Les objectifs de LeWYS sont multiples. A savoir : autoriser la construction d'applications de supervision pour des systèmes de tailles variables ; permettre au développeur de construire des collecteurs ad hoc pour générer les informations de supervision dont il a besoin ; fournir des composants pour l'acheminement et le traitement des informations. Par ailleurs, l'architecture de LeWYS a été conçue de manière à minimiser l'instrusivité des applications de supervision.

Remerciements : les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Hazem Elmeleegy, doctorant à l'Université de Purdue (USA), ainsi qu'Oussama Layaida et Emmanuel Cecchet, respectivement doctorant et chargé de recherches au sein du projet SARDES.

Chapitre 11

Freecast : un protocole de diffusion avec ordre total uniforme

Sommaire

Ce chapitre présente FREECAST, un protocole de diffusion avec ordre total uniforme. Outre le fait que la communication de groupe est un excellent terrain d'expérimentation pour DREAM, nous avons réalisé ce protocole car il permet d'implanter des protocoles de réplication active qui sont nécessaires pour construire des systèmes d'administration tolérant les fautes, comme nous le présentons dans le chapitre suivant.

Nous commençons ce chapitre par une introduction qui justifie ce travail et en présente les éléments-clés. Nous décrivons ensuite le protocole de diffusion et les différents choix possibles quant à la dissémination des messages dans les sections 11.2 et 11.3. La section 11.4 présente l'implantation du protocole. Enfin, la section 11.5 dresse un bref état de l'art, avant de conclure dans la section 11.6.

11.1 Introduction

11.1.1 Réplication par protocole de diffusion

L'augmentation de la puissance des ordinateurs prédite par la loi de Moore n'a pas été accompagnée d'une augmentation similaire de leur fiabilité. En revanche, la diminution rapide des coûts du matériel a favorisé la mise en place de systèmes de fiabilité basés sur la réplication. La clé pour faire fonctionner ces mécanismes de réplication est de fournir une couche logicielle qui en masque les difficultés aux développeurs d'applications et qui les rend transparents aux clients des services développés.

Le principe de la réplication est simple : chaque processus maintient une copie de l'objet répliqué. Toutes les invocations sont diffusées à l'ensemble des processus qui les invoquent sur la copie qu'ils possèdent[1]. Pour que ce mécanisme fonctionne, il est nécessaire de fournir un mécanisme d'ordonnancement garantissant (1) que toutes les invocations diffusées sont reçues dans le même ordre par l'ensemble des processus, et (2) que chaque invocation est, soit exécutée par tous les processus, soit par aucun. On parle alors de *protocole de diffusion avec ordre total uniforme* (utoBcast pour *uniform total order broadcast* [HT93a]).

11.1.2 Métriques d'évaluation des performances

Le moyen le plus simple de raisonner sur les systèmes se basant sur des envois de messages est d'utiliser le *modèle de rondes* qui stipule qu'un processus peut envoyer un message à destination d'un où plusieurs processus au début de chaque ronde et peut recevoir des messages émis par les autres processus à la fin de chaque ronde. Si l'on considère que le système se comporte de façon synchrone la plupart du temps, ce modèle permet de définir le *coût* d'une opération comme le nombre de rondes nécessaires à son exécution.

Ce modèle est utilisé dans la plupart des papiers théoriques se focalisant sur l'évaluation de la *latence* des algorithmes, c'est-à-dire sur le nombre de rondes nécessaires pour délivrer un message. En pratique, néanmoins, le *débit* — correspondant au nombre de messages que chaque processus peut recevoir par unité de temps — est aussi important, si ce n'est plus, que la latence. En effet, il y a deux sources de latence qui doivent être considérées au sein d'un processus : le temps nécessaire pour délivrer un message émis et le temps que ce message passe dans des files d'attente avant de pouvoir être émis. Si le processus reçoit peu de messages, sa file d'attente est vide et la latence totale observée correspond à la latence de l'algorithme de diffusion. Quand, au contraire, un processus reçoit un grand nombre de messages simultanément, la latence totale observée devient très dépendante du débit du processus : plus le processus peut traiter de messages par unité de temps, plus la file d'attente des messages à envoyer est petite.

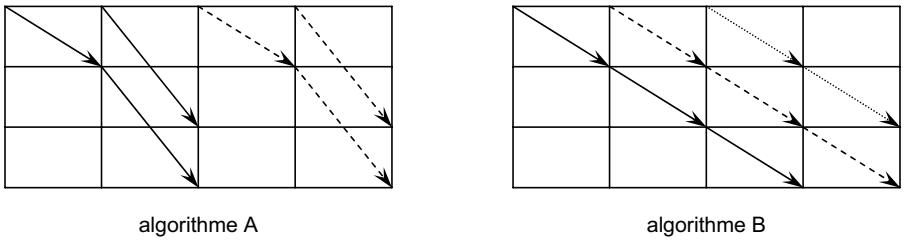

FIG. 11.1 – Deux exemples de protocoles de diffusion non fiables.

[1] En pratique, seules les invocations modifiant l'état de l'objet doivent être diffusées aux autres processus ; les autres invocations peuvent être exécutées en parallèle.

Il n'est pas difficile de comprendre pourquoi un algorithme optimisant la latence n'optimise pas nécessairement le débit : considérons pour cela les deux algorithmes de diffusion (non fiables) présentés sur la figure 11.1.

- Dans l'algorithme $\mathcal{A}$, le processus p_1 envoie tout d'abord le message à p_2. Dans la ronde suivante, le message est simultanément transmis de p_2 vers p_4 et de p_1 vers p_3.
- Dans l'algorithme $\mathcal{B}$, le processus p_1 envoie tout d'abort le message à p_2. Dans la ronde suivante, p_2 transmet le message à p_3 qui le transmet, à son tour, à p_4 dans la ronde suivante.

En conséquence, l'algorithme $\mathcal{A}$ a une latence de 2 et l'algorithme $\mathcal{B}$ a une latence de 3. L'algorithme $\mathcal{A}$ est donc optimal pour la latence. Cependant, si l'on considère le débit, on constate que l'algorithme $\mathcal{A}$ permet de débuter une nouvelle diffusion de message toutes les 2 rondes, alors que l'algorithme $\mathcal{B}$ permet d'en débuter une toutes les rondes. En conséquence, le débit obtenu avec l'algorithme $\mathcal{B}$ est le double de celui obtenu avec l'algorithme $\mathcal{A}$.

11.1.3 Proposition

Les protocoles de diffusion avec ordre total uniforme sont considérés comme très coûteux à mettre en œuvre. Nous montrons, dans ce chapitre, que ce n'est pas nécessairement le cas si l'algorithme est implanté avec des hypothèses en accord avec l'architecture du système ciblé. Nous considérons l'implantation d'un tel protocole pour des grappes de machines. Nous énonçons l'hypothèse qu'une seule panne de machine est susceptible de se produire par période stable de l'algorithme, c'est-à-dire qu'une seule machine tombe en panne à un moment donné et que le protocole retrouve un état stable avant qu'une nouvelle panne intervienne. Cette hypothèse est réaliste pour la plupart des grappes de machines. Par exemple, sur la grappe de 200 nœuds de l'INRIA, il se produit une faute par jour en moyenne. Par ailleurs, soit la faute est isolée, soit il s'agit d'une panne électrique ou de climatisation, cas dans lesquels toutes les machines tombent en panne simultanément. Notons, enfin, que si ne tolérer qu'une faute procure un bénéfice net en termes de performances, il serait aisé de modifier l'algorithme que nous proposons afin qu'il tolère plus d'une faute.

Nous émettons, par ailleurs, l'hypothèse que nous disposons d'un détecteur de fautes parfait [CT96]. Son implantation est simple du fait qu'elle repose sur TCP qui fournit un modèle de transfert de données orienté connexion. Enfin, bien que le système puisse être asynchrone, les performances de notre algorithme sont optimisées pour les périodes stables pendant lesquelles le système se comporte de façon synchrone, c'est-à-dire lorsque le réseau assure une délivrance des messages dans un certain laps de temps et qu'aucune faute ne se produit.

Comme les autres protocoles de diffusion avec ordre total uniforme, notre protocole peut être séparé en deux parties [YMV+03] : ordonnancement des messages et dissémination fiable de chaque message à l'ensemble des processus. De façon intuitive, l'ordonnancement consiste à donner un numéro de séquence unique à chaque message. Concernant la dissémination des messages, nous étudions trois choix possibles :

- roundAlg qui utilise le modèle de rondes et dont l'optimalité en termes de latence et de débit peut être prouvée de façon théorique dans le cas 1-N.
- treeAlg qui utilise des arbres couvrants. Les résultats théoriques prouvent que treeAlg est optimal en termes de débit dans le cas 1-N et qu'il n'est optimal en termes de latence que si aucune primitive native de diffusion (e.g. IP multicast) n'est disponible.
- chainAlg qui utilise une chaîne de processus pour la diffusion. Les résultats théoriques montrent que chainAlg est uniquement optimal en termes de débit dans le cas 1-N ; en revanche, sa mise en œuvre est très simple.

11.1.4 Principaux résultats

Protocole expérimental

Nous avons réalisé des expériences sur deux grappes de machines différentes : une grappe de 100 machines avec des bi-processeurs *Itanium-2* 900MHz, 3Go de mémoire et réseau Fast Ethernet, exécutant linux 2.4.21 et une grappe de 8 machines avec des bi-processeurs *Xeon* 1.8 Ghz, 1Go de mémoire et réseau Fast Ethernet, exécutant Linux 2.4.20. Les deux grappes utilisent un réseau entiérement switché. Les performances des réseaux sur les deux grappes mesurées à l'aide de Netperf [Net] sont résumées dans le tableau 11.1.

Cluster	Protocol	Bandwith
Itanium	TCP	94 Mb/s
Itanium	UDP	93 Mb/s
Xeon	TCP	91 Mb/s
Xeon	UDP	92 Mb/s

TAB. 11.1 – Performances des réseaux des deux grappes utilisées pour les expérimentations.

Les tests effectués impliquent n processus parmi lesquels soit (1) un seul processus diffuse des messages aux autres processus (1-N), soit (2) tous les processus diffusent des messages aux autres processus (N-N). On mesure le temps nécessaire aux processus émetteurs de messages pour diffuser l'ensemble de leurs messages. Nous faisons varier la fréquence à laquelle les messages sont émis par les différents émetteurs. Notons que les différents émetteurs sont synchronisés par un processus *leader* via des messages unicast dont la fréquence est négligeable et qui ne perturbent donc pas les expériences.

Résultats

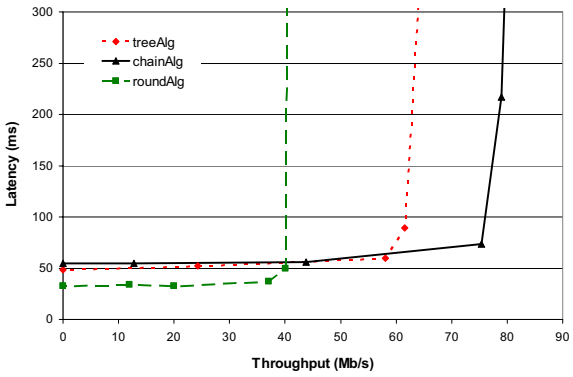

FIG. 11.2 – Latence en fonction du débit pour les trois choix possibles pour la diffusion des messages.

Les trois choix possibles pour la dissémination des messages ont été comparés. La figure 11.2 montre les résultats (débit en fonction de latence) obtenus pour un test N-N faisant intervenir cinq processus (sur grappe Itanium) diffusant des messages d'une taille de 100Ko. Bien que l'algorithme roundAlg obtienne la meilleure latence, la différence est non significative en comparaison de la perte de débit ($\approx 50\%$) par rapport à l'algorithme chainAlg. L'algorithme treeAlg

est comparable en latence et offre un débit également nettement inférieur ($\approx 20\%$) à celui de chainAlg.

Les mauvaises performances de l'algorithme roundAlg sont dues à l'utilisation du protocole de communication IP multicast dont les performances lors de contentions du réseau sont très mauvaises (jusqu'à 90% de pertes de paquets) comme il a été montré dans [ACL04]. Ces mauvaises performances s'expliquent en partie du fait que certains équipements réseau privilégient le trafic unicast au trafic multicast. Notons qu'il est possible d'améliorer ces performances en mettant en place des mécanismes de contrôle de flux, mais qu'il n'existe pas de solution satisfaisante à l'heure actuelle.

Les mauvaises performances de l'algorithme treeAlg peuvent être expliquées par le fait que les hypothèses sur la synchronie du système ne sont pas vérifiées — en particulier dès lors que le nombre de processus augmente. De fait, les enchevêtrements d'arbres couvrants utilisés dans l'algorithme induisent un grand nombre de collisions sur le réseau, ce qui dégrade significativement les performances.

11.1.5 Organisation du chapitre

Ce chapitre s'organise de la façon suivante : nous décrivons le protocole de diffusion avec ordre total uniforme dans la section 11.2. Nous détaillons les trois algorithmes de dissémination — roundAlg, treeAlg et chainAlg — dans la section 11.3. La section 11.4 présente l'implantation du protocole. Enfin, nous dressons un état de l'art dans la section 11.5, avant de conclure dans la section 11.6.

11.2 Le protocole de diffusion avec ordre total uniforme

Cette section présente le protocole de diffusion avec ordre total uniforme. Nous commençons par une description informelle du protocole, puis donnons sa description algorithmique.

11.2.1 Description informelle

Le protocole de diffusion est défini par deux primitives — utoBroadcast et utoDeliver — et par les quatre propriétés suivantes :

- **validité** (validity) : si un processus correct p_i diffuse (utoBroadcast) un message m à un autre processus correct p_j, alors p_j finira par délivrer (utoDeliver) m.
- **intégrité**[2] (integrity) : pour tout message m, si un processus p_j délivre (utoDeliver) m, il le délivre au plus une fois et il existe un processus p_i qui a diffusé (utoBroadcast) m.
- **accord uniforme** (uniform agreement) : si un processus p_i délivre (utoDeliver) un message m, alors chaque processus correct p_j finira par délivrer (utoDeliver) m.
- **ordre total uniforme** (uniform total order) : pour chaque couple de messages m_1 et m_2, si un processus p_i délivre (utoDeliver) m_1 sans avoir délivré (utoDeliver) m_2, alors aucun processus p_j ne délivrera (utoDeliver) m_2 avant m_1.

