



THÈSE

présentée

DEVANT L'UNIVERSITÉ DE RENNES I

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES I

Mention : *Informatique*

par

Selma MATOUGUI

Équipe d'accueil : *Laboratoire d'Informatique des Télécommunications,
ENST-Bretagne*

École doctorale : *Matisse*

Proposition d'un processus de réification d'abstraction de communication comme un connecteur associé à des générateurs

soutenue le 12 décembre 2005 devant la commission d'examen :

Composition du Jury :

Président : Mme Françoise André, Professeur à l'Université de Rennes 1

Rapporteurs : Mme Isabelle Borne, Professeur à l'Université de Bretagne-Sud
M. Mourad Oussalah, Professeur à l'Université de Nantes

Examineurs : M. Bruno Traverson, Ingénieur de Recherche à EDF R&D
M. Antoine Beugnard, Enseignant-Chercheur à l'ENST-Bretagne
M. Jean-Marc Jézéquel, Professeur à l'Université de Rennes 1

Résumé

Les approches de conception et de programmation à base de composants et les architectures logicielles séparent les composants de leurs interconnexions. Ces dernières, souvent appelées connecteurs, possèdent des définitions diverses et contradictoires dans la communauté. Le but de cette thèse est de clarifier le concept de connecteur et de proposer un processus pour son implémentation sous la forme d'un ensemble de générateurs. Ainsi, nous définissons un connecteur comme un élément d'architecture qui évolue et se concrétise durant son cycle de vie. Nous discutons en détails les différents concepts relatifs à ce cycle de vie, et nous les illustrons au travers de la mise en œuvre et l'évaluation d'un connecteur d'équilibrage de charge. Pour finir, nous dressons une classification entre les deux types d'abstraction de communication : les connecteurs et les composants de communication. Ainsi, nous mettons en avant les différences et les principaux critères de choix entre ces deux entités.

Mots clés : architecture et composants logiciels, connecteurs, interactions, générateurs, cycle de vie, assemblage de composants.

Abstract

Component based programming approaches and software architectures distinguish the components from their interconnections. The interactions between components, also called connectors, have different and contradictory definitions in the literature. The aim of this PhD work is to provide a clearer definition of what a connector is and to propose a process for its implementation as a set of generators. First, we define a connector as an architectural element that evolves and becomes concrete during its life cycle. Then, we study in more details the different concepts of this life cycle and illustrate them with the realization and the evaluation of a load balancing connector. Finally, we present a classification between the two kinds of communication abstraction : the connector and the communication component. Hence, we discuss the differences and the main choice criteria between these two entities.

Key words : software components and architecture, connectors, interactions, generators, life cycle, component assembly.

Remerciements

Ce travail de thèse ne serait bien entendu pas le même sans la rencontre, l'apport et le soutien de nombreuses personnes.

Je tiens tout d'abord à remercier Antoine Beugnard, Enseignant-Chercheur à l'ENST-Bretagne, pour avoir suivi et encadré ma thèse. Par son enthousiasme et sa gentillesse, il a su me guider pour choisir les voies les plus valorisantes et m'encourager pendant les moments difficiles. Merci pour tous les conseils et les nombreuses discussions qui ont permis de faire mûrir ce travail de recherche.

J'adresse toute ma gratitude à Mme Isabelle Borne, Professeur à l'Université de Bretagne-Sud, et à M. Mourad Oussalah, Professeur à l'Université de Nantes, pour avoir immédiatement accepté la charge de rapporteurs. Merci pour le temps passé à lire et à évaluer mon travail au travers de ce document de thèse ainsi que pour toutes les remarques et commentaires constructifs qui ont pu en découler.

Je voudrais aussi exprimer mes plus vifs remerciements à Mme Françoise André, Professeur à l'Université de Rennes 1, pour m'avoir fait l'honneur de présider mon jury de thèse ; à M. Bruno Traverson, Ingénieur de recherche à EDF R&D, pour avoir sans aucune hésitation accepté de faire partie de mon jury de thèse et de l'intérêt qu'il a porté à mon travail ; et à M. Jean-Marc Jézéquel, Professeur à l'Université de Rennes 1, pour avoir supervisé mes travaux et veillé au bon déroulement de ma thèse.

Je tiens à remercier tous les membres du département Informatique de l'ENST-Bretagne pour m'avoir accueillie et pour avoir contribué à rendre mon séjour agréable en Bretagne. J'aimerais tout particulièrement citer Anne-Marie L'Hostis et Odile Ely qui ont bien voulu relire et corriger ma thèse ; Fabien Dagnat, Siegfried Rouvrais, Julien Mallet et Christophe Lohr pour leurs suggestions et conseils. Mes remerciements vont également aux participants du projet ACCORD avec lesquels j'ai eu le grand plaisir de partager de nombreuses et très enrichissantes discussions scientifiques, et ce tout au long des deux années du projet.

Enfin, je tiens à remercier tous les membres de ma famille qui me sont chers ainsi que tous mes amis qui m'ont tout le temps soutenue. J'ai toujours pu compter sur eux que ce soit dans les moments de joie mais aussi dans tous les moments difficiles. Une mention particulière est dédiée à Yacine qui m'a épaulé et cru en moi jusqu'au bout. Je remercie également tous ceux que je n'ai pas cités et qui ont contribué de pré ou de loin au bon déroulement de ma thèse.

*À la mémoire de mon père,
à ma mère.*

Table des matières

Résumé	i
Abstract	iii
Remerciements	v
Introduction	1
1 Etat de l'art	5
1.1 Petit historique	5
1.2 Les interactions comme entité non réifiée	9
1.2.1 Approches académiques	9
1.2.2 Approches industrielles	13
1.3 Les interactions comme entité réifiée	15
1.3.1 Approches avec un nombre limité de connecteurs	15
1.3.2 Approches avec des connecteurs définis par l'utilisateur	17
1.4 Synthèse	20
2 Les connecteurs	21
2.1 Problématique et motivation	21
2.1.1 Besoin de nouvelles notations et de nouveaux concepts	23
2.1.2 Besoin de nouvelles abstractions de communication et de leurs implémentations	31
2.1.3 Synthèse	36
2.2 Définition et cycle de vie des connecteurs	37
2.2.1 Travaux connexes	37
2.2.2 Notre Approche	39
2.2.3 Définition du connecteur	40
2.2.4 Cycle de vie du connecteur	43
2.2.5 Synthèse	47