Pour diffuser (utoBroadcast) un message, un processus p_i l'envoie à un processus particulier, appelé *leader*[3], et le garde en mémoire jusqu'à ce qu'il le délivre (utoDeliver) lui-même. Le rôle du *leader* est d'ordonnancer les messages qu'il reçoit. Pour ce faire, il assigne un numéro de séquence unique incrémenté de façon monotone à chaque message reçu. Le *leader* envoie ensuite le message à un processus, appelé *backup*, dont le rôle est de prendre le rôle du *leader* lorsque celui-ci défaille.

[2] Cette propriété est parfois appelée *validité*.

[3] Une manière très simple d'élire un *leader* est de prendre le processus correct possèdant le plus petit identifiant.

La paire *leader* - *backup* implante un séquenceur [Bir85][4]. Lorsque le *backup* reçoit le message m, il peut le délivrer (utoDeliver) et le diffuser à tous les autres processus. Les processus *leader* et *backup* peuvent supprimer le message m de leur mémoire dès lors qu'ils ont reçu un acquittement de chaque processus correct.

Lorsque le *leader* défaille, *backup* devient le nouveau *leader* et procède à l'election d'un nouveau *backup*. Le nouveau *leader* transfère son état vers le nouveau *backup* avant d'accepter toute diffusion d'un nouveau message.

11.2.2 Description algorithmique

La description algorithmique du protocole est donnée ci-après. Le détecteur de fautes parfait est symbolisé par la variable $\mathcal{D}$ et contient la liste des processus qui ont défailli. Nous formulons l'hypothèse qu'un seul processus peut défaillir par période stable, c'est-à-dire avant qu'un nouveau *leader* et qu'un nouveau *backup* aient été élus et que tous deux connaissent l'ensemble des messages qui n'ont pas encore été reçus par tous les processus. Ces messages sont stockés dans la variable *pendingMessages*.

Le protocole fait intervenir différents types de messages. Chaque message a un champ qui définit son *type* et un champ de *données*. Les types de messages utilisés sont les suivants :

- [DAT, m] contient un message m émis par un processus client à destination du *leader*.
- [ACK, m] contient un acquittement du message m. Notons que pour des raisons de performances, le message ne contient pas m, mais un identifiant de m.
- [REP, m] contient un message m émis par le *leader* à destination du *backup*.
- [UTO, m] contient un message m à délivrer (*utoDeliver*) par le processus récepteur. Cette délivrance est faite en accord avec le numéro de séquence contenu dans le message.
- [UPD, *state*] contient l'information envoyée par le nouveau *leader* au processus choisi pour devenir le nouveau *backup*.
- [BAK, p] contient l'information envoyée par le nouveau *leader* à l'ensemble des processus pour les informer de l'identité du nouveau *backup*.

Dans la description algorithmique, le symbole $*$ est utilisé pour dénoter le fait que le champ d'un message peut prendre n'importe quelle valeur. Par ailleurs, les processus *leader* et *backup* maintiennent une liste des processus S dont la mise à jour n'est pas décrite par souci de clarté. Enfin, la dissémination des messages UTO est représentée par les deux primitives beBroadcast et beDeliver. Cette dissémination doit garantir une sémantique *au mieux* (*best effort*) caractérisée par les deux propriétés suivantes :

- **validité** (validity) : si un processus correct p_i dissémine (beBroadcast) un message m, alors tout processus correct p_j finira par délivrer (beDeliver) m.
- **intégrité** (integrity) : tout message m est délivré (beDeliver) au plus une fois et seulement s'il a été diffusé (beBroadcast) par un processus p_i.

Notons que lorsqu'un processus p_i reçoit un message de type DAT, il devient le nouveau *leader*. En effet, pour qu'un processus p_j envoie un message de type DAT au processus p_i, il faut qu'il ait élu ce dernier *leader*. Du fait que nous supposons l'existence d'un détecteur de fautes parfait, nous sommes assurés que la suspicion du processus p_j à l'égard du précédent *leader* est correcte. Des situations similaires peuvent se produire avec les autres types de messages.

[4]Le séquenceur n'est composé que deux processus car le protocole décrit ne tolère la faute que d'un seul processus. Il est possible d'implanter un séquenceur tolérant f fautes à l'aide de f processus.

Procédure exécutée par tout processus p_i

```
 1: procedure utoBroadcast [m]
 2:    while true do
 3:       send [DAT, m] to p_leader
 4:       wait until (i) utoDelivered m or (ii) p_leader ∈ D
 5:       if utoDelivered m then
 6:          send [ACK, m] to p_leader        {Can be piggy-backed on next message that is to be UTO-broadcast}
 7:          break
 8:       else
 9:          p_leader := p_backup
10:       end if
11:    end while
12: end

13: upon receiving [UPD, S', sn'] do                                   {Process p_i is elected backup}
14:    S := S'
15:    sn := sn'
16: end upon

17: upon receiving [BAK, p] do                {Backup has crashed and a new process has been elected backup}
18:    p_backup := p
19: end upon

20: nextToDeliv := 0                          {Sequence number of next message to be delivered is initially zero}
21: upon beDeliver [UTO, sn, m, p_j] from p_i do
22:    if [*, m, *] ∉ toDeliv then
23:       toDeliv := toDeliv ∪ {[sn, m, p_j]}
24:       while [nextToDeliv, *, *] ∈ toDeliv do
25:          utoDeliver [m, p_j]
26:          toDeliv := toDeliv\{[nextToDeliv, *, *]}
27:          nextToDeliv := nextToDeliv + 1
28:       end while
29:    end if
30:    send [ACK, m] to p_leader and p_backup  {ACK messages can be sent at a fixed interval instead of each time.}
31: end upon
```

Procédure exécutée par le processus p_{leader}

```
1: S := initial set of participating processes                    {Initial configuration is known}
2: sn := 0                                                        {Sequence number is initially zero}
3: upon receiving [DAT, m] from p_i do
4:    pendingMessages := pendingMessages ∪ {[sn, m, p_i]}
5:    send [REP, sn, m, p_i] to p_backup
6:    sn := sn + 1
7: end upon

8: upon p_backup ∈ D do                                           {Backup crashes}
9:    p_backup := first(S\D)
10:   send [BAK, p_backup] to all but p_backup
11:   send [UPD, S, sn] to p_backup
12:   for all element ∈ pendingMessages do
13:      send [REP, element] to p_backup
14:   end for
15: end upon

16: upon receiving [ACK, m] do
17:   if [*, m, *] ∉ toDeliv then
18:      toDeliv := toDeliv ∪ {[sn, m, p_j]}
19:      while [nextToDeliv, *, *] ∈ toDeliv do
20:         utoDeliver [m, p_j]
21:         toDeliv := toDeliv\{[nextToDeliv, *, *]}
22:         nextToDeliv := nextToDeliv + 1
23:      end while
24:   end if
25: end upon
```

Procédure exécutée par les processus p_{leader} et p_{backup}

```
1: upon receiving [ACK, m] from all processes not belonging to D do
2:    pendingMessages := pendingMessages\{[*, m, *]}
3: end upon
```

Procédure exécutée par le processus p_{backup}

```
1: upon receiving [REP, sn, m, p_j] from p_i do
2:    pendingMessages := pendingMessages ∪ {[sn, m, p_j]}
3:    beBroadcast [UTO, sn, m, p_j] to all but p_leader
4: end upon

5: upon p_leader ∈ D do                                           {Leader crashes, backup becomes leader}
6:    p_backup := first(S\D)
7:    send [BAK, p_backup] to all but p_backup
8:    send [UPD, S] to p_backup
9:    for all element ∈ pendingMessages do
10:      send [REP, element] to p_backup
11:   end for
12: end upon
```

11.3 Trois choix possibles pour la dissémination des messages

La dissémination des messages de type UTO est effectuée par le processus *backup* suivant une sémantique *au mieux* ; elle est représentée par les primitives beBroadcast et beDeliver dans la description algorithmique donnée dans la section précédente. Dans cette section, nous étudions trois choix possibles pour effectuer cette dissémination : l'utilisation d'IP multicast (optimale en latence et débit dans le cas 1-N dans le modèle de rondes), l'utilisation d'arbres couvrants (optimal en débit dans le cas 1-N et en latence lorsqu'il n'existe pas de procédure native de diffusion) et l'utilisation d'une chaîne de processus (proposition dite pragmatique).

11.3.1 Dissémination dans le modèle de rondes

Dissémination

La dissémination dans l'algorithme roundAlg est réalisée à l'aide d'IP multicast. Afin de respecter les hypothèses de validité et d'intégrité de la dissémination, il est nécessaire d'utiliser un protocole fiabilisant les échanges effectués à l'aide d'IP multicast. Nous avons décidé d'utiliser un protocole à base d'acquittements négatifs. La mise en œuvre de ce protocole n'est pas triviale du fait qu'elle nécessite de mettre en place un ramasse-miettes distribué en charge de détruire les messages reçus par l'ensemble des processus.

Il peut être prouvé que l'algorithme roundAlg est optimal en termes de nombre de rondes nécessaires (=3) à la diffusion d'un message avec ordonnancement total et uniforme. Par ailleurs, chaque processus peut délivrer un message par ronde, ce qui rend également l'algorithme optimal en termes de débit dans le cas 1-N.

Performances

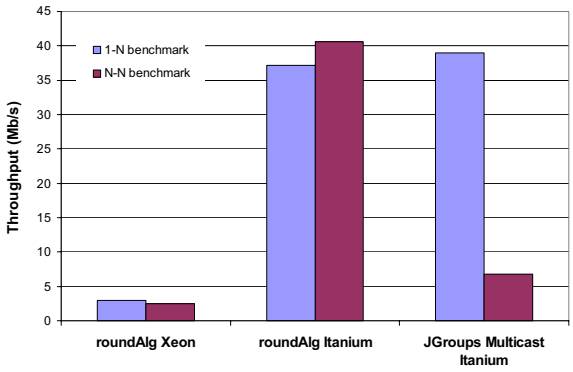

FIG. 11.3 – Comparaisons de l'algorithme roundAlg et de JGroups.

Les performances de l'algorithme roundAlg sont représentées sur la figure 11.3. L'expérience réalisée mesure le débit maximum obtenu pour la diffusion (1-N et N-N) de messages de 10Ko entre 5 processus. Nous avons également effectué les mêmes tests en remplaçant le protocole de communication de diffusion que nous proposons par le protocole de diffusion avec ordre FIFO proposé par JGroups [jgr05]. JGroups est un canevas permettant de construire des protocoles de communication de groupe. Le protocole de diffusion avec ordre total étant inexploitable pour raison de bogues, nous avons utilisé le protocole de diffusion avec ordre FIFO.

Il est tout d'abord intéressant de noter la disparité entre les performances de la grappe de Xeon et la grappe d'Itanium. Les performances très mauvaises de la grappe Xeon s'expliquent

par un très fort taux de perte des paquets IP multicast ($\approx 70\%$). Les résultats sont les mêmes avec un nombre différent de processus. Les raisons de ces pertes de paquets sont à la fois d'origine matérielle (cartes réseaux) et logicielle (implantation de la pile IP multicast).

Par ailleurs, il est intéressant de noter que le débit obtenu par l'algorithme roundAlg sur la grappe Itanium dans le test 1-N est légèrement inférieur à celui de JGroups. Ceci peut être expliqué par le fait que le protocole utilisé par JGroups n'impose pas d'ordre total. En revanche, dans les tests N-N, l'algorithme roundAlg assure des performances nettement supérieures à celles de JGroups. Ceci s'explique très simplement par le fait que dans l'algorithme que nous proposons, toutes les dissémination de messages émanent du même processus (*backup*). De fait, il y a moins de collisions que dans le protocole utilisé par JGroups dans lequel les dissémination peuvent être initiées par chaque processus.

11.3.2 Dissémination à l'aide d'arbres couvrants

Dissémination

La dissémination dans l'algorithme treeAlg est effectuée à l'aide d'arbres couvrants. Les processus représentent les nœuds de l'arbre et les communications entre processus sont effectuées à l'aide du protocole TCP. La dissémination de messages à l'aide de protocoles unicast a été étudiée dans le contexte des grappes de machines de haute performance ne bénéficiant pas de primitives de diffusion natives [KSSS93, BNK94, BNK97, ABM87]. Ces travaux formulent l'hypothèse selon laquelle l'envoi d'un message entre chaque processus nécessite le même temps.

Nous utilisons les résultats qui ont été validés dans le modèle de l'envoi/réception simultané [BNK94] dans lequel les processus peuvent recevoir et envoyer un et un seul message par ronde. Il a été montré que, dans ce modèle, les arbres binaires sont optimaux en latence et en débit dans le cas 1-N : le nombre de rondes nécessaires pour disséminer un message aux processus de l'arbre est $\lceil log\ n \rceil$; par ailleurs, en choisissant correctement les différents arbres utilisés pour les envois successifs de messages, il est possible d'obtenir que chaque processus puisse délivrer un message à chaque ronde. Notons que cela nécessite néanmoins d'avoir plusieurs processus qui jouent simultanément le rôle de *backup*. En conséquence, le protocole présenté dans la section 11.2 doit être légèrement modifié : en cas de faute, il est nécessaire de consulter l'ensemble des processus *backup* pour déterminer la liste des messages pour lesquels des acquittements sont attendus.

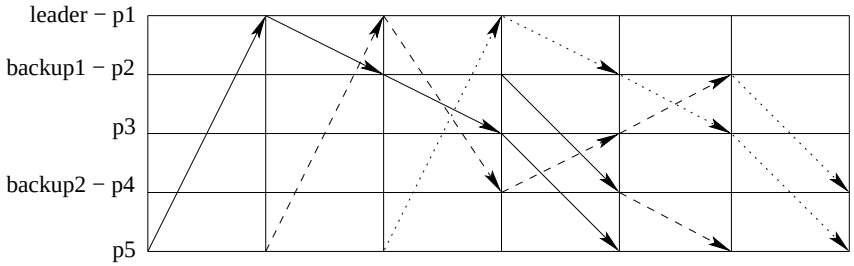

FIG. 11.4 – Diffusion de messages à cinq processus à l'aide de l'algorithme treeAlg.