2.3	Processus de conception et d'utilisation des connecteurs	48
2.3.1	Identification et description	48
2.3.2	Sélection et assemblage	52
2.3.3	Conception et implémentation	55
2.3.4	Génération et déploiement	59
2.4	Synthèse	61
3	Mise en œuvre	63
3.1	Description de l'équilibrage de charge	63
3.1.1	Description générale	63
3.1.2	Caractéristiques de la propriété d'équilibrage de Charge	64
3.2	Connecteur d'équilibrage de charge	67
3.2.1	Description du connecteur d'équilibrage de charge	67
3.2.2	Exigences d'un service d'équilibrage de charge	68
3.3	Présentation de Jonathan	70
3.3.1	Principaux concepts de Jonathan	70
3.3.2	Architecture de Jonathan	72
3.3.3	Etude de Jeremie : la personnalité RMI de Jonathan	73
3.4	Mise en œuvre du connecteur d'équilibrage de charge	75
3.4.1	Réalisation des générateurs	76
3.4.2	Nouvelle architecture de l'application	81
3.5	Tests et résultats	82
3.5.1	Description du matériel et de la plate-forme de test	82
3.5.2	Résultats et performances	83
3.6	Synthèse	91
4	Classification	93
4.1	Deux types d'abstraction de communication	93
4.1.1	Les composants de communication : Définition	94
4.1.2	Les connecteurs vs les composants de communication	96
4.1.3	Exemples	98
4.2	Différences et éléments de choix et d'utilisation	102
4.2.1	Éléments de différenciation	102
4.2.2	Éléments de choix d'utilisation et de fabrication	104
4.3	Classification de quelques éléments de connexion	106
4.4	Synthèse	109
	Conclusion	113

Bibliographie

117

Liste des figures

1.1	Description d'une architecture Darwin	9
1.2	Description d'une architecture Fractal	11
1.3	Description d'une architecture CCM	14
1.4	Description d'une architecture Wright	18
2.1	Différents types de connexions dans le modèle <i>boxes-and-lines</i>	22
2.2	Détails d'une connexion RPC au déploiement	24
2.3	Connexion complexe. (a) plusieurs clients - un serveur, (b) un client - plusieurs serveurs	26
2.4	Détails d'une connexion RPC au déploiement avec découpage	27
2.5	Nouvelle forme d'un connecteur : une ellipse	30
2.6	Connexion complexe avec un connecteur RMI	32
2.7	Les propriétés synchrone et asynchrone et quelques réalisations	33
2.8	Connexion complexe avec un connecteur d'équilibrage de charge	36
2.9	Le connecteur	42
2.10	Le générateur	44
2.11	La connexion	45
2.12	Le composant de liaison	46
2.13	Diagramme de cas d'utilisation du connecteur	48
2.14	Phases identification et description	49
2.15	Description d'un connecteur	49
2.16	Phases sélection et assemblage	52
2.17	Description d'une connexion	53
2.18	Phases conception et implémentation	55
2.19	Description du générateur	56
2.20	Phases génération et déploiement	59
2.21	Cycle de vie de la prise	60
2.22	Description du composant de liaison	60

2.23	Les phases du connecteur	62
3.1	Liaison client/serveur avec un objet de liaison	71
3.2	Schéma d'une invocation entre une application cliente et un objet distribué dans Jonathan	73
3.3	Connexion complexe avec un connecteur RMI	74
3.4	Niveau d'intervention sur Jeremie pour la réalisation du générateur d'équilibrage de charge non adaptatif	77
3.5	Diagramme de séquence de la politique d'équilibrage de charge Round Robin	77
3.6	Niveau d'intervention sur Jeremie pour la réalisation du générateur d'équilibrage de charge adaptatif	79
3.7	Diagramme de séquence pour la politique d'équilibrage de charge du serveur le moins chargé	80
3.8	Une application qui utilise un connecteur d'équilibrage de charge	82
3.9	Productivité avec un serveur	84
3.10	Temps de réponse avec un serveur	84
3.11	Résultats de la productivité avec la politique aléatoire	85
3.12	Résultats de la latence avec la politique aléatoire	86
3.13	Résultats de la productivité avec la politique Round Robin	87
3.14	Résultats de la latence avec la politique Round Robin	87
3.15	Résultats de la productivité avec la politique du moins chargé	88
3.16	Résultats de la latence avec la politique du moins chargé	88
3.17	Résultats de productivité avec 2 serveurs	89
3.18	Résultats de productivité avec 8 serveurs	90
3.19	Résultats de latence avec 8 serveurs	91
4.1	Relation entre un rôle et un médium	95
4.2	Deux types d'abstraction de communication. (a) médium, (b) connecteur	96
4.3	Diagramme de collaboration du médium d'équilibrage de charge	98
4.4	Equilibrage de charge avec un composant de communication	99
4.5	Diagramme de séquence pour la politique d'équilibrage de charge du serveur le moins chargé	101
4.6	Une application qui utilise un médium de réservation	105

Introduction

Le domaine de l'ingénierie du logiciel a connu ces dernières années de nombreuses avancées notamment par l'apparition de plusieurs approches de développement d'applications. Parmi celles-ci, nous nous intéressons dans cette thèse à l'approche de développement de logiciels à base de composants. Cette approche est inspirée du principe des composants électroniques où les systèmes sont obtenus en interconnectant plusieurs composants. Un composant est une boîte noire qui cache sa complexité de réalisation et qui est accessible uniquement à travers ses interfaces. Un constat relatif à cette analogie entre ces deux domaines très différents est que leurs évolutions ne sont pas équivalentes. D'un côté, l'évolution du matériel suit la loi de Moore¹ : « il est possible de placer 4 fois plus de transistor sur une puce tous les 3 ans. On devrait ainsi arriver à 1 milliard de transistors sur une puce aux alentours de 2010 » (Intel©). D'un autre côté, dans le domaine du logiciel il n'est apparu ces dix dernières années que quelques modèles à composants qui sont réellement utilisés dans l'industrie comme CCM, EJB et .Net. À quoi serait due cette différence d'évolution ?