La figure 11.4 représente l'exemple de la diffusion de trois messages à cinq processus à l'aide de l'algorithme treeAlg. Le processus p_5 débute les trois diffusions en envoyant un message au *leader*. Ce dernier transmet le message alternativement aux processus *backup1* et *backup2*. Les deux *backup* disséminent les messages à l'aide d'arbres couvrants n'interférant pas.

Performances

La figure 11.5 présente les résultats de tests de comparaison entre les algorithmes roundAlg et treeAlg. Les tests effectués sont des diffusions N-N de messages de 100Ko. Nous avons effectué les tests avec 5 et 9 processus.

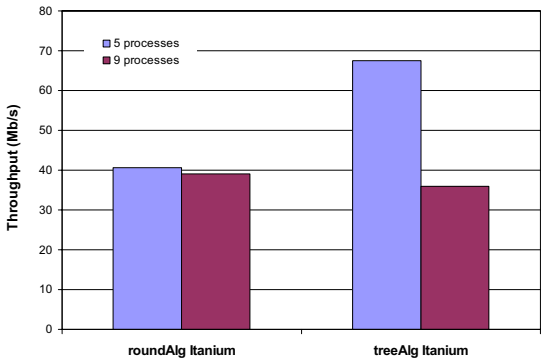

FIG. 11.5 – Comparaison des débits obtenus avec les algorithmes roundAlg et treeAlg.

Dans le test avec 5 processus, l'algorithme treeAlg offre un débit nettement supérieur à l'algorithme roundAlg. Notons néamoins que la latence minimum obtenue est moins bonne avec l'algorithme treeAlg (48ms avec 5 processus et 63ms avec 9 processus) qu'avec l'algorithme roundAlg (33ms avec 5 processus et 34ms avec 9 processus). Par ailleurs, le débit obtenu diminue considérablement dans l'expérience avec 9 processus et devient inférieur au débit obtenu par l'algorithme roundAlg. Cette chute du débit peut s'expliquer par le fait que l'hypothèse de synchronie faite pour construire un enchevêtrement d'arbres couvrants sans interférences ne tient plus lorsque le nombre de processus augmente. Notons par ailleurs, que cette observation était la même avec des tailles de messages inférieures.

11.3.3 Dissémination pragmatique

Dissémination

Dans cette section, nous présentons un mode de dissémination *pragmatique* qui consiste simplement à chaîner les processus auxquels les messages sont disséminés par le *backup*. Les communications entre processus sont réalisées à l'aide du protocole TCP. L'illustration de cette dissémination est donnée sur la figure 11.6. La dissémination est effectuée par le *backup* dans les étapes 3, 4, 5. L'étape 6 permet d'acquitter le message pour l'ensemble des processus. Un avantage indéniable de ce mode de dissémination est sa simplicité de mise en œuvre.

Le débit obtenu par cet algorithme est optimal dans le cas 1-N du fait qu'il permet de délivrer un message par ronde. En revanche, sa latence n'est pas optimale, car elle est linéaire en fonction du nombre de processus, alors que la latence optimale dans le modèle théorique utilisé est logarithmique (obtenue avec l'algorithme treeAlg).

Performances

La figure 11.7 présente la latence minimum (i.e. système hors contention) en fonction du nombre de processus pour les trois choix proposés dans le cas de diffusions N-N de messages de 100Ko sur la grappe Itanium. Les résultats obtenus sont conformes à ceux attendus. A première vue, l'algorithme roundAlg semble intéressant, car il a une latence constante. Cependant, comme

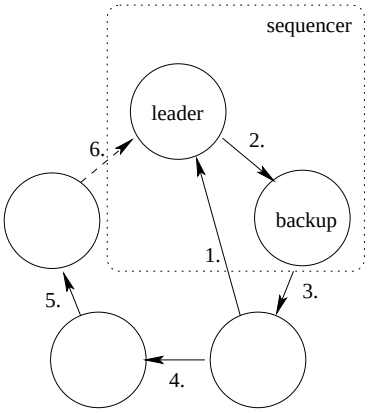

FIG. 11.6 – Fonctionnement de l'algorithme chainAlg pour 5 processus.

nous le mentionnons dans la section 11.1, la latence minimum est négligeable comparée au temps induit par l'attente dans un système dont le débit est faible.

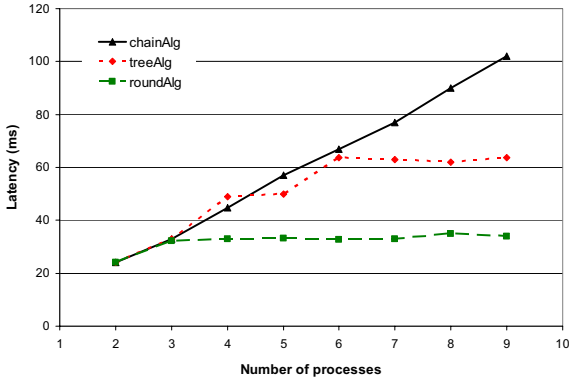

FIG. 11.7 – Latence minimum en fonction du nombre de processus.

La figure 11.8 représente le débit maximum obtenu pour chaque algorithme en fonction du nombre de processus. Le test effectué est une diffusion N-N de messages de 100Ko sur grappe Itanium. L'avantage de l'utilisation de l'algorithme chainAlg est évident : le débit obtenu est constant (il ne décroît pas comme c'est le cas de l'algorithme treeAlg) et il vaut le double de celui de roundAlg. Ces bons résultats s'expliquent par le fait que la dissémination de l'algorithme chainAlg est triviale : elle ne nécessite pas d'hypothèse forte de synchronie comme c'est le cas de l'algorithme treeAlg. Par ailleurs, elle repose sur l'utilisation de TCP dont les mécanismes de retransmission sont plus efficaces que ceux que nous avons dû mettre en œuvre pour IP multicast.

11.3.4 Récapitulatif

La figure 11.9 fournit un récapitulatif des principaux résultats sur le débit maximal obtenu par les trois choix possibles que nous proposons et par JGroups (avec protocoles IP multicast et TCP). Les tests effectués sont des diffusions (1-N et N-N) de messages de 100Ko sur les grappes Itanium et Xeon.

Il est tout d'abord intéressant de noter que les résultats de l'algorithme chainAlg sur la grappe Xeon sont bons. Rappelons que ce n'était pas le cas de l'algorithme roundAlg du fait des très

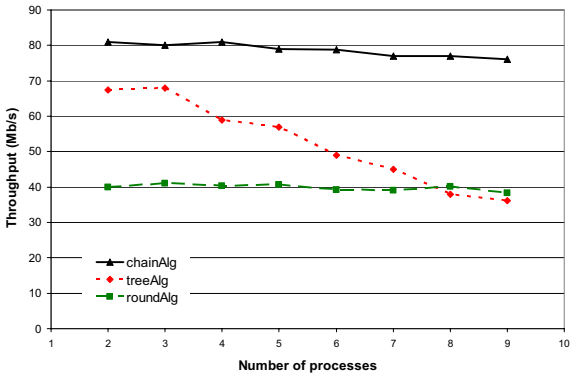

FIG. 11.8 – Débit maximum en fonction du nombre de processus.

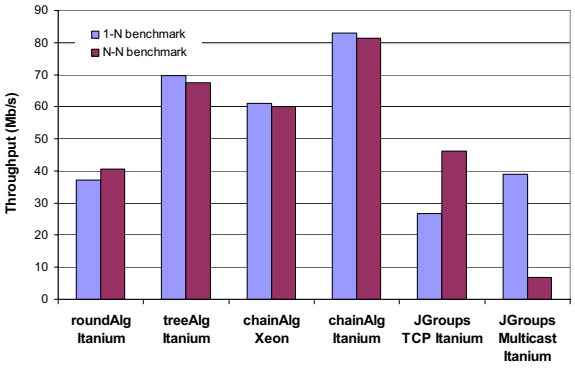

FIG. 11.9 – Récapitulatif des débits maximum obtenus par les trois algorithmes et par JGroups.

mauvaises performances du protocole IP multicast sur cette grappe. Ce résultat encourage la construction d'algorithmes basés sur TCP plutôt que sur IP multicast.

Par ailleurs, il est important de noter que le protocole que nous proposons permet d'avoir des performances similaires dans les cas 1-N et N-N. Ce n'est pas le cas de JGroups : dans le cas IP multicast, les performances diminuent considérablement en N-N du fait de l'augmentation des collisions ; au contraire, dans le cas TCP, les performances N-N sont meilleures que les performances 1-N. Ceci s'explique par le fait que le protocole de diffusion à base de TCP utilise un schéma de diffusion "naïf" : le processus émetteur du message effectue des envois unicast séquentiels à l'ensemble des processus du groupe. Ce mode de diffusion n'est bien sûr pas optimal dans le cas 1-N. En revanche, dans le cas N-N il y a une augmentation du débit liée à une meilleure utilisation de la bande passante.

11.4 Mise en œuvre

L'implantation du protocole décrit dans la section 11.2 à l'aide de DREAM est représentée sur les figures 11.10 et 11.11. Ces deux figures représentent deux sous-ensembles du même composite correspondant à deux rôles différents : la figure 11.10 illustre les composants nécessaires à un processus standard, alors que la figure 11.11 représente les composants utilisés par le *leader*. Par manque de place, nous n'avons pas représenté les composants nécessaires pour le *backup*.

11.4.1 Les composants du rôle processus

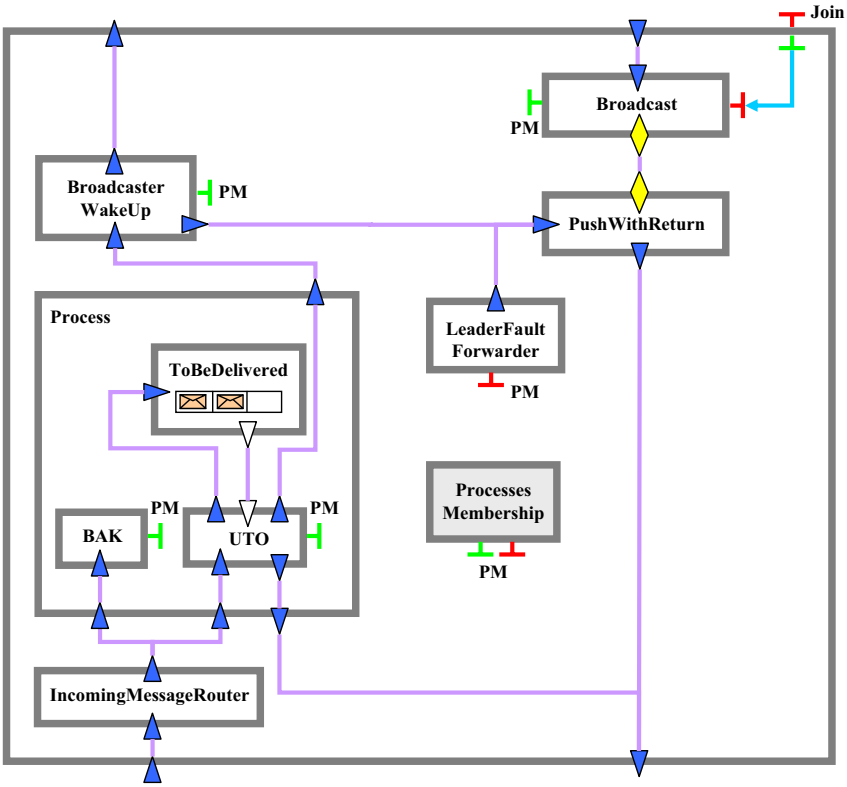

FIG. 11.10 – Architecture du composant freecast (partie processus).

Le composite possède une entrée/sortie pour les messages entrants et une entrée/sortie pour

les messages sortants. Le composant grisé `ProcessesMembership` est un composant partagé auquel les composants possédant une interface `PM` sont liés. Ce composant permet de connaître la liste des processus du groupe, ainsi que *leader* et *backup*.

La partie droite du composite représente les composants en charge des messages sortants. Ces messages transitent par les composants `Broadcast` — qui ajoute le chunk correspondant au protocole et qui ajoute un chunk de destination avec l'adresse du *leader* — et `PushWithReturn` — qui implante une interface `PushWithReturn`[5] qui permet d'envoyer des messages et de se bloquer jusqu'à ce qu'une réponse au message soit reçue. Dans le cas du protocole, le composant attend que le message lui-même soit reçu par le processus, et dans ce cas il peut le délivrer localement. C'est ce qui garantit l'uniformité de la diffusion.

La partie gauche du composite contient les composants en charge des messages entrants. Le composant `IncomingMessageRouter` est un routeur qui utilise le chunk du protocole contenu dans le message pour déterminer son type et donc le composant qui doit le traiter. Dans le cas d'un processus "standard", ce type peut être BAK où UTO. Le composant `BAK` met à jour le composant `ProcessesMembership`. Le composant `UTO` délivre les messages qu'il reçoit en garantissant que les messages sont délivrés dans l'ordre du numéro de séquence déterminé par *leader*. Pour ce faire, il utilise une file de messages `ToBeDelivered` qui permet de trier les messages par numéro de séquence. Enfin, les composants `BroadcasterWakeUp` et `LeaderFaultForwarder` sont en charge de débloquer les appels traités par le composant `PushWithReturn` lorsque le message reçu est un message local ou qu'une faute du *leader* a été détectée.