Si les ingénieurs en matériel électronique devaient partir uniquement de moyens élémentaires à chaque fois qu'ils conçoivent un nouveau dispositif, et si leur première étape était d'abord d'extraire de la matière première pour fabriquer leurs composants (par exemple du silicium pour construire leurs circuits intégrés), ils ne progresseraient pas aussi vite². Un concepteur de matériel construit toujours un système à partir de composants préparés, sa tâche est considérablement simplifiée par le travail de ses prédécesseurs. Ce concept de réutilisation des moyens existants a été adopté par les concepteurs de logiciel. Les applications sont créées à partir de composants existants et de nouveaux concepts apparaissent à partir de concepts existants. Les concepts orientés objet reposent sur les définitions des procédures, et les concepts des approches à base de composant reposent sur ceux des objets. Ainsi, les concepteurs de logiciel tirent profit du travail de leurs prédécesseurs quant à la réalisation des composants logiciels. Cependant, ce concept n'est pas totalement exploité pour les *interconnexions* entre les composants. En effet, selon qu'on se positionne dans un contexte à objet ou à composant, les plates-formes associées, comme les intergiels, n'offrent qu'un nombre limité de formes d'interaction entre composants ou objets distribués. Ces interactions sont souvent basées sur les appels de procédure à distance ou la diffusion d'événements. Ainsi, pour réaliser les interactions entre les composants, les développeurs d'applications utilisent toujours des abstractions de communication simples et élémentaires qui peuvent être comparées à de la matière première. Le résultat des développements utilisant ces moyens de communication est souvent excellent mais peut être amélioré. La réutilisation est une voie pour créer de meilleurs logiciels et la conception d'interactions complexes pourrait être

¹Gordon Moore : Cofondateur de la société Intel.

²Extrait du livre de David Chappell [19].

capitalisée pour être réutilisée.

De plus, si les concepteurs de matériel réussissent à fabriquer des composants fermés (boîte noire) qui ne sont accédés et connectés qu'à partir de leurs interfaces, les concepteurs de logiciels ont la possibilité de modifier les composants logiciels. Ces derniers sont encore considérés comme des boîtes grises où il est toujours possible de rajouter du code. Il est également possible à ces composants logiciels de ne pas communiquer qu'à travers leurs interfaces, ce qui limite leur réutilisation et rend leur assemblage compliqué. Ces deux problèmes de réutilisation des interactions complexes et d'assemblage de composants que nous avons évoqués participent sans doute à la différence d'évolution des domaines matériels et logiciels !

Ce thème de l'assemblage de composants a été l'objet du projet ACCORD [45], dans lequel a commencé notre travail. L'objectif principal du projet était de proposer aux architectes de systèmes d'information un cadre d'analyse et de conception adapté à l'approche de conception de systèmes logiciels par composants. Le résultat est un modèle de composants [36] abstrait et indépendant des plates-formes, qui inclut les concepts de contrat et de *connecteur* nécessaires pour l'assemblage des composants. Des outils de mise en œuvre qui projettent le modèle de composants d'ACCORD sur différentes plates-formes (EJB et CCM) ont également été fournis. Cependant, la gestion des connexions dans le modèle a été traitée de manière ad hoc car elles dépendent des plates-formes utilisées. L'objet de cette thèse est de fournir un modèle de connecteur indépendant des plates-formes. Nous proposons également un processus d'implémentation des connecteurs.

Contributions

Le concept de connecteur a fait débat lors du projet ACCORD. Il a fallu du temps et plusieurs discussions pour pouvoir se comprendre et pour converger vers le même sens en évoquant le mot « connecteur ». En revanche, s'il y a eu un accord sur ce terme dans le projet, ceci n'est pas vraiment le cas dans les approches existantes. En effet, bien qu'il existe plusieurs travaux sur les interconnexions entre les composants, le concept de connecteur reste mal défini. Il existe des considérations très différentes et contradictoires concernant les connecteurs. Par exemple, certaines approches, comme les modèles de composants, ne définissent les interconnexions entre les composants qu'avec des liaisons simples point-à-point. Ceci laisse une grande partie des communications à la charge des composants et limite ainsi leur réutilisation. D'autres approches, notamment en architecture logicielle, les décrivent souvent comme des entités à part entière mais les réalisent à base de connexions simples ou sous forme de composants. Il est ainsi difficile de préserver un connecteur comme une entité unique dans les différentes phases du processus de développement.

Dans cette optique, le but de cette thèse est de définir un modèle de connecteur ainsi que son processus d'implémentation. Nous défendons l'existence des connecteurs et nous adoptons les approches qui les considèrent comme des entités à part entière, afin de permettre une meilleure séparation des responsabilités et d'offrir plus de réutilisation. Cependant, nous révisons cette notion de connecteur en donnant une définition plus précise. Pour cela, nous définissons un connecteur comme une entité d'architecture abstraite qui se transforme dans le processus de développement pour se concrétiser. Il existe par sa propriété et non pas par ses services et il définit des interfaces implicites appelées prises. Un connecteur est indépendant de toute plate-forme. Nous lui associons un cycle de vie particulier et nous définissons un vocabulaire précis pour désigner les transformations du connecteur dans son cycle de vie. L'étude de ce cycle de vie a été publiée dans [52].

D'un point de vue implémentation, et afin de préserver les propriétés de communication du connecteur comme une entité à part entière dans le processus de développement, nous proposons un processus d'implémentation des connecteurs comme une famille de générateurs. Le mécanisme de génération permet d'intégrer la propriété de communication du connecteur au-dessus d'une plate-forme de mise en œuvre et offre ainsi l'indépendance du connecteur à la plate-forme au niveau architecture. Sa mise en œuvre sur différentes plate-formes constitue la famille de générateurs. Ce mécanisme de génération permet également d'intégrer la propriété de communication aux composants avec transparence ce qui apporte une grande flexibilité pour la reconfiguration des composants et la substitution des connecteurs. Ainsi, notre approche permet de capitaliser le savoir-faire des interactions complexes qui sont implémentées pour être réutilisées.

Pour assurer la transparence à la communication les connecteurs possèdent des interfaces implicites ce qui les rend différents des composants. En effet, en plus de la différence de fonctionnalité classiquement exprimée, nous définissons une différence de nature reliée aux interfaces des deux entités. Ceci donne au connecteur un vrai statut d'entité de première classe dans tout le processus de développement logiciel. Ainsi, notre travail possède en plus une vocation de classification des moyens de communication. Cette classification a été présentée dans [53] et la différence des entités a été illustrée dans un cas d'étude de EDF (R&D) [6].