11.4.2 Les composants du rôle *leader*

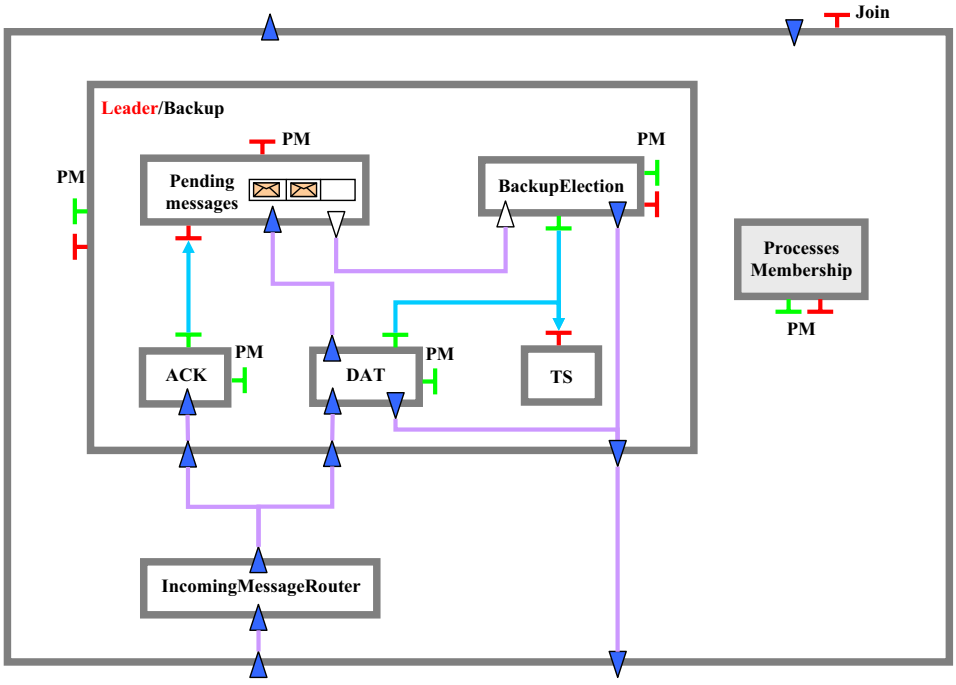

FIG. 11.11 – Architecture du composant freecast (partie *leader*).

Les composants nécessaires au processus *leader* sont représentés sur la figure 11.11. Ces composants permettent de gérer les messages entrant à destination du *leader*. Le composant

[5]Cette interface est représentée par le losange jaune.

`IncomingMessageRouter` est le même que celui représenté sur la figure 11.10. Néanmoins, il possède des liaisons vers les composants `ACK` et `DAT` qui correspondent aux deux types de messages que le *leader* est susceptible de recevoir. Le composant `DAT` reçoit des messages à diffuser. Il stocke chaque message reçu dans la file de messages `PendingMessages`, demande au composant `TS` un numéro de séquence et transmet le message au *backup*. Le composant `ACK` supprime de la file `PendingMessages` les messages pour lesquels tous les acquittements ont été reçus. Enfin, le composant `BackupElection` est en charge d'élire un nouveau *backup*, lorsqu'une défaillance du *backup* actuel est détectée.

11.4.3 Les composants du rôle *backup*

Les composants nécessaires au processus *backup* sont très semblables à ceux nécessaires au processus *leader*. La principale différence réside dans le fait que le composant `DAT` est remplacé par les composants `UPD` et `REP`. Le composant `UPD` met à jour la file de messages `PendingMessages` et le composant `ProcessesMembership`. Le composant `REP` implante la logique de dissémination. Il stocke les messages qu'il reçoit dans la file `PendingMessages` et utilise un des trois choix présentés dans ce chapitre pour disséminer les messages.

11.5 Travaux connexes

La réplication de machines à états est un sujet qui a été très étudié. Les premiers concepts de réplication ont été introduits par Lamport dans [Lam78] ; ils ont ensuite été raffinés par Schneider [Sch93]. Plus récemment, des travaux ont été consacrés à l'algorithme Paxos [Lam89]. Depuis lors, cet algorithme a été amélioré de façon à l'adapter à différents contextes, tels que l'utilisation de disques bas de gamme pour stocker l'état répliqué [CM02, GL00].

Les algorithmes de diffusion ont également été très étudiés [DSU00]. Un état de l'art des propositions existantes a été dressé dans [HT93b]. [FLP85, Lam89] proposent d'implanter la diffusion avec ordre total à partir d'un consensus. Le consensus est un problème qui a été étudié ces dernières années : [CBS00, KR01] prouvent une borne inférieure sur la complexité en temps du consensus dans un modèle purement synchrone, tandis que [DG02] donne une borne inférieure pour un système asynchrone dans lequel on suppose l'existence d'un détecteur de fautes non fiable.

Il y a également eu plusieurs implantations de logiciels de communication de groupe, certains incluant des primitives de diffusion avec ordre total [AMMS+95, Bir85, DM96, KT96, vRBM96]. Nous ne sommes assurément pas les premiers à mettre en avant cette différence entre les coûts des protocoles de communication de groupe évalués par la communauté théorique et ceux mesurés dans la pratique. Par exemple, les développeurs d'Isis [SBS91] soulignaient que la phase de dissémination était cruciale et que la diffusion logicielle n'avait absolument pas le même coût que des communications point à point.

11.6 Conclusion

Ce chapitre a présenté FREECAST, un protocole de diffusion avec ordre total uniforme. Nous avons motivé l'utilisation d'autres métriques de performances que celles généralement utilisées par la communauté d'algorithmique théorique. Nous avons ainsi montré qu'un algorithme pragmatique pouvait être beaucoup plus efficace que deux algorithmes prouvés optimaux dans leurs

contextes. Par ailleurs, nous avons présenté l'implantation de ces protocoles qui a été réalisée à l'aide de DREAM.

Remerciements : les travaux présentés dans ce chapitre ont été réalisés en collaboration avec Rachid Guerraoui, Ron Levy et Bastian Pochon, respectivement professeur et doctorants au sein du laboratoire de programmation distribuée de l'Ecole Polytechnique Fédérale de Lausanne.

Chapitre 12

Vers la construction de systèmes autonomes

Sommaire

Les différents systèmes décrits dans ce rapport ont été réalisés au sein du projet SARDES, dont l'ambition est de construire des infrastructures logicielles autonomes, c'est-à-dire capables de se reconfigurer automatiquement lors de l'occurence de certains événements (fautes logicielles et matérielles, dégradation de performances, etc.). En parallèle des travaux sur DREAM, LEWYS et FREECAST, des membres du projet SARDES ont donc débuté la réalisation de JADE, un intergiciel permettant de construire des systèmes auto-administrables intégrant à la fois des fonctions d'observation et de contrôle.

Bien qu'ayant uniquement participé aux étapes préliminaires de conception de JADE, nous avons décidé d'en présenter l'architecture afin de mettre en perspective les travaux présentés dans ce rapport. Néanmoins, l'implantation décrite dans ce chapitre ne reflète pas l'implantation actuelle de JADE qui ne repose pas sur l'utilisation des systèmes présentés dans cette thèse. Nous montrons comment DREAM et LEWYS pourraient être utilisés pour réaliser deux des fonctions de JADE : (i) l'observation du système administré, et (ii) les communications entre le système d'administration et le système administré.

Nous proposons également une contribution au système JADE : un protocole de réplication active basé sur l'utilisation de FREECAST, permettant de rendre tolérant aux fautes les boucles de commande.

L'organisation de ce chapitre est la suivante : la section 12.1 justifie le développement du système JADE ; son architecture est détaillée dans la section 12.2. La section 12.3 présente un protocole de réplication active basé sur FREECAST permettant de rendre le système d'administration tolérant aux fautes. Enfin, la section 12.4 présente les travaux connexes, avant de conclure en section 12.5.

12.1 Motivations

La structure des systèmes évolue vers une plus grand complexité et une grande hétérogénéité. L'*administration* de ces systèmes est un défi majeur. L'administration d'un système réparti couvre un ensemble de tâches variées telles que le déploiement (téléchargement, installation, configuration et lancement), la réparation en cas de panne, l'optimisation des performances, etc. Cette tâche d'administration est actuellement effectuée par l'homme, ce qui engendre un coût en ressources :

- *humaines* du fait qu'elle nécessite systématiquement une intervention humaine en réaction à des événements (pannes par exemple)
- *matérielles* car la solution généralement adoptée pour prendre en compte des incidents (pannes ou surcharge) est la surréservation de ressources

Pour réduire ces coûts, il est nécessaire de construire des systèmes autonomes [KC03] capables d'administrer les éléments qui les composent sans intervention humaine. Un système autonome peut fournir différentes propriétés, parmi lesquelles :

- le *self-configuring* qui est la capacité de paramétrer automatiquement ses composants
- le *self-healing* qui est la faculté de détecter et de corriger la panne d'un ou plusieurs composants
- le *self-optimizing* qui consiste à effectuer des mesures pour remédier aux éventuels problèmes de performances
- le *self-protecting* qui est la capacité de se prémunir de la malveillance humaine

Les outils d'administration existants se focalisent sur des aspects spécifiques tels que l'observation ou le déploiement. Par exemple, les intergiciels complexes, tels les serveurs d'applications J2EE, proposent des outils de configurations manuels souvent basés sur une vue statique du système. L'intégration des différents outils nécessite une expertise dans de nombreux domaines : systèmes d'exploitation, intergiciels, applications pour les utilisateurs, etc. Cela se traduit souvent par le développement d'outils complexes et ad hoc, uniquement adaptés aux besoins d'un scenario applicatif. Ces outils sont, de fait, très difficiles à adapter à d'autres contextes que ceux pour lesquels ils ont été développés.

Tout comme White et al. [WHW+04], l'équipe SARDES pense qu'il est nécessaire de proposer des des techniques logicielles, des patrons de conception et des principes architecturaux adaptés à la construction de systèmes autonomes. L'objectif est de parvenir à un découplage aussi fort que possible entre les fonctions d'administration et les caractéristiques d'implantation du système administré. Pour cela, l'approche proposée consiste à utiliser une architecture modulaire permettant de créer de façon systématique des boucles de commande. Ces boucles sont constituées de divers composants : les *capteurs* collectent de l'information sur le système administré ; cette information est véhiculée par des canaux à un *composant gestionnaire* qui choisit les actions à effectuer sur le système ; ces actions sont effectuées par des composants *actionneurs*.

Dans ce chapitre, nous présentons une approche à composants pour la construction de telles boucles. Le système présenté repose sur l'utilisation de FRACTAL, DREAM, LEWYS et FREECAST. L'approche décrite est modulaire, dynamiquement configurable et facilement réutilisable.

12.2 Architecture d'une boucle de commande

Nous effectuons tout d'abord une présentation générale de la structure des boucles de commande. Nous présentons ensuite les différents composants qui les constituent.

12.2.1 Présentation générale

La structure générale des boucles de commande est représentée sur la figure 12.1. Cette structure est semblable à celle des boucles de commande utilisées dans la théorie de la commande [Oga97]. Ces boucles sont en charge de la régulation et de l'optimisation du comportement du système administré. Ce sont des boucles fermées dans le sens où le comportement du système administré est influencé à la fois par des entrées opérationnelles (fournies par les clients du système) et par des entrées de contrôle (fournies par le système de gestion en réaction à l'observation du système). Les différents composants intervenant dans la boucle sont les suivants : le *système administré*, des *capteurs* permettant de collecter des informations sur le système administré et un composant *gestionnaire* qui analyse les données collectées par les capteurs et ordonne à des *actionneurs* d'effectuer des opérations sur le système administré.

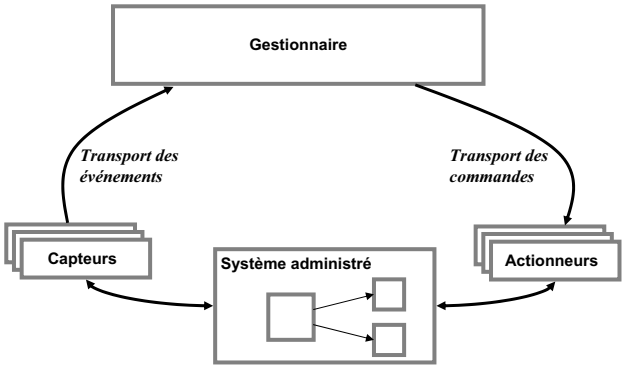

FIG. 12.1 – Structure d'une boucle de commande.

12.2.2 Le système administré et les actionneurs

Le système administré est modélisé à l'aide du modèle de composants FRACTAL. Ce dernier permet de représenter des architectures de composants hiérarchiques avec liaisons explicites entre les composants. Notons que le système peut être soit développé à l'aide d'une implantation du modèle, soit encapsulé dans des composants FRACTAL.

L'intérêt de l'utilisation de FRACTAL réside dans le fait qu'il est possible d'associer aux composants du système administré un nombre arbitraire d'interfaces de contrôle. Il est ainsi possible de créer des actionneurs à l'aide de contrôleurs. Ces actionneurs sont à même d'effectuer des actions de base sur les composants du système administré. Nous appelons *action de base*, une action qui ne fait intervenir aucune décision complexe dans sa réalisation — au moins du point de vue du système d'administration. Les actionneurs fournissent ainsi des mécanismes de base qui sont utilisés par les composants du système d'administration qui peuvent, eux, implanter des politiques complexes.

Les actions qui peuvent être réalisées dépendent des contrôleurs fournis par le système administré. Parmi les exemples d'actions de base, citons :

– les actions sur le cycle de vie : démarrer/stopper l'exécution d'un composant.

– les actions de configuration : ajout/retrait d'un sous-composant d'un composite, création/-destruction d'une liaison, modification du code d'un composant, etc.

12.2.3 Les capteurs

Le rôle des capteurs est d'observer les changements d'état du système. Les capteurs génèrent des événements de supervision qui sont transmis au composant gestionnaire. Les capteurs et le système de transport des événements de supervision peuvent être implantés à l'aide de LeWYS. Comme nous l'avons présenté au chapitre 10, LeWYS est un canevas logiciel à composants dédié à la construction d'applications de supervision. LeWYS permet de déployer une *pompe* sur chacun des nœuds sur lesquels le système à admininistrer est déployé. Cette pompe permet de déployer des *sondes* générant des informations de supervision. L'un des intérêts de LeWYS réside dans le fait qu'il est aussi bien possible de déployer des *sondes* générant des informations sur les ressources du système (e.g. CPU, disque, réseau) que des sondes ad hoc, adaptées au système administré. Une telle sonde peut, par exemple, intéragir avec des contrôleurs du système administré pour collecter des données applicatives (e.g. nombre de requêtes en cours de traitement, taille de tampons, nombre d'erreurs). Ces sondes peuvent également être utilisées pour détecter la faute de composants où la panne de nœuds.

LeWYS utilise Dream pour la propagation des événements générés par les sondes. Il est ainsi possible de construire des canaux de propagation des événements arbitrairement complexes. Ces canaux peuvent notamment fournir différentes garanties sur la transmission de ces événements. Par exemple, certains événements (e.g. occurence de fautes) peuvent être transmis de façon fiable, alors que d'autres événements (e.g. consommation CPU) peuvent ne faire l'objet d'aucune garantie de délivrance. Il est également possible d'insérer des composants de traitement des événements au sein des canaux. Par exemple, il peut être intéressant d'introduire des filtres permettant de limiter le nombre d'événements transmis.

12.2.4 Le gestionnaire

Le gestionnaire implante l'étage d'analyse et de décision de la boucle de commande. Il contient plusieurs sous-composants en charge de différents aspects de l'administration : gestion des nœuds, déploiement, tolérance aux fautes, optimisation des performances, etc. Ces différents composants utilisent un composant particulier, appelé *représentation du système*, dont le rôle est de maintenir une représentation globale du système (système administré et système d'administration) qui est isomorphe, introspectable et causalement connectée aux composants en cours d'exécution. Cette représentation est implantée par une architecture Fractal qui manifeste les mêmes liaisons et relations d'encapsulation que les composants du système administré. La connexion entre le système administré et sa représentation peut être effectuée à l'aide de liaisons distribuées implantées à l'aide de la bibliothèque Dream.

Le composant de représentation du système encapsule un ensemble de *méta-composants*. Un *méta-composant* a les mêmes interfaces, liaisons, attributs et configuration interne (en termes de *méta-composants*) que le composant qu'il réifie. Par ailleurs, chacun de ses contrôleurs implante une connexion causale vers le composant réifié. Par exemple, l'appel d'une opération de liaison sur le contrôleur de liaisons du *méta-composant* est répercuté sur le contrôleur de liaisons du composant réifié.

Le gestionnaire contient obligatoirement un composant de déploiement (appelé *déployeur*) qui utilise l'usine Fractal ADL présentée au chapitre 6. Ce composant utilise une description ADL du système administré et du système d'administration et procède à leur déploiement. Par

ailleurs, l'usine ADL a été étendue de façon à générer le composant de représentation du système à partir de la description du système lui-même.

Les communications entre les différents composants d'administration du gestionnaire et le système administré peuvent être implantées à l'aide de canaux DREAM. Comme dans le cas des capteurs, l'intérêt de l'utilisation de DREAM réside dans le fait qu'il est possible de construire des canaux implantant différentes sémantiques de communication. Par exemple, il est possible de construire des canaux implantant une intéraction synchrone afin de procéder à la reconfiguration d'un composant. Il est également possible d'utiliser un canal implantant une communication de groupe fiable pour ordonner l'arrêt de l'exécution d'un ensemble de composants.

12.2.5 Exemple d'une boucle de commande pour la tolérance aux fautes

Supposons que l'on souhaite construire une boucle de commande ayant pour but de tolérer les pannes franches des nœuds hébergeant une application distribuée. Une telle boucle de commande fait intervenir des capteurs de détection de fautes et des actionneurs permettant des actions sur le cycle de vie ainsi que des capteurs en charge d'ajouter et de retrancher des composants (afin de remplacer les composants fautifs). Par ailleurs, cette boucle doit contenir un gestionnaire encapsulant des composants implantant un algorithme de tolérance aux fautes. L'architecture d'un tel gestionnaire est représentée sur la figure 12.2. Celui-ci encapsule différents sous-composants :

- Les composants **représentation du système** et **déployeur** décrits dans le paragraphe précédent.
- Le composant **gestionnaire de fautes** implante la politique de tolérance aux fautes. Une politique simple consiste (1) à rompre les liaisons vers le composant fautif, (2) à remplacer ce dernier, et (3) à établir des liaisons vers le nouveau composant. Notons qu'une telle politique ne se préoccupe pas de l'état des composants fautifs. Il est possible de mettre en place des politiques le prenant en compte si le système administré y donne accès.
- Le composant **console** fournit une console d'observation et un interprète de script permettant à un administrateur de contrôler l'état du système et d'effectuer manuellement les opérations de reconfiguration effectuées par le gestionnaire de fautes.

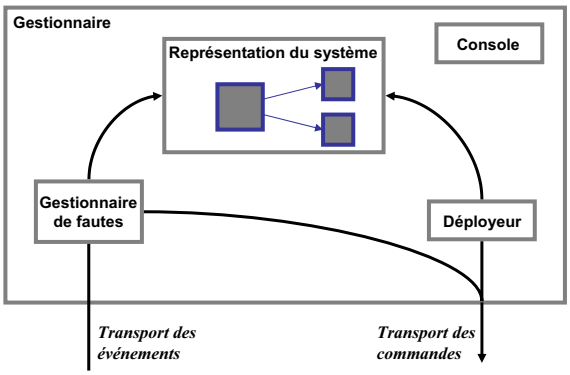

FIG. 12.2 – Structure interne du composant gestionnaire.

12.3 Réplication des boucles de commande

La structure de boucle de commande présentée dans la section précédente peut être sujette à des fautes survenant au niveau du gestionnaire. Dans cette section, nous proposons un algorithme

de réplication active permettant de rendre tolérant aux fautes ce dernier.

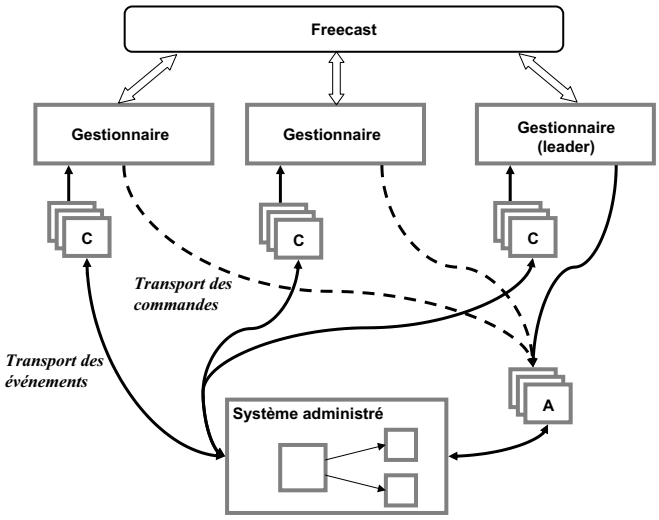

FIG. 12.3 – Réplication active d'une boucle de commande.

La structure de réplication, représentée sur la figure 12.3, est obtenue :

1. en répliquant de façon active sur différents nœuds les composants du système d'administration qui ne nécessitent pas d'être localisés sur les nœuds applicatifs[1]. Ces composants sont le gestionnaire et certains des capteurs, tels les capteurs de détection de fautes.
2. en s'assurant que les composants de représentation du système contiennent chacun une représentation des composants d'administration répliqués.

Le second point est trivial à obtenir. En revanche, il y plusieurs précautions à prendre pour mettre en œuvre le premier point : un simple mécanisme de réplication active n'est pas suffisant pour garantir que le système d'administration est tolérant aux fautes. Il y a plusieurs précautions à prendre quant au déroulement des actions effectuées par les différents composants.

Le protocole de réplication active peut être implanté à l'aide de FREECAST. Comme nous l'avons vu dans le chapitre 11, ce protocole distingue deux processus particuliers : *leader* et *backup*. Afin de garantir un état cohérent des composants gestionnaire répliqués, il est nécessaire que ces derniers sachent s'ils sont *leader* ou non. La réplication du composant gestionnaire peut alors être faite en suivant le protocole suivant :

- le composant gestionnaire et les capteurs sont répliqués sur différents nœuds. Le nombre de réplicas dépend du niveau de tolérance aux fautes requis : f fautes peuvent être tolérées à l'aide de $f + 1$ réplicas.
- les gestionnaires diffusent aux autres réplicas chaque événement reçu des capteurs, ainsi que chaque commande ordonnée aux actionneurs. Les gestionnaires forment un groupe de diffusion qui est mis à jour dynamiquement par le protocole FREECAST lorsqu'un gestionnaire tombe en panne.
- seul le composant gestionnaire du nœud œuvrant comme *leader* du protocole de communication de groupe agit lors de la réception d'événements de la part des capteurs. Ceci est représenté sur la figure 12.3 par le fait que seul le gestionnaire du nœud *leader* possède une liaison en trait plein vers les composants actionneurs.

[1] Les composants d'administration nécessitant d'être localisés sur les nœuds applicatifs (e.g. actionneurs) ne peuvent pas, par nature, être répliqués.

- chaque paire (événement d'un capteur, commande résultante) se voit assigner un numéro de séquence unique journalisé au niveau de chaque réplica. Cela permet, par exemple, de terminer une action d'administration inachevée lors de l'élection d'un nouveau *leader*.
- le numéro de séquence associé à chaque paire (événement d'un capteur, commande résultante) est transmis comme paramètre additionnel de chaque commande, afin que les actionneurs n'exécutent chaque commande qu'une fois, et une seule. Cela est nécessaire pour éviter d'éxécuter deux fois des commandes au cours de la fenêtre de vulnérabilité existant entre la terminaison d'une commande et la faute du gestionnaire *leader*.

12.4 Travaux connexes

Il existe de nombreux travaux qui sont connexes à JADE : l'exploitation de connaissances sur l'architecture logicielle pour la construction de systèmes auto-réparables, l'administration de systèmes s'exécutant sur des supports à grande échelle et sur des grappes de machines et les systèmes de gestion de configurations distribuées.

Exploitation de connaissances sur l'architecture logicielle pour la construction de systèmes auto-réparables. L'architecture de tolérance aux fautes proposée dans ce chapitre est un exemple de système basé sur une connaissance explicite de l'architecture logicielle du système administré [DvdHT02]. En effet, les fonctions d'administration exploitent une représentation explicite et causalement connectée du système administré. Il y a différents travaux qui ont proposé une exploitation des connaissances de l'architecture logicielle [OGT+99, GMK02, BCB+02]. Parmi ces travaux, celui qui est le plus proche de JADE est [GMK02] qui se base sur le langage de description d'architectures Darwin[2]. JADE se distingue par plusieurs caractéristiques : tout d'abord, il repose sur l'utilisation du modèle de composants FRACTAL qui est plus flexible que Darwin et qui facilite la mise en œuvre des reconfigurations dynamiques. Par ailleurs, JADE présente une structure réflexive dans laquelle les composants de l'infrastructure d'administration (capteurs, actionneurs, composants du gestionnaire) sont eux-mêmes des composants et peuvent être administrés de la même façon que le système lui-même. Ce n'est pas le cas dans le système basé sur Darwin. Enfin, tout comme JADE, le système basé sur Darwin utilise une représentation du système ; néanmoins, JADE ne maintient pas cette représentation sur tous les nœuds, ce qui réduit les interférences entre le système administré et le système d'administration et réduit les problèmes de passage à l'échelle dont le système basé sur Darwin souffre.

Administration de systèmes s'exécutant sur des supports à grande échelle et sur des grappes de machines. De nombreux travaux existent sur l'administration de systèmes s'exécutant sur des infrastructures à grande échelle et sur des grappes de machines. Une grande part d'entre eux [ADZ00, CIG+03, AFF+01, FCC+03] se focalisent sur le problème de la gestion des ressources et non sur celui de la tolérance aux fautes. Le système le plus proche de JADE est BioOpera [BPSA02] qui propose une représentation explicite du système administré (sous la forme d'un workflow instrumenté représentant les tâches applicatives et des informations sur l'état des nœuds) et un système de recouvrement après panne, basé sur des points de reprises des tâches. Il y a de nombreuses différences entre JADE et BioOpera. Tout d'abord, BioOpera ne traite que des applications sous forme de workflow. Par ailleurs, contrairement à JADE qui permet de prendre en compte des changements arbitraires de configurations, BioOpera ne permet que l'introduction de nouveaux workflow et ne traite que les fautes de nœuds ou de workflow.

[2]Darwin est présenté au chapitre 3.

Enfin, le système d'administration de BioOpera n'est pas tolérant aux fautes, du fait qu'il repose sur l'utilisation d'une base de données centralisée.

Systèmes de gestion de configurations distribuées. Des travaux de recherche ont été menés sur la gestion de configurations distribuées [SW98, BFPDR02, PBJ98, Sal02]. Le travail le plus proche de JADE est [KC00][3] qui présente un modèle permettant de réifier les dépendances entre composants d'un système distribué. Les auteurs considèrent deux types de dépendances : les *pré-requis* sont les dépendances d'un composant envers des composants matériels et logiciels permanents dans le système ; les *dépendances dynamiques* sont les dépendances entre les composants déployés. Chaque composant est géré par un *configurateur de composants* qui permet d'assurer que les dépendances d'un composant sont satisfaites à la fois lors de son déploiement et lors de reconfigurations. JADE diffère de ces travaux par trois caractéristiques principales :

- le composant de représentation du système utilisé dans JADE a un pouvoir d'expressivité supérieur ; outre les dépendances entre composants — qui sont réifiées sous forme de liaisons FRACTAL —, la représentation du système permet de décrire les attributs des composants et pourrait facilement être étendue pour supporter une description du comportement des composants.
- JADE étant développé à l'aide du modèle de composants FRACTAL, il est possible de représenter de façon uniforme des architectures hiérarchiques complexes faisant intervenir des composants de bas niveau (système d'exploitation), des composants intergiciels (e.g. serveur J2EE), et des composants applicatifs (e.g. EJB s'exécutant dans le serveur J2EE). Il est ainsi possible d'administrer le système à différents niveaux de granularité.
- JADE permet d'appliquer les reconfigurations implantées par les composants gestionnaires aux gestionnaires eux-mêmes, ce qui n'est pas le cas des *configurateurs de composants* qui ne peuvent pas être reconfigurés dynamiquement. Cela permet de modifier dynamiquement une politique d'administration qui s'avère insuffisante.

12.5 Conclusion

Ce chapitre avait pour objet JADE, un intergiciel développé au sein du projet SARDES pour la construction de systèmes d'administration d'applications distribuées. Bien qu'il n'ait pas été développé dans le cadre de cette thèse, nous avons présenté JADE afin de montrer comment les différents systèmes décrits dans cette thèse pourraient être utilisés pour construire des systèmes autonomes. L'implantation actuelle de JADE n'utilise pas ces systèmes. Néanmoins, nous avons montré comment ils pourraient l'être : DREAM peut être utilisé pour construire des canaux de communication entre capteurs/actionneurs et composants gestionnaires ; LEWYS peut être utilisé pour implanter les capteurs d'observation des ressources du systèmes et de détection de fautes. Par ailleurs, nous avons proposé un protocole de réplication active permettant de rendre tolérant aux fautes les boucles de commande. Un tel algorithme n'existe pas, aujourd'hui, dans JADE ; il pourrait être implanté à l'aide de FREECAST.