Plan de la thèse

Cette thèse est décomposée en quatre chapitres :

Le premier chapitre présente un état de l'art sur les approches de composant et d'architecture logicielle où nous étudions les définitions des interactions entre les composants logiciels adoptées par ces approches. Ce chapitre dresse ainsi un constat sur l'état actuel de la prise en compte des interconnexions entre les composants. Nous y étudierons les approches qui donnent aux connecteurs un second rôle et celles qui les considèrent comme des entités de première classe.

Les deux chapitres suivants constituent le cœur de notre proposition. Le chapitre 2 détaille la problématique et les motivations qui sont à l'origine de nos réflexions pour la proposition d'un modèle de connecteur et son implémentation. Ensuite, ce chapitre pose notre définition du connecteur : nous le définissons comme une entité d'architecture abstraite qui se concrétise au cours de son cycle de vie. Il décrit ainsi les différentes entités qui représentent le connecteur dans son cycle de vie. Un connecteur est implémenté comme un générateur, il forme une connexion lors de son assemblage avec les composants, et il est considéré comme un composant de liaison lors de son déploiement. Enfin, ce chapitre décrit le processus d'implémentation et les relations entre les différentes transformations du connecteur. Pour valider l'approche, le chapitre 3 présente un exemple de mise en œuvre d'un connecteur d'équilibrage de charge. Il décrit la propriété d'équilibrage de charge qui sera réifiée comme un connecteur. Il décrit la plate-forme de mise en œuvre et enfin la réalisation de ce connecteur comme des générateurs.

Le dernier chapitre a pour but de faire ressortir les principales différences entre un composant de communication et un connecteur et de comparer leurs caractéristiques. Il donne des éléments de choix de réification des abstractions de communication sous forme d'une des deux entités ainsi que des éléments de choix pour l'utilisation de ces entités. Il dresse une classification de quelques moyens de communication usuels selon qu'ils soient des connecteurs ou des composants de communication.

Pour finir, nous concluons en résumant les principaux apports et contributions de cette thèse, et nous présentons également les perspectives de notre travail.

Le développement logiciel a beaucoup évolué depuis ses débuts et a connu un grand progrès depuis l'apparition des premières machines il y a plus de cinquante ans. Un des concepts clés mis en avant pour un bon développement du logiciel est la séparation des responsabilités. Ce concept a particulièrement été exploité par le domaine de l'architecture logicielle pour devenir une branche à part entière et un thème de recherche académique important au cours de ces dix dernières années. Il s'agit dans ce cas de séparer les éléments dédiés aux fonctionnalités d'une application des éléments dédiés à la communication¹ au sein de l'application. Toutefois, avant que l'architecture logicielle ne devienne une discipline dans le processus de développement et ne trouve une reconnaissance dans la communauté du génie logiciel, ce dernier a connu de nombreuses évolutions. Les notions de communication et de fonctionnalité se sont distinguées et à partir de là des abstractions, des démarches et des outils appropriés sont apparus.

Nous allons voir dans ce chapitre les considérations actuelles des interconnexions entre entités logicielles dans les différentes approches liées au domaine de l'architecture logicielle. Nous nous intéressons pour chacune des approches à la façon d'exprimer et de considérer les connexions. Nous allons traiter ces approches suivant qu'elles définissent ou pas une entité explicite pour contenir les connexions. Mais avant d'entamer cette étude, et afin de définir ce nouveau domaine en pleine émergence, nous allons commencer par un bref rappel historique sur l'évolution du logiciel pour éclaircir comment l'architecture logicielle est apparue, comprendre les bases de ce domaine et l'importance et la place que détiennent les connexions dans ce domaine.

1.1 Petit historique

Origine du génie logiciel

Aux débuts de l'ère informatique, les programmeurs arrivaient à produire des programmes qui étaient très courts mais très bien structurés mathématiquement. Progressivement, les programmes se sont complexifiés et les besoins fonctionnels sont devenus de plus en plus importants. L'augmentation exponentielle de la taille des programmes et de leur coût à la fin des années 60, ainsi que l'avènement des gros systèmes a provoqué une crise du logiciel. En effet, les mécanismes utilisés pour les petits systèmes atteignaient leurs limites sur ces grosses applications : les gros projets étaient en retard de plusieurs années, le coût de production était bien supérieur à l'estimation d'origine, et les systèmes étaient peu fiables, difficiles à

¹Dans ce chapitre, nous utilisons les termes : interconnexion, connexion, interaction pour désigner les différentes formes de communication entre les composants, d'une manière très générale et non spécifique à un domaine particulier de l'informatique. Ce vocabulaire sera affiné dans les autres chapitres de cette thèse.

maintenir et avaient de faibles performances. De plus, il y avait très peu de réutilisation, et les besoins correspondaient directement aux implémentations sans mettre assez d'efforts sur la structure du logiciel. Ces problèmes ont forcé la communauté des programmeurs à formaliser leurs méthodes de production. C'est la naissance du Génie Logiciel [61].

Les premiers fondements pour la conception des logiciels

Durant les années 70 et 80, différents travaux se sont concentrés sur l'étude du développement, de la maintenance et de l'évolution des logiciels. Ces recherches ont montré l'intérêt de s'abstraire du code source pour modéliser, et ainsi de dissocier les aspects de conception des aspects de programmation. Ces différents résultats se sont succédés et ont contribué à mieux concevoir les logiciels en les structurant mieux. Nous rappelons dans ce qui suit quelques unes des étapes les plus importantes.

L'étude de la structure d'un logiciel a commencé en 1968 quand E. Dijkstra montra qu'il était avantageux de s'intéresser à la structure d'un programme plutôt qu'à sa programmation [28]. Il créa et utilisa en premier la structure en couches pour écrire un système d'exploitation. Il souligna les bénéfices de cette structure pour le développement et la maintenance. À cette époque, les programmes se servaient de branchements (*goto*), et des appels de procédure pour réaliser les connexions.

Un apport a été réalisé grâce à F. DeRemer et H. Kron en opposant « *Programming-in-the-large versus Programming-in-the-small* » [27]. Les langages de l'époque ne permettaient que de rester au niveau du code (*Programming-in-the-small*) et ne permettaient pas un raisonnement à un niveau élevé (*Programming-in-the-large*). Ceci allait à l'encontre des recommandations faites par les recherches sur la programmation structurée, c'est-à-dire trouver des solutions lisibles, compréhensibles et modifiables. Les auteurs argumentaient que la création des modules de programmes et les connexions de ceux-ci sont *deux efforts de conception différents*. Afin d'aider l'effort de connexion, ils inventèrent le premier « *Module Interconnection Language* » (MIL) nommé MIL75. Ensuite d'autres MILs et des systèmes utilisant ces MILs sont apparus [75].