[3] Ce système de gestion de dépendances est utilisé dans l'intergiciel *dynamic*TAO qui est présenté au chapitre 4.

Conclusion

Dans ce chapitre qui constitue la conclusion de notre travail, nous dressons un bilan des principaux apports de nos travaux, puis nous en présentons les perspectives.

Principaux apports

L'augmentation du nombre et de l'hétérogénéité des équipements — processeurs, infrastructures réseaux, etc. — intervenant dans les systèmes informatiques modernes nécessite de construire des **architectures logicielles autonomes** capables de se reconfigurer dynamiquement lors de la survenue de certains événements tels qu'une panne de machine, une dégradation des performances, etc. Cette autonomie des systèmes permet d'automatiser les tâches d'administration traditionnellement effectuées par l'homme, ce qui engendre une diminution des coûts d'exploitation des applications et une amélioration de leur disponibilité.

La construction de systèmes autonomes nécessite :

- d'une part, de disposer d'une technologie logicielle permettant de développer des systèmes administrables.
- d'autre part, de posséder la faculté de construire des boucles de contrôle comprenant des éléments d'observation, d'analyse et de contrôle.

Les deux sections suivantes rappellent nos principales contributions concernant ces deux aspects.

Technologie logicielle pour la construction de systèmes administrables

Dans ce rapport, nous avons soutenu la thèse selon laquelle il était nécessaire de construire des infrastructures logicielles radicalement configurables. Pour cela, nous avons proposé une nouvelle démarche, appelée *exogiciel*, inspirée de la philosophie exo-noyaux [EKO95]. Cette démarche consiste à utiliser des canevas logiciels à composants comprenant :

- un *modèle de composants réflexif* permettant de construire des architectures logicielles distribuées. Nous avons insisté sur la nécessité de pouvoir construire des architectures hiérarchiques et de pouvoir programmer un méta-niveau arbitrairement complexe.
- une *bibliothèque de composants* contenant divers composants à partir desquels on peut constuire des systèmes complexes.
- un *ensemble d'outils de gestion de configuration* permettant de décrire, déployer et administrer des systèmes réalisés à l'aide de la bibliothèque de composants.

Nous avons prouvé que cette démarche était intéressante à travers l'implantation de **Dream**, un canevas logiciel à composants destiné à la construction d'intergiciels de communication. Dream est un premier exemple de logiciel construit suivant la philosophie exogiciel. Il rassemble différents concepts qui n'avaient pas ou peu été intégrés jusqu'à présent. Ainsi, Dream combine l'utilisation d'un modèle de composants novateur — par ses possibilités de programmation

du méta-niveau et son modèle de composition étendu — et d'outils de gestion de configuration couvrant, outre le déploiement et la configuration des intergiciels, la vérification statique de leur architecture et leur reconfiguration dynamique. Ces deux technologies ont été mises à profit pour la construction d'intergiciels de communication, offrant un gain en configurabilité sensible sans pertes de performances. Nous avons illustré l'apport de DREAM à travers la reconstruction et l'extension du canevas SEDA de construction de serveurs de communicaton haute performance, et de l'intergiciel asynchrone JORAM.

Notons, par ailleurs, que dans le cadre de DREAM, nous avons réalisé un ensemble d'extensions au modèle de composants FRACTAL ainsi qu'à la chaîne d'outils qui le mettent en œuvre. Ces extensions permettent, notamment, une gestion au méta-niveau des ressources utilisées dans une structure logicielle FRACTAL quelconque, en découplant le code fonctionnel des composants des politiques de gestion de ressources proprement dites.

Boucles de commande pour l'administration autonome de systèmes

La seconde partie de ce rapport a été consacrée à la description de trois logiciels permettant la mise en place de boucles de commande pour l'administration autonome de systèmes. Nous avons montré que la philosophie exogiciel constituait une base intéressante pour la construction des divers composants intervenant dans la boucle de commande. Nous avons ainsi présenté :

LEWYS, un canevas logiciel à composants pour la construction de systèmes de supervision. LEWYS permet de déployer des systèmes de supervision adaptés à diverses tailles de systèmes et à divers besoins d'administration. Il autorise le déploiement de sondes de collecte d'événements et la construction de canaux de propagation et de traitement de ces événements arbitrairement complexes. Ces canaux sont contruits à l'aide de DREAM. Par ailleurs, les systèmes d'observation bâtis à l'aide de LEWYS sont dynamiquement configurables. Il est ainsi possible de changer dynamiquement l'ensemble des indicateurs qui sont collectés, les traitements qui sont effectués sur les données, ou encore leur moyen d'acheminement vers les destinataires. Notons enfin que les composants de LEWYS ont été conçus de manière à minimiser l'instrusivité des applications de supervision.

FREECAST, un protocole de diffusion de groupe avec ordre total uniforme implanté à l'aide de DREAM. Ce protocole est nécessaire pour réaliser une réplication active des composants intervenant dans les boucles de commande afin de rendre ces dernières tolérantes aux fautes. Bien que le sujet des protocoles de communication de groupe fasse l'objet d'une publication abondante, le protocole que nous avons proposé est à la fois plus simple d'implantation et plus efficace en pratique que des protocoles prouvés théoriquement optimaux vis-à-vis de la latence et du débit atteignable. Il démontre ainsi l'intérêt d'une analyse expérimentale permettant de confirmer (ou, en l'occurence, d'infirmer) des hypothèses d'environnement cruciales pour la performance d'algorithmes répartis.

Par ailleurs, nous avons décrit **JADE**, un intergiciel permettant la construction de boucles de commande. JADE n'a pas été développé dans le cadre de cette thèse et, par conséquent, ne repose pas sur l'utilisation des systèmes présentés précédemment. Néanmoins, nous avons montré comment LEWYS et DREAM pourraient être utilisés pour prendre en charge certaines de ces fonctions. Par ailleurs, nous avons présenté une contribution au système JADE : un protocole de réplication active permettant de rendre tolérantes aux fautes les boucles de commande. Ce protocole pourrait être implanté à l'aide de FREECAST.

Perspectives

Les travaux que nous avons effectués durant cette thèse ouvrent différentes perspectives que nous listons dans cette section.

Modèle de composants

Dans le cadre de cette thèse, nous avons utilisé et étendu le modèle de composants FRACTAL. Voici un certain nombre d'axes de recherches qui peuvent être poursuivis pour améliorer ce modèle.

Support pour la persistance Le modèle de composants FRACTAL ne fournit actuellement aucun mécanisme pour rendre un composant persistant. Une possibilité est de développer un contrôleur permettant de sauvegarder et de restaurer l'état d'un composant (primitif ou composite). Ce contrôleur peut s'interfacer avec différents supports de persistance : base de données, système de fichiers, base de données à objets, etc. Cet interfaçage peut être réalisé en utilisant un intergiciel approprié, tel JORM [jor05b].

Contrôleurs sous forme de composants Dans l'implantation Java de référence du modèle FRACTAL, les contrôleurs sont développés à l'aide de classes *mixins*. L'intérêt de ce mécanisme est d'autoriser la création de contrôleurs par fusion de fragments de classes implantant des aspects de contrôle déterminés. Si ce mécanisme est approprié pour les contrôleurs simples, il devient difficile à utiliser dès lors que les contrôleurs développés se compliquent. Une évolution souhaitable de l'implantation de FRACTAL serait de pouvoir développer des contrôleurs sous forme de composants. Il serait ainsi possible de bénéficier des mêmes abstractions d'architecture que pour la programmation des composants fonctionnels : liaison et contenance. Notons que, outre une modification du support d'exécution de FRACTAL, cette proposition nécessite une modification de l'ADL afin de permettre la description de tels contrôleurs.

Intégration des techniques de programmation à composants et de la programmation orientée aspects (AOP) L'AOP est un outil largement utilisé aujourd'hui pour garantir une programmation respectant la séparation des préoccupations. Bien qu'ayant des objectifs très similaires, AOP et programmation à composants ont rarement été utilisées conjointement. Nous pensons qu'une réalisation intéressante serait de combiner l'utilisation de l'AOP, des composants et des mécanismes d'annotation fournis par la version 5 du kit de développement Java (JDK). Les annotations peuvent être utilisées pour personnaliser les interfaces des composants. Des aspects peuvent ensuite *consommer* ces annotations, afin de traiter une préoccupation orthogonale à l'application. Un exemple de telle préoccupation est la détection des relations de causalité existant entre les interactions des différents composants.

Outils d'administration

Dans ce rapport, nous avons présenté trois outils permettant d'effectuer, respectivement, des vérifications de type sur les architectures, des reconfigurations d'implantation et des reconfigurations de structure. Nous proposons trois pistes de recherche pour étendre cette suite d'outils.

Déploiement Le déploiement réalisé par l'usine Fractal ADL est synchrone et centralisé : la machine sur laquelle l'ordre de déploiement est donné exécute une séquence d'actions (éventuellement distribuées à l'aide d'appels de procédure distants) dont le résultat est l'activation de l'application déployée. Une piste de recherche serait de développer une bibliothèque de composants pouvant être ajoutés à l'usine pour modifier la sémantique du déploiement. Des exemples de modification de sémantique sont les suivants : le traitement des pannes, l'aspect transactionnel du déploiement, sa gestion décentralisée, ou encore son exécution asynchrone.

Reconfiguration avec prise en compte des contraintes d'intégrité L'outil de reconfiguration de structure présenté dans ce rapport permet d'effectuer des ajouts de composants et de liaisons à des architectures en cours d'exécution. Cet outil est un mécanisme de base qui ne garantit pas de conservation de l'intégrité de l'application reconfigurée. Nous pensons qu'il est nécessaire de pouvoir décrire des invariants dans les architectures, afin que ceux-ci soient garantis lors des reconfigurations. Par exemple, dans le cadre de Dream, des invariants pourraient être utilisés pour spécifier que la reconfiguration d'un protocole sur un site doit engendrer une modification du protocole sur les autres sites utilisant le même protocole. Le respect de ces invariants peut être à la charge de composants composites encapsulant les différents composants sujets de l'invariant.

Usine à liaisons pour Fractal L'établissement d'un connecteur Dream (e.g. topic, bus à message, RPC asynchrone, etc.) entre des composants Fractal doit être fait de façon programmatique par le développeur de l'application. Il est également à la charge du développeur de configurer le connecteur. Par exemple, dans le cas d'un bus de message garantissant un ordonnancement causal des messages véhiculés par le bus, le développeur doit fournir des fichiers utilisés pour mettre en place la configuration initiale des horloges logiques utilisées pour l'ordonnancement. Un apport intéressant pour Fractal serait de construire une usine de liaisons, intégrée à l'usine ADL, et à même de mettre en place et de configurer automatiquement ces connecteurs. Cette usine reposerait sur la spécification, via la description ADL de l'application, des connecteurs à mettre en place. Comme nous l'avons montré dans [QB02, QBL02, QC03], la configuration des connecteurs peut elle même être réalisée de façon automatique en analysant la description des composants à connecter.

Bibliothèque de composants Dream

Nous avons présenté une bibliothèque de composants dédiée à la construction d'intergiciels de communication. Deux travaux permettraient d'améliorer cette bibliothèque.

Extension de la bibliothèque La bibliothèque présentée comprend de nombreux composants. Néanmoins, il en reste un nombre important à implanter pour fournir une bibliothèque (presque !) complète : protocoles de diffusion probabiliste, protocoles de communication de groupes, protocoles d'appels de procédures à distance, etc. Le développement de ces composants soulève les mêmes défis que ceux soulevés par le développement des autres composants : il est nécessaire de choisir un grain de composants qui permet une configuration aisée des fonctions potentiellement offertes par le composant. Par ailleurs, les propriétés non fonctionnelles de ces composants doivent pouvoir être insérées/retirées (éventuellement) dynamiquement. Enfin, il faut veiller à ce que configurabilité et dynamisme ne soient pas obtenus au détriment des performances.

Support pour les transactions Dans les travaux présentés dans ce rapport, nous n'avons pas abordé la problématique des échanges de messages transactionnels. Un travail intéressant consisterait à étudier l'intégration des transactions au sein des personnalités construites à l'aide de la bibliothèque DREAM. L'enjeu sera de parvenir à une intégration aussi transparente que possible, l'idéal étant de pouvoir rendre une personnalité transactionnelle en modifiant un minimum de composants. Un moyen pour y parvenir serait d'utiliser des files de messages persistantes et de rendre transactionnels les échanges entre files de messages.

Divers

Cette section présente les perspectives à envisager pour deux travaux qui ont été initiés dans le cadre de cette thèse.

Modification dynamique des modèles de concurrence Nous avons montré dans ce document que DREAM permettait de modifier statiquement les modèles de concurrence utilisés dans les personnalités construites à l'aide de la bibliothèque de composants. Cette modification s'effectue par ajout/retrait de files de messages actives utilisant un pool de threads pour transmettre les messages reçus sur leur entrée. L'introduction d'une file de message a pour effet de désynchroniser l'exécution des composants producteurs de messages et des composants consommateurs de messages. Dans le cadre du serveur HTTP construit à l'aide de la version DREAM de SEDA, nous avons montré qu'une version mono-threadée du serveur (i.e. un seul flot d'exécution traversant l'ensemble des étages) avait de meilleures performances qu'une version multi-threadée (i.e. chaque étage possède son propre flot d'exécution) lorsque le serveur n'est pas en surcharge. Ce résultat montre que la désynchronisation de l'exécution des différents étages n'est bénéfique que dans les périodes de surcharge. Une perspective de recherche intéressante serait de construire des mécanismes permettant de détecter ces périodes de surcharge et d'introduire/retrancher dynamiquement des files de messages au sein d'une personnalité afin d'améliorer les performances en découplant l'exécution des différents composants.

Approche pragmatique de l'algorithmique distribuée Les travaux que nous avons conduits sur FREECAST ont prouvé qu'il était nécessaire de considérer d'autres métriques de performances que celles généralement utilisées par la communauté théorique. Une perspective de recherche serait de définir un modèle de complexité *pragmatique* plus proche de la réalité que les modèles disponibles. Ce modèle devra à la fois prendre en considération les métriques traditionnelles (nombre de messages, latence) et le débit — qui est souvent négligé. Ce modèle pourra ensuite être utilisé pour proposer de nouvelles versions des différents algorithmes destinés à la construction de systèmes distribués fiables : consensus, agrément, etc.