Pendant les années 75/85 il y a eu un gros travail sur les modules qui a débouché sur les types abstraits de données (*Abstract Data Type - ADT*) incorporés dans les langages de programmation comme PASCAL ou ADA. La conception d'un type de données passe par une description formelle de chacune de ses opérations. Plusieurs langages dont CLU [38] ont été construits pour faciliter la gestion de ces types abstraits.

De nouveaux concepts sont apparus avec l'arrivée du paradigme de la programmation orientée objet [60] comme les notions de classe, d'héritage, d'encapsulation et de polymorphisme. Cette approche considère la structure d'un logiciel comme une collection d'objets dissociés intégrant des attributs (état de l'objet) et des opérations (comportement de l'objet). Une classe regroupe les objets de même nature. Les concepts d'encapsulation et de polymorphisme ont amélioré sensiblement la réutilisation et l'extension du code [72] par rapport aux technologies procédurales et fonctionnelles. Cependant, ces approches n'offrent pas encore assez d'abstractions notamment en matière de relations entre les objets.

Au début des années 80, de nombreuses recherches ont été réalisées par l'université de Carnegie Mellon pour réussir à faire communiquer des systèmes écrits séparément dans les langages éventuellement différents. Pour pouvoir réaliser cela, les chercheurs ont créé un langage sophistiqué appelé IDL (*Interface Description Language*) [62]. Les modèles de composant récents (COM/DCOM, CORBA, .NET) incorporent leur propre IDL. Cette période marque

aussi l'avènement des systèmes distribués et des applications réparties.

Parallèlement à cela, de nombreuses méthodes de conception et techniques de spécification ont émergé [77, 10, 35]. Pour ce qui est des méthodes de conception, les plus connues sont SADT (*System Analysis and Design Technique*) [78] pour les méthodes fonctionnelles et OOA (*Object-Oriented Analysis*) [23] et OMT (*Object Modeling Technique*) [79] pour les méthodes objet. Dans les années 90, un effort de standardisation a permis de regrouper les différentes notations de ces différentes méthodes en une seule nommée UML (*Unified Modeling Language*) [66]. Au niveau de la spécification, il y a eu une évolution de l'énoncé informel (langage naturel) aux techniques semi formelles ou graphiques (modèle entité-relation, etc), pour arriver aux spécifications formelles (types abstraits algébriques, Z, B, etc) [34, 88]. L'avantage des spécifications formelles est qu'elles reposent sur des sémantiques bien définies permettant ainsi de vérifier des propriétés et de lever toute ambiguïté. Cependant, leur écriture n'est pas facile et elles sont difficiles à comprendre pour les non initiés.

L'ensemble de ces travaux a abouti à une amélioration et une meilleure maîtrise de la conception d'un logiciel. Toutes ces avancées ont permis de réaliser de très gros systèmes. Cependant, il n'est pas difficile de faire le constat que ces avancées n'ont pas permis de résoudre toutes les difficultés. Les retards dans la production des logiciels ont été mieux maîtrisés mais beaucoup de projets ne finissaient toujours pas dans les temps prévus. En outre, le coût de production était largement supérieur à celui prévu initialement !

Premiers pas vers l'architecture logicielle

Parallèlement aux avancées logicielles, le matériel et les technologies ont considérablement été améliorés, ce qui a rendu possible la réalisation d'applications plus grandes et plus complexes afin de répondre aux besoins sans cesse grandissants des utilisateurs. Ces deux raisons ont incité la création de nouvelles technologies et méthodologies pour concevoir les applications. De nouvelles approches en génie logiciel sont apparues dans les années 90 comme les modèles de composants, la création d'un standard de modélisation à travers UML, et l'architecture logicielle [74].

Avec l'avènement du domaine de l'architecture logicielle, les concepteurs de logiciels ont peu à peu pris conscience du rôle déterminant que joue ce domaine dans la réussite du développement, de la maintenance et de l'évolution de leurs systèmes. La notion d'architecture devient vitale car elle pose un trait d'union entre le cahier des charges d'une application (les besoins), les méthodes de conception et la mise en œuvre. Elle permet de remonter, de manière compréhensible et synthétique, la complexité d'un système tout en ouvrant les portes pour satisfaire les besoins en réutilisation, et en offrant une certaine souplesse d'évolution des applications. Cette souplesse se traduit par la possibilité de supprimer, remplacer et reconfigurer des composants sans perturber les autres parties de l'application. La conception d'une bonne architecture logicielle peut amener à un produit qui répond aux besoins exprimés et qui peut être facilement modifiable pour ajouter une nouvelle fonctionnalité, alors qu'une architecture inappropriée peut avoir des conséquences désastreuses jusqu'à l'arrêt du projet [33].

L'architecture logicielle correspond à l'établissement d'un plan de construction du logiciel. Il est alors plus facile de gérer et d'anticiper les éventuelles évolutions. En effet, la modification d'un plan est plus simple que la modification d'un système complet. L'architecture logicielle est beaucoup plus visible et devient une activité de conception explicite dans le processus de développement. De nombreuses recherches ont ainsi fait apparaître l'architecture logicielle comme une filière explicite du génie logiciel. Ce domaine tire partie des travaux des années 70

et 80 décrits précédemment, mais il se veut plus formel et plus orienté sur l'architecture des systèmes que sur le code. Bien que cette filière existe depuis plus de dix ans, elle reste récente dans le processus de développement et il n'existe pas encore de définition de l'architecture logicielle qui soit acceptée par tous. Nous avons repris une définition générale tiré du livre de Shaw et Garlan [83] qui fut le premier livre consacré à l'architecture logicielle :

« L'architecture d'un système logiciel définit ce système en termes de composants de calcul et des interactions de ces composants². »

Les auteurs définissent l'architecture logicielle de manière assez simple comme un ensemble de composants, d'interactions et une configuration de ceux-ci guidée par des critères de choix. Sur cette base, les recherches effectuées sur ce domaine de l'architecture logicielle ont permis d'améliorer la formalisation de l'architecture d'un système en définissant des notations formelles et en produisant des outils d'analyse de spécification architecturale. Elles ont permis de réaliser de nombreuses avancées, notamment par la reconnaissance de deux entités indépendantes dans l'architecture, une dédiée aux calculs appelée « composant », et l'autre dédiée aux interactions entre ces composants, souvent assimilée à des « connexions » ou « connecteurs ». L'interconnexion des composants constitue la configuration de l'application. Ainsi, la définition d'une architecture comme un ensemble de composants interconnectés (par des connecteurs ou des connexions) devient largement accepté dans la communauté.

Pour définir, spécifier et manipuler des architectures logicielles, de nombreux travaux ont défini des langages et développé des outils pour permettre la construction des applications. Il s'agit principalement de la définition des langages de description d'architecture (*Architecture Description Languages* - ADLs) dans le domaine académique, et des modèles à composants [71] dans le domaine industriel. Toutefois, cette discipline est loin d'avoir atteint sa maturité et plusieurs pistes restent à explorer. D'un côté, comme leur nom l'indique, les modèles de composants se focalisent essentiellement sur la définition et la formalisation des descriptions des composants des architectures globales. S'il existe un quasi-consensus sur ce qu'est un composant, la situation n'est pas équivalente concernant la communication entre ces composants. D'un autre côté, bien que la classification de référence des ADLs [56] définit trois éléments essentiels dans un langage de description d'architecture — à savoir le composant, le connecteur et la configuration — la plupart des approches se focalisent sur les composants et définissent des modèles de connexion spécifiques à ces composants. Il existe ainsi des représentations et des considérations contradictoires des interconnexions entre les composants. Dans notre travail, nous nous intéressons aux interactions entre les composants et à la définition d'un modèle générique de connecteur.

Ainsi, et avant d'exposer notre point de vue sur les connecteurs, il est nécessaire de présenter comment les connexions sont considérées dans les différentes approches de l'architecture logicielle. Dans la suite de ce chapitre, nous allons présenter quelques concepts de base pour chacune des approches et nous focalisons sur la façon dont elles expriment et considèrent les interactions entre les composants. Nous les abordons selon qu'elles intègrent ou n'intègrent pas explicitement un modèle de connexion, c'est-à-dire selon que les interactions y sont réifiées ou non.

²“The architecture of a software system defines that system in terms of computational components and interactions among those components.”

1.2 Les interactions comme entité non réifiée

Nous traitons dans cette section les approches qui ne réifient pas et n'explicitent pas les interactions dans une entité indépendante. Pour ce faire, nous distinguons deux familles d'approches : les approches académiques et les approches industrielles.

1.2.1 Approches académiques

Les approches académiques qui ne réifient pas les interactions dans une entité à part regroupent un ensemble de langages de description d'architecture issus des universités et des organismes de recherche. Nous allons décrire dans cette section quelques ADLs que nous considérons comme étant les plus représentatifs et les plus référencés : Darwin et Fractal issus des universités et organismes de recherche européens et Rapide issu d'une université américaine.

Darwin

Darwin [49, 51] est un langage de description d'architecture, plus souvent appelé langage de configuration, créé à l'Imperial College de Londres. Ce langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description de la structure de l'architecture d'un système.

Dans Darwin, un programme se compose d'un ensemble de types de composants avec plusieurs instances de ces types qui communiquent entre eux. Ce langage décrit un type composant par une interface composée d'une collection de services fournis ou requis. Les services d'un type composant s'apparentent plus aux entrées et sorties de flots de communication qu'à la notion de fonction. Ainsi, le protocole de communication sur ces connexions est uniquement du type envoi de message. Chaque service fourni est modélisé comme un nom de canal, alors que chaque déclaration de liaison est un processus qui transmet le nom de ce canal à un composant qui requiert le service.

Deux types de composants existent. Le premier correspond aux composants *primitifs* qui intègrent du code logiciel. Le second aux *composites* qui sont des entités de configuration décrivant les interconnexions entre les services fournis et requis des composants primitifs ou composites. La figure 1.1 montre la description formelle (et graphique) d'un composant composite.

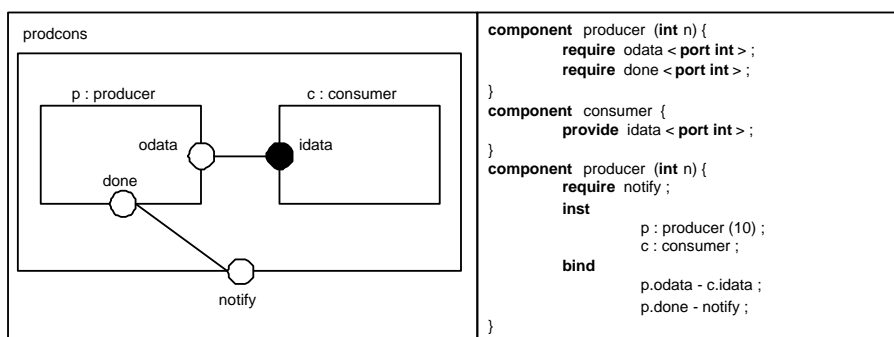


Figure 1.1 — Description d'une architecture Darwin

Ainsi, du point de vue Darwin, une configuration n'est rien d'autre qu'un composant composé de sous-composants liés entre eux. Les composants composites sont définis en déclarant les instances d'autres composants qu'ils contiennent et les liaisons entre eux. Les services requis ou fournis correspondent à des types d'objets de communication que le composant utilise pour communiquer avec un autre composant ou pour recevoir une communication d'un autre composant. Ces objets peuvent être contraints par des types de communication et des types de données. La section *bind* dans la figure ci-dessus montre les interconnexions entre les composants comme des interactions simples (une simple transmission de nom de canal).

Analyse. Dans Darwin, les interactions entre les composants sont spécifiées comme des liaisons directes entre les interfaces requises et offertes des composants impliqués dans l'application. La sémantique d'une connexion est définie par l'environnement sous-jacent, et les composants communicants doivent en tenir compte. Darwin est considéré comme le langage typique représentant les ADLs qui utilisent des connexions implicites entre les composants. Il a gardé comme objectif l'implémentation, il reste alors très dépendant de la plate-forme. Ce langage est couplé à l'environnement Regis [50] qui le transforme en C++, donc à l'exécution on se retrouve confronté aux problèmes des langages de programmation.