Bibliographie

[ABM87] N. Alon, A. Barak, and U. Manber, *On Disseminating Information Reliably without Broadcasting*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS'87) (Berlin, Germany), September 1987, pp. 74–81.

[ACL04] T. Abdellatif, E. Cecchet, and R. Lachaize, *Evaluation of a Group Communication Middleware for Clustered J2EE Application Servers*, Proceedings of the Symposium on Distributed Objects and Applications (DOA'04) (Agia Napa, Cyprus), October 2004, pp. 1571–1589.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin, *Architectural Reasoning in ArchJava*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02) (Malaga, Spain), june 2002, pp. 334–367.

[ADZ00] M. Aron, P. Druschel, and W. Zwaenepoel, *Cluster Reserves : a Mechanism for Resource Management in Cluster-Based Network Servers*, Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'00) (Santa Clara, California), June 2000, pp. 90–101.

[AFF+01] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, *Oceano – SLA-Based Management of a Computing Utility*, Proceedings of the International Symposium on Integrated Network Management (IM'01) (Seattle, USA), May 2001, pp. 855–868.

[AG97] R. Allen and D. Garlan, *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology **6** (1997), no. 3, 213–249.

[AGD97] R. Allen, D. Garlan, and R. Douence, *Specifying Dynamism in Software Architectures*, Proceedings of the Workshop on Foundations of Component-Based Software Engineering (Zürich, Switzerland), September 1997, pp. 11–22.

[Agh86] G. Agha, *Actors : a Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, USA, 1986.

[ALZ00] D. Ancona, G. Lagorio, and E. Zucca, *A Smooth Extension of Java with Mixins*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00) (Sophia Antipolis and Cannes, France), June 2000, pp. 154–178.

[AMMS+95] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, *The Totem Single-Ring Ordering and Membership Protocol*, ACM Transactions on Computer Systems **13** (1995), no. 4, 311–342.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin, *Language Support for Connector Abstractions*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'03) (Darmstadt, Germany), July 2003, pp. 74–102.

[ASM02] *ASM : a Java Byte-Code Manipulation Framework*, 2002, Objectweb, `http ://www.objectweb.org/asm/`.

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury, *Architecturing and Configuring Distributed Applications with Olan*, Proceedings of the Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98) (The Lake District, UK), September 1998, pp. 241–256.

[BBC+04] S. Bouchenak, F. Boyer, E. Cecchet, S. Jean, A. Schmitt, and J.-B. Stefani, *A Component-Based Approach to Distributed System Management – A Use Case with Self-Manageable J2EE Clusters*, Proceedings of the ACM SIGOPS European Workshop (Leuven, Belgium), September 2004.

[BBI+00] G. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras, *The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms*, Proceedings of the Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00) (New York, USA), April 2000, pp. 164–184.

[BCA+01] 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.

[BCB+02] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, *Reflection, Self-Awareness and Self-Healing in OpenORB*, Proceedings of the Workshop on Self-Healing Systems (WOSS'02), 2002.

[BCL+04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, *An Open Component Model and its Support in Java*, Proceedings of the International Symposium on Component-Based Software Engineering (CBSE'04) (Edinburgh, Scotland), 2004.

[BCM03] F. Baude, D. Caromel, and M.Morel, *From Distributed Objects to Hierarchical Grid Components*, Proceedings of the Symposium on Distributed Objects and Applications (DOA'03) (Catania, Italy), November 2003.

[BCS03] E. Bruneton, T. Coupaye, and J.B. Stefani, *The Fractal Component Model*, Tech. report, Specification v2, ObjectWeb Consortium, http ://www.object.org/fractal, 2003.

[BdPF+99] L. Bellissard, N. de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte, *An Agent Plateform for Reliable Asynchronous Distributed Programming*, Proceedings of the Symposium on Reliable Distributed Systems (SRDS'99) (Lausanne, Switzerland), October 1999.

[BEA05] BEA WebLogic JMS, 2005, `http ://www.bea.com/`.

[BFPDR02] M. Blay-Fornarino, A.-M. Pinna-Dery, and M. Riveill, *Towards Dynamic Configuration of Distributed Applications*, Proceedings of the Workshop on Aspect Oriented Programming for Distributed Computing Systems, in association with ICDCS'02, 2002.

[Bir85] K. Birman, *Replication and Fault-Tolerance in the ISIS System*, Proceedings of the Symposium on Operating Systems Principles (SOSP'85) (Orcas Island, United States), 1985, pp. 79–86.

[Bis02] T. Bishop, *A Survey of Middleware*, Master's thesis, Townson University, MA, USA, August 2002.

[BNK94] A. Bar-Noy and S. Kipnis, *Broadcasting Multiple Messages in Simultaneous Send/-receive Systems*, Discrete Applied Mathematics **55** (1994), no. 2, 95–105.

[BNK97] ———, *Multiple Message Broadcasting in the Postal Model*, Networks **29** (1997), no. 1, 1–10.

[BPSA02] W. Bausch, C. Pautasso, R Schaeppi, and G. Alonso, *BioOpera : Cluster-Aware Computing*, Proceedings of the International Conference on Cluster Computing (Cluster'02), IEEE Computer Society, 2002.

[CBCP01] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, *An Efficient Component Model for the Construction of Adaptive Middleware*, Proceedings of the Conference on Distributed Systems Platforms (Middleware'01) (Heidelberg, Germany), November 2001, pp. 160–178.

[CBGM03] J. Corwin, D. Bacon, D. Grove, and C. Murthy, *MJ : a Rational Module System for Java and its Applications*, Proceedings of the International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'03) (Anaheim, California, USA), 2003.

[CBS00] B. Charron-Bost and A. Schiper, *Uniform Consensus Harder than Consensus*, Tech. Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[CELQ04] E. Cecchet, H. Elmeleegy, O. Layaïda, and V. Quéma, *Implementing Probes for J2EE Cluster Monitoring*, Proceedings of the International Workshop on Component and Middleware Performance, in association with OOPSLA (Vancouver, Canada), October 2004.

[CELQ05] ———, *Implementing Probes for J2EE Cluster Monitoring*, Studia Informatica Universalis **4** (2005), no. 1.

[CIG+03] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle, *Dynamic Virtual Clusters in a Grid Site Manager*, Proceedings of the International Symposium on High Performance Distributed Computing (HPDC'03) (Seattle, Washington), June 2003.

[CJCP00] I. Cardei, R. Jha, M. Cardei, and A. Pavan, *Hierarchial Architecture for Real-Time Adaptive Resource Management*, Proceedings of the Conference on Distributed Systems and Platforms (Middleware '00) (New York, USA), 2000, pp. 415–434.

[CLQ05] E. Cecchet, O. Layaïda, and V. Quéma, *LeWYS : un Canevas Logiciel à Composants pour Construire des Applications de Supervision*, Actes des Journées sur les Systèmes à Composants Adaptables et Extensibles (Le Croisic, France), 6–8 avril 2005.

[CM02] G. Chockler and D. Malkhi, *Active Disk : Paxos with Infinitely Many Processes*, Proceedings of the Symposium on Principles of Distributed Computing (PODC'02), July 2002.

[con] W3C consortium, *The Document Object Model (DOM)*, `http ://www.w3.org/DOM`, 2005.

[cor02] *CORBA/IIOP Specification*, OMG TC Document formal/02-06-01, 2002.

[CRW01] A. Carzaniga, D. Rosenblum, and A. Wolf, *Design and Evaluation of a Wide-Area Event Notification Service*, ACM Transactions on Computer Systems **19** (2001), no. 3.

[CSB99] S. Chatterjee, B. Sabata, and M. Brown, *Adaptive QoS Support for Distributed, Java-Based Applications*, Proceedings of the Internal Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 99), 1999, pp. 203–212.

[CT96] T. Chandra and S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM **43** (1996), no. 2, 225–267.

[DG02] P. Dutta and R. Guerraoui, *The inherent price of indulgence*, Proceedings of the 21^{st} ACM Symposium on Principles of Distributed Computing (Monterey, California), July 2002.

[DHTS99] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani, *Jonathan : an Open Distributed Processing Environment in Java*, Distributed Systems Engineering **6** (1999), no. 1, 3–12.

[DK76] F. DeRemer and H. Kron, *Programming in-the-Large Versus Programming in-the-Small*, IEEE Transactions on Software Engineering, Vol SE-2, june 1976, pp. 80–86.

[DLB04] H. Duran-Limon and G. Blair, *QoS Management specification support for multimedia middleware*, Journal of Systems and Software **72** (2004), no. 1, 1–23.

[DLBC04] H. Duran-Limon, G. Blair, and G. Coulson, *Adaptive Resource Management in Middleware : a Survey*, IEEE Distributed Systems Online **5** (2004), no. 7.

[DM96] D. Dolev and D. Malki, *The Transis Approach to High Availability Cluster Communication*, Communications of the ACM **39** (1996), no. 4, 64–70.

[DSU00] X. Défago, A. Schiper, and P. Urbán, *Totally Ordered Broadcast and Multicast Algorithms : A Comprehensive Survey*, Tech. Report DSC/2000/036, École Polytechnique Fédérale de Lausanne, Switzerland, September 2000.

[DvdHT02] E. Dashofy, A. van der Hoek, and R. Taylor, *Towards Architecture-Based Self-Healing Systems*, Proceedings of the Workshop on Self-Healing Systems, WOSS 2002, ACM, 2002.

[EGH+01] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kerrmarec, and P. Kouznetsov, *Lightweight Probabilistic Broadcast*, Int. Conf. on Dependable Systems and Networks (DSN), 2001.

[EGH+03] P. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, and P. Kouznetsov, *Lightweight Probabilistic Broadcast*, ACM Transactions on Computer Systems **21** (2003), no. 4.

[EKO95] D. Engler, F. Kaashoek, and J. O'Toole, *Exokernel : an Operating System Architecture for Application-Level Resource Management*, Proceedings of the Symposium on Operating System Principles (SOSP) (Copper Mountain Resort, USA), December 1995, pp. 251–266.

[Ent02] Enterprise JavaBeansTM Specification, Version 2.1, August 2002, Sun Microsystems, `http ://java.sun.com/products/ejb/`.

[FCC+03] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, *SHARP : an Architecture for Secure Resource Peering*, Proceedings of the Symposium on Operating Systems Principles (SOSP'03) (Bolton Landing, NY, USA), ACM Press, 2003, pp. 133–148.

[FLP85] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, Journal of the ACM **32** (1985), no. 2, 374–382.

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, *THINK : A Software Framework for Component-Based Operating System Kernels*, Proceedings of the USENIX Annual Technical Conference (Monterey, USA), June 2002, pp. 73–86.

[gan02] *Ganglia Toolkit 2.5.0*, September 2002, University of California, Berkeley, USA, http ://ganglia.sourceforge.net/.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Object-Oriented Software*, Addison-Wesley, 1995.

[GL00] E. Gafni and L. Lamport, *Disk Paxos*, Proceedings of the Symposium on Distributed Computing, 2000, pp. 330–344.

[GMK02] I. Georgiadis, J. Magee, and J. Kramer, *Self-Organizing Software Architecture for Distributed Systems*, Proceedings of the Workshop on Self-Healing Systems, WOSS 2002, ACM, 2002.

[GNV02] S. Gutierrez-Nolasco and N. Venkatasubramanian, *A Reflective Middleware Framework for Communication in Dynamic Environments*, Proceedings of the International Symposium on Distributed Objects and Applications (DOA'02) (Irvine, CA), October 2002.

[Gro01] Object Management Group, *Dynamic Scheduling, final adopted specification, ptc/01-08-34*, formal/02-06-65, 2001.

[Gro02] ———, *CORBA Components*, formal/02-06-65, June 2002.

[Hal04] R. S. Hall, *A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks*, Proceedings of the International Conference on Component Deployment (CD'04) (Edinburgh, Scotland), 2004.

[Hil98] M. Hiltunen, *Configuration Management for Highly-Customizable Software*, IEE Proceedings - Software, vol. 145, October 1998, pp. 180–188.

[HMT+04] A. Hachichi, C. Martin, G. Thomas, S. Patarin, and B. Folliot, *Reconfigurations Dynamiques de Services dans un Intergiciel à Composants CORBA CCM*, Proceedings of the 1ère Conférence Française sur le Déploiement et la (Re)Configuration de Logiciels (DECOR'04) (Grenoble, France), October 2004.

[Hoa85] C. Hoare, *Communicating Sequential Processes*, 1985.

[HS00] M. Hiltunen and R. Schlichting, *The Cactus Approach to Building Configurable Middleware Services*, Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC'00) (Nuremberg, Germany), October 2000.

[HT93a] V. Hadzilacos and S. Toueg, *Fault-Tolerant Broadcasts and Related Problems*, 97–145.

[HT93b] ———, *Reliable Broadcasts and Related Problems*, Distributed Systems (S. Mullender, ed.), Addison-Wesley, 1993, pp. 97–145.

[HT03] P. Hnetynka and P. Tuma, *Fighting Class Name Clashes in Java Component Systems*, Proceedings of the Joint Modular Language Conference (JMLC'03) (Klagenfurt, Austria), 2003.

[IB96a] 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), May 1996, pp. 194–201.

[IB96b] ———, *Aster : A Framework for Sound Customization of Distributed Runtime Systems*, Proceedings of the International Conference on Distributed Computing Systems (Hong-Kong, Japon), May 1996, pp. 586–593.

[Iss97] V. Issarny, *Architectures Logicielles pour Systèmes Distribués*, Habilitation à diriger des recherches, IRISA+IFSIC, October 1997.

[ISZ98] V. Issarny, T. Saridakis, and A. Zarras, *A Survey of Architecture Description Languages*, C3DS Deliverable A3.1, ESPRIT LTR Project, 1998.

[j2e02] *J2EE : Java 2 Platform, Enterprise Edition*, 2002, http ://-java.sun.com/j2ee/docs.html.

[JGKvS04] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, *The Peer Sampling Service : Experimental Evaluation of Unstructured Gossip-Based Implementations*, Proceedings of the International Conference on Middleware (Toronto, Canada), 2004, pp. 79–98.

[jgr05] *JGroups - A Toolkit for Reliable Multicast Communication*, 2005, http ://www.jgroups.org.

[JOn04] JOnAS class loader hierarchy, 2004, `http ://jonas.objectweb.org/`.

[jor05a] *JORAM : Java Open Reliable Asynchronous Messaging*, 2005, Objectweb, `http ://joram.objectweb.org/`.

[jor05b] *JORM : Java Object Repository Mapping*, 2005, Objectweb, `http ://jorm.objectweb.org/`.

[jul02] *Julia : Fractal Composition Framework Reference Implementation*, 2002, Objectweb, http ://www.objectweb.org/fractal/.

[KC00] F. Kon and R. Campbell, *Dependence Management in Component-Based Distributed Systems*, vol. 8, pp. 26–36, February 2000.

[KC03] J. O. Kephart and D. M. Chess, *The Vision of Autonomic Computing*, IEEE Computer 36(1) (2003).

[KCBC02] F. Kon, F. Costa, G. Blair, and R. Campbell, *The Case for Reflective Middleware*, Communications of the ACM **45** (2002), no. 6, 33–38.

[KGCM00] F. Kon, B. Gill, R. Campbell, and M. Mickunas, *Secure Dynamic Reconfiguration of Scalable CORBA Systems with Mobile Agents*, Proceedings of the IEEE Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA'00) (Zurich, Swiss), 2000, pp. 86–98.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97) (Jyväskylä, Finland), June 1997.

[KM85] J. Kramer and J. Magee, *Dynamic Configuration for Distributed Systems*, IEEE Transactions on Software Engineering **11** (1985), no. 4, 424–436.

[KM90] ———, *The Evolving Philosophers Problem : Dynamic Change Management*, IEEE Transactions of Software Engineering **16** (1990), no. 11, 1293–1306.

[KR01] I. Keidar and S. Rajsbaum, *On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial*, Tech. report, MIT Technical Report MIT-LCS-TR-821, 2001, (Preliminary version in SIGACT News, Distributed Computing Column, 32(2) :45–63, 2001).

[KRL+00] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. Magalhães, and R. 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'00) (New York, USA), LNCS, no. 1795, Springer-Verlag, April 2000, pp. 121–143.

[KS05] S. Krakowiak and J.-B. Stefani, *`export-bind` : un Patron d'Architecture pour la Liaison Adaptable*, Informatique Répartie, Hermès, 2005, Hors-série de la Revue des sciences et technologies de l'information.

[KSD02] M. Knop, J. Schopf, and P. Dinda, *Windows Performance Monitoring and Data Reduction Using WatchTower*, Proceedings of the Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN) (New York City), June 2002.

[KSSS93] R. Karp, A. Sahay, E. Santos, and K. Schauser, *Optimal Broadcast and Summation in the LogP Model*, ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 142–153.

[KT96] F. Kaashoek and A. Tanenbaum, *An Evaluation of the Amoeba Group Communication System*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS'96), 1996, pp. 436–448.

[KYH+01] F. Kon, T. Yamane, K. Hess, R. Campbell, and M. Mickunas, *Dynamic Resource Management and Automatic Configuration of Distributed Component Systems*, Proceedings of the Conference on Object-Oriented Technologies and Systems (COOTS'01) (San Antonio, USA), January 2001.

[Lam78] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM **21** (1978), no. 7.

[Lam89] ———, *The Part-Time Parliament*, Tech. Report 49, DEC Systems Research Center, 1989, Also published in ACM Transactions on Computer Systems (TOCS), Vol. 16, No. 2, 1998.

[LH05] O. Layaïda and D. Hagimont, *Designing Self-Adaptive Multimedia Applications Through Hierarchical Reconfiguration*, Proceedings of the Conference on Distributed Applications and Interoperable Systems (DAIS'05) (Athens, Greece), June 2005, pp. 95–107.

[LKA+95] D. Luckham, J. Kenney, L. 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 **21** (1995), no. 4, 336–355.

[LQS05] M. Leclercq, V. Quéma, and J.-B. Stefani, *The Dream Framework Library*, 2005, http ://dream.objectweb.org/dreamlib/.

[LSZ+01] J. Loyall, R Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, *Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01) (Washington, USA), April 2001.

[LV95] D. Luckham and J. Vera, *An Event-Based Architecture Definition Language*, IEEE Transactions on Software Engineering **21** (1995), no. 9, 717–734.

[LvdH04] C. Luer and A. van der Hoek, *JPloy : User-Centric Deployment Support in a Component Platform*, Proceedings of the International Working Conference on Component Deployment (CD'04) (Edinburg, Scotland), 2004.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*, Proceedings of the European Software Engineering Conference (Sitges, Spain), September 1995, pp. 137–153.

[MDK94] J. Magee, N. Dulay, and J. Kramer, *Regis : a Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering **1** (1994), no. 5, 304–312.

[MFH01] S. McDirmid, M. Flatt, and W. Hsieh, *Jiazzi : New-Age Components for Old-Fashioned Java*, Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), ACM Press, 2001.

[MM00] R. Marvie and P. Merle, *Vers un modèle de composants pour CESURE, Le CORBA Component Model*, Tech. Report Projet RNRT, LIFL, November 2000.

[MPR01] H. Miranda, A. Pinto, and L. Rodrigues, *Appia : a Flexible Protocol Kernel Supporting Multiple Coordinated Channels*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01) (Mesa, USA), April 2001, pp. 707–710.

[MRT99] N. Medvidovic, D. Rosenblum, and R. Taylor, *A Language and Environment for Architecture-Based Software Development and Evolution*, Proceedings of the International Conference on Software Engineering (ICSE'99), 1999, pp. 44–53.

[MT00] N. Medvidovic and R. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering **26** (2000), no. 1.

[Net] Netperf, http ://www.netperf.org/.

[net02] *The NetLogger Toolkit : End-to-End Monitoring and Analysis of Distributed Systems*, June 2002, DIDC, Lawrence Berkley National Laboratory, http ://www-didc.lbl.gov/NetLogger/.

[OFT04] F. Ogel, B. Folliot, and G. Thomas, *A Step Towards Ubiquitous Computing : an Efficient Flexible Micro-ORB*, Proceedings of the ACM SIGOPS European Workshop (Leuven, Belgium), September 2004.

[Oga97] K. Ogata, *Modern Control Engineering, 3rd edition*, Prentice-Hall, 1997.

[OGT+99] P. Oriezy, M. Gorlick, R. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems 14(3) (1999).

[Ope03] Open Services Gateway Initiative, OSGi service gateway specification, Release 3, April 2003, `http ://www.osgi.org`.

[pab99] *Scalable Performance Tools (Pablo Toolkit)*, June 1999, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, http ://www-pablo.cs.uiuc.edu/Project/Pablo/ScalPerfToolsOverview.htm.

[PBJ97] F. Plasil, D. Balek, and R. Janecek, *Dynamic Component Updating in Java/-CORBA Environment*, Tech. Report 97/10, Department of Software Engineering, Charles University, Prague, 1997.

[PBJ98] ———, *SOFA/DCUP : Architecture for Component Trading and Dynamic Updating*, Proceedings of International Conference on Configurable Distributed Systems (ICCDS'98) (Annapolis, Maryland, USA), 1998.

[POF01] I. Piumarta, F. Ogel, and B. Folliot, *YNVM : Dynamic Compilation in Support of Software Evolution*, Proceedings of the Workshop on Engineering Complex Object Oriented System for Evolution, in association with OOPSLA (Tampa Bay, USA), October 2001.

[Pur94] J. Purtilo, *The POLYLITH Software Bus*, ACM Transaction on Programming Languages and Systems **16** (1994), no. 1, 151–174.

[QB02] V. Quéma and L. Bellissard, *Configuration de Middleware Dirigée par les Applications*, Actes des Journées sur les Systèmes à Composants Adaptables et Extensibles (Grenoble, France), 17-18 octobre 2002.

[QBB+04] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacourte, *Scalagent : une plate-forme à composants pour applications asynchrones*, Technique et Science Informatiques **23** (2004), no. 2.

[QBL02] V. Quéma, L. Bellissard, and P. Laumay, *Application-Driven Customization of Message-Oriented Middleware for Consumer Devices*, Workshop on Software

Infrastructures for Component-Based Applications on Consumer Devices, in association with EDOC'02 (Lausanne, Switzerland), September 2002.

[QC03] V. Quéma and E. Cecchet, *The Role of Software Architecture in Configuring Middleware : the ScalAgent Experience*, Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'03) (La Martinique, France), 2003.

[RAJ02] E. Román, S. Ambler, and T. Jewell, *Mastering Enterprise JavaBeans, second edition*, Wiley Computer Publishing, 2002.

[Rém93a] D. Rémy, *Type Inference for Records in a Natural Extension of ML*, Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design (Carl A. Gunter and John C. Mitchell, eds.), MIT Press, 1993.

[Rém93b] ———, *Typing Record Concatenation for Free*, Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design (Carl A. Gunter and John C. Mitchell, eds.), MIT Press, 1993.

[rem02] *Remos : Resource Monitoring for Network-Aware Applications*, April 2002, http ://www-2.cs.cmu.edu/~cmcl/remulac/remos.html.

[RKC99] M. Román, F. Kon, and R. Campbell, *Design and Implementation of Runtime Reflection in Communication Middleware : the dynamicTAO Case*, May 1999.

[RMKC00] M. Román, D. Mickunas, F. Kon, and R. Campbell, *LegORB and Ubiquitous CORBA*, Proceedings of the Middleware'00 Workshop on Reflective Middleware (New-York, USA), April 2000.

[Rog97] D. Rogerson, *Inside COM*, Microsoft Press, Redmond, USA, 1997.

[Sal02] C. Salzmann, *Invariants of Component Reconfiguration*, Proceedings of the International Workshop on Component-Oriented Programming (WCOP'02), 2002.

[SB98] L. Sheng and G. Bracha, *Dynamic class loading in the java virtual machine*, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98) (Vancouver, Canada), 1998.

[SBC+98] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward, *Gryphon : an Information Flow Based Approach to Message Brokering*, Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'98), fast abstract (Paderborn, Germany), November 1998.

[SBF02] C. Boutros Saab, X. Bonnaire, and B. Folliot, *PHOENIX : a Self Adaptable Monitoring Platform for Cluster Management*, Cluster Computing **5** (2002), no. 1, 75–85.

[SBS91] A. Schiper, K. Birman, and P. Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems **9** (1991), no. 3, 272–314.

[SBS93] D. Schmidt, D. Box, and T. Suda, *The Adaptive Communication Environment : An Object-Oriented Network Programming Toolkit for Developing Communication Software*, Concurrency : Practice and Experience **5** (1993), no. 4, 269–286.

[SC99] D. Schmidt and C. Cleeland, *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Communications Magazine Special Issue on Design Patterns (1999).

[Sch93] F. Schneider, *Replication Management using the State Machine Approach*, Distributed Systems (S. Mullender, ed.), Addison-Wesley, 1993.

[SDK+95] 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.

[Sel00] B. Selic, *A Generic Framework for Modeling Resources with UML*, IEEE Computer **33** (2000), no. 6, 64–69.

[SH02] C. Smith and D. Henry, *High-Performance Linux Cluster Monitoring Using Java*, Proceedings of the Linux Cluster International Conference, 2002.

[SLR+03] R. Schantz, J. Loyall, C. Rodrigues, D. Schmidt, Y. Krishnamurthy, and I. Pyarali, *Flexible and Adaptive QoS Control for Distributed Real-Time and Embedded Middleware*, Proceedings of the Middleware Conference (Middleware 2003) (Rio de Janeiro, Brazil), June 2003, pp. 374–393.

[son02] *sonicMQ*, 2002, Sonic software, http ://www.sonicsoftware.com/.

[SW98] S. Shrivastava and S. Wheater, *Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications*, Proceedings of the International Conference on Configurable Distributed Systems (ICCDS'98), 1998.

[Szy02] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2002.

[TCG+00] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson, *A Monitoring Sensor Management System for Grid Environments*, Proceedings of the Symposium on High Performance Distributed Computing (HPDC'00) (Pittsburgh, USA), August 2000, p. 97.

[VDM+01] N. Venkatasubramanian, M. Deshpande, S. Mohapatra, S. Gutierrez-Nolasco, and J. Wickramasuriya, *Design and Implementation of a Composable Reflective Middleware Framework*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS'01) (Washington, USA), April 2001, pp. 644–653.

[Ven98] N. Venkatasubramanian, *An Adaptive Resource Management Architecture for Global Distributed Computing*, Ph.D. thesis, University of Illinois at Urbana-Champaign, USA, 1998.

[Ven02] ———, *Safe 'Composability' of Middleware Services*, Communications of the ACM **45** (2002), no. 6, 49–52.

[vRBM96] R. van Renesse, K. Birman, and S. Maffeis, *Horus : a Flexible Group Communication System*, Commun. ACM **39** (1996), no. 4, 76–83.

[VT95] N. Venkatasubramanian and C. Talcott, *Reasoning about Meta Level Activities in Open Distributed Systems*, Proceedings of the Symposium on Principles of Distributed Computing (PODC'95) (Ottowa, Canada), 1995, pp. 144–152.

[WCB01] M. Welsh, D. Culler, and E. Brewer, *SEDA : An Architecture for Well-Conditioned, Scalable Internet Services*, Proceedings of the Symposium on Operating Systems Principles (SOSP'01) (Banff, Canada), October 2001.

[Web03] WebSphere Software Information Center, Class loaders, 2003, `http ://www.ibm.com/`.

[Web04] WebLogic Server Application Classloading, 2004, `http ://www.bea.com/`.

[Wel02] M. Welsh, *An Architecture for Highly Concurrent, Well-Conditioned Internet Services*, Ph.D. thesis, University of California at Berkeley, 2002.

[WHW+04] S. White, J. Hanon, I. Whalley, D. Chess, and J. Kephart, *An Architectural Approach to Autonomic Computing*, Proceedings IEEE Int. Conference on Autonomic Computing (ICAC 2004), 2004.

[WSH99] R. Wolski, N. Spring, and J. Hayes, *The Network Weather Service : a Distributed Resource Performance Forecasting Service for Metacomputing*, Future Generation Computer Systems **15** (1999), no. 5-6, 757–768.

[YMV+03] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, *Separating Agreement from Execution for Byzantine Fault Tolerant Services*, Proceedings of the Symposium on Operating Systems Principles (SOSP'03) (Bolton Landing, USA), October 2003, pp. 253–267.

[ZBS97] J. Zinky, D. Bakken, and R. Schantz, *Architectural Support for Quality of Service for CORBA Objects*, Theory and Practice of Object Systems **3** (1997), no. 1.