Le langage Darwin offre la possibilité de capitaliser les calculs dans les composants en définissant les systèmes comme un ensemble de composants et les configurations comme des composants composites. Cependant, son support pour les descriptions architecturales reste limité à cause de la faible sémantique de ses connexions. Le modèle de connexion « fourni/requis », avec des liaisons directes, impose un modèle *asymétrique* d'interaction. Il implique qu'il y ait toujours une partie dans le composant qui soit responsable de la définition du protocole. Ainsi, les interactions ne peuvent pas être décrites indépendamment du composant qui les fournit. Il fournit une syntaxe riche pour décrire plus finement les interactions avec les instructions *forall* et *when* pour décrire la coordination. Cependant, comme la notion de connecteur comme entité de première classe n'est pas supportée par Darwin, ces instructions décrivent des interactions pour des composants spécifiques. Ces interactions ne sont donc pas réutilisables.

Fractal

Fractal [16, 25] est une infrastructure générale de composition de logiciel du consortium Objectweb [63] qui supporte la programmation à base de composants. Elle est née d'une collaboration entre France Telecom R&D et l'INRIA. C'est un modèle de composant qui permet la configuration et la reconfiguration dynamique de composants. Il offre également une séparation claire entre les besoins fonctionnels et non fonctionnels d'une application. Comme Darwin, ce modèle est hiérarchique et récursif : un composant est de type primitif ou composite. Dans ce dernier cas, le composant correspond à un assemblage d'autres composants primitifs ou composites. Par contre, une particularité des composants Fractal est qu'ils peuvent être partagés entre différents composites. Cet assemblage constitue la configuration de l'application. Ce langage peut ainsi être apparenté à un langage de description d'architecture. En effet, les descriptions des configurations de composants sont basées sur XML (*eXtensible Markup Language*).

Les interfaces jouent un rôle central dans Fractal. Il en existe deux catégories : les interfaces fonctionnelles et celles de contrôle. Les interfaces fonctionnelles sont les points d'accès externes d'un composant. Fractal offre des interfaces client et serveur. Une interface serveur reçoit des

opérations d’invocation et une interface client en émet. Ainsi, une liaison Fractal représente une connexion entre deux composants (liaison primitive). Des liaisons de type multiples sont autorisées (liaisons composites). Ces dernières représentent un ensemble de liaisons simples et de composants et relient plusieurs interfaces locales ou distantes. Les interfaces de contrôle quant à elles, et comme leur nom l’indique, permettent un certain niveau de contrôle du composant auquel elles sont associées. Ces interfaces ont à charge les besoins non fonctionnels du composant, par exemple la gestion du cycle de vie ou des liaisons à d’autres composants. La figure 1.2 montre une représentation d’une architecture Fractal.

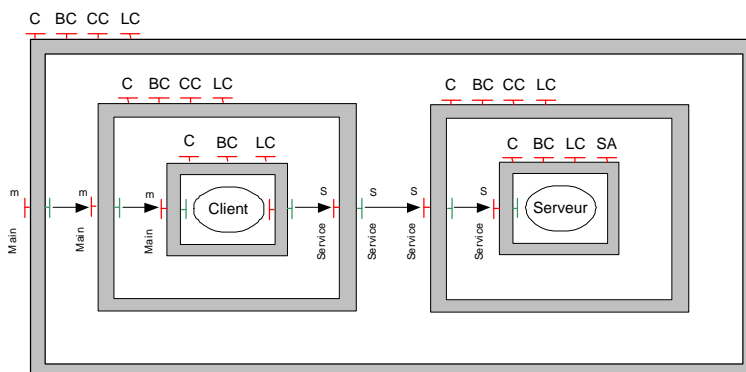


Figure 1.2 — Description d’une architecture Fractal

Analyse. Fractal ne considère que les composants et leur structure, par conséquent un système n’est rien d’autre qu’un groupe de composants reliés entre eux. Les interactions entre composants n’ont pas suscité beaucoup d’attention. En effet, dans Fractal les connecteurs sont définis implicitement et leur sémantique est enfouie au sein des composants. Les connexions sont représentées par des liaisons point-à-point, dans une clause *bindings*, sans aucun comportement associé. Donc, de ce point de vue, il présente les mêmes inconvénients que le modèle d’interaction « fourni/requis » de Darwin. En revanche, il n’existe pas de dépendance du modèle abstrait par rapport aux langages de programmation. Notons que la simplicité du modèle, qui se traduit par la définition d’un modèle de composant uniquement sans la définition d’un modèle d’interconnexion, fait qu’il existe plusieurs implémentations disponibles. Il existe une implémentation de référence appelée Julia [15].

Rapide

Rapide développé par Luckham [47, 48] et son équipe à l’Université de Stanford, est un langage de description d’architecture d’un système distribué. Il fournit les constructions nécessaires à la création de prototypes exécutables. Le résultat de l’exécution forme un PO-SET (*Partially Ordered Set*) qui est l’ensemble des événements produits pendant l’exécution avec leurs relations causales et temporelles.

Pour la description d’une application, Rapide dispose de cinq langages de description distincts :

- Le langage des *types* permet de décrire les interfaces des composants ;
- Le langage d’*architecture* permet de construire la configuration de l’application par la mise en place des connexions entre les interfaces ;
- Le langage des *modules exécutables* permet de programmer les composants ;
- Le langage des *contraintes* permet de décrire le comportement des composants en termes

de POSET ;

- Le langage de *gabarit* permet de décrire des gabarits d'événements.

Un composant est défini par une interface (décrit par le langage des types) définissant les caractéristiques d'interaction avec les autres composants, et par un module qui encapsule soit un prototype exécutable (décrit par le langage des modules), soit d'autres composants (décrits par le langage d'architecture). Une interface est constituée d'un ensemble de services fournis et d'un ensemble de services requis qui sont de trois types : les services *Provides* et *Requires*, fournis et demandés par le composant et appelés de manière synchrone ; et les *Actions* qui correspondent à des appels asynchrones entre composants. Une interface permet de décrire le comportement d'un composant (*Behavior*) et des contraintes (*Constraints*).

Une architecture qui est un module, est décrite par le langage d'architecture. Elle est composée d'un ensemble de composants avec leurs interfaces, et éventuellement avec la description du module associé, et d'un ensemble de connexions. Une connexion relie les constituants requis et fournit des interfaces des composants entre eux. Elle établit un échange de données et de synchronisation entre les composants. La connexion est composée de deux parties, gauche et droite, reliées par trois types d'opérateurs possibles. Le premier opérateur, « *to* », est dit basique et connecte deux événements. Il ne permet qu'un événement possible vers un composant. Si la partie gauche est vérifiée alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression. Le second opérateur, « *|| >* » dit de diffusion, connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Le dernier opérateur, « *=>* » de type pipe-line, est identique au précédent mais l'ordre de déclenchement des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette même règle. Le langage offre aussi des facilités syntaxiques pour permettre la connexion entre un ensemble de composants.

Analyse. Rapide est un ADL dont l'outillage fournit des capacités intéressantes pour la modélisation et l'analyse. Il permet de vérifier, par la simulation, la validité d'une architecture logicielle donnée en confrontant l'ensemble des propriétés obtenues, formant un *POSET*, aux propriétés souhaitées. Il offre également des sémantiques de communication plus riches avec les communications synchrones, asynchrones, et les opérateurs qui permettent de regrouper des destinataires. Cependant, Rapide ne modélise pas les connexions comme des entités de première classe. Ceci limite leur réutilisabilité et rend leur vérification difficile, car chaque connexion doit être analysée individuellement. Les stratégies et directives d'implémentation sont requises pour chaque connexion individuelle et non pas pour chaque type de connecteur.

Une particularité de Rapide est la possibilité de réaliser des interactions complexes mais sous forme de composant avec la notion de « *connection component* ». La description du comportement d'un composant, avec la clause *Behavior*, permet ainsi de décrire une coordination. Cependant, ces descriptions concernent des composants spécifiques et ne sont pas réutilisables.

Ces trois modèles que nous avons vus (Darwin, Fractal, Rapide) ont amélioré la compréhension des architectures mais le fait qu'ils ne définissent pas explicitement l'interaction les rend peu flexibles. Ces avancées académiques n'ont pas percé dans l'industrie. Les modèles à composants se sont plus rapidement répandus et sont utilisés plus largement.

1.2.2 Approches industrielles

Les approches industrielles que nous allons évoquer concernent les modèles à composants comme les Enterprise Java Beans (EJB), le modèle de composants CORBA (CCM) et les composants .NET. Nous étudions dans cette section les deux premiers modèles qui constituent les deux modèles de référence. Ces modèles n'explicitent pas la notion de connecteur et utilisent des connexions simples pour exprimer les interactions entre les composants. Pour leurs communications, ces modèles reposent sur les intergiciels. Nous allons voir dans ce qui suit les principales caractéristiques de chacun de ces modèles.

Enterprise Java Beans (EJB)

À la fin des années 1990, Sun Microsystems avait commencé à travailler sur plusieurs modèles de composants pour le développement d'applications commerciales distribuées basées sur le langage Java [7]. En 1997 fut présentée une première version qui a été ensuite largement adoptée dans la communauté industrielle : le modèle Enterprise Java Beans (EJB) [86].

Les EJB sont des composants utilisés dans le cadre de l'architecture J2EE (*Java 2 Enterprise Edition*). Ces composants permettent de mettre en œuvre la logique métier d'une application codée suivant une architecture logiciels 3 tiers. Il s'agit d'une séparation entre la couche présentation (client lourd ou léger), la couche métier (mise en œuvre des composants), et la couche de données. Ce modèle à composants est centré sur la définition du serveur et distingue trois profils de composants appelés *bean* :

- Un *bean* session représente une session établie par le client du côté du serveur. Ce *bean* offre dans son interface les services offerts par le serveur. Sa durée de vie est liée à celle du client auquel il est associé. Il peut avoir un état associé qui est géré en mémoire ou non (*statefull* et *statless*). L'état d'un *bean* session est perdu en fin de session ;
- Un *bean* entité est une ressource partagée et persistante utilisée pour représenter des données stockées dans un système de stockage persistant, typiquement dans une base de donnée. À la différence d'un *bean* session, la durée de vie d'un *bean* entité n'est pas liée aux clients ;
- Un *bean* conducteur de message a été conçu pour gérer les messages asynchrones. Ce type de *bean* n'a pas d'état interne, n'est pas persistant et peut être partagé entre plusieurs clients. Ce sont des instances neutres réagissant à l'arrivée d'un message.

Analyse. En plus de ces diverses descriptions de *bean*, le modèle EJB offre une infrastructure riche pour le serveur. Ceci est réalisé grâce à un conteneur qui fournit les interfaces nécessaires pour les services système tels que les transactions, la persistance et la sécurité, tout en garantissant la transparence par rapport à la plate-forme et au système sous-jacent. Le conteneur permet d'intercepter les requêtes venant du client pour que les services système puissent les exécuter. Il permet ainsi d'abstraire le code applicatif du code technique ainsi que la prise en charge automatique des liaisons entre composants.

En revanche, il offre un support limité pour les interactions et une sémantique de communication faible entre le client et le serveur. Les communications entre un client et un serveur s'appuient sur l'intergiciel RMI (pour *Remote Method Invocation*). Cet intergiciel offre une gestion très intéressante de la distribution et de l'hétérogénéité des plates-formes système mais il ne permet d'assurer que la communication synchrone qui constitue la principale sémantique de communication offerte pour les EJB. La communication asynchrone est assurée grâce au *bean* conducteur de message. Toutes les applications doivent ainsi s'adapter à ces deux types de communication [12]. Les connexions sont décrites avec un descripteur XML lors la descrip-

tion des composants. L'assemblage des composants est décrit d'abord par la description de dépendances envers les autres composants (avec la clause `<ejb-ref>`) et des liaisons point-à-point (avec la clause `<ejb-link>`). Les connexions ne sont établies qu'au déploiement.

Corba Component Model (CCM)

Le modèle CCM (*CORBA Component Model*) est un modèle de composant similaire à EJB qui constitue une partie de la spécification CORBA 3.0 [69, 84] de l'OMG (*Object Management Group*). Comme EJB, le modèle CCM a été conçu pour la création des applications industrielles nécessitant le support pour des services systèmes tels que la sécurité, la persistance et les transactions. Il étend le modèle à objets CORBA par une notion de composants qui explicite les interfaces fournies (*facettes*) et requises (*réceptacles*) ainsi qu'un mécanisme d'abonnement et d'observation d'événements (*source* et *puits d'événement*) par le biais de l'IDL3. Le modèle définit des services qui permettent aux développeurs d'implanter, gérer, configurer et déployer des composants qui s'intègrent avec les services CORBA. Comparé à EJB, il apporte les principaux avantages qu'offre CORBA tel que le support multi-langages, multi-ORBs et multi-fournisseurs. Il offre plus d'ouverture, d'interopérabilité, d'hétérogénéité et de portabilité. La figure 1.3 montre un exemple d'assemblage d'un ensemble de composants CCM et leurs propriétés.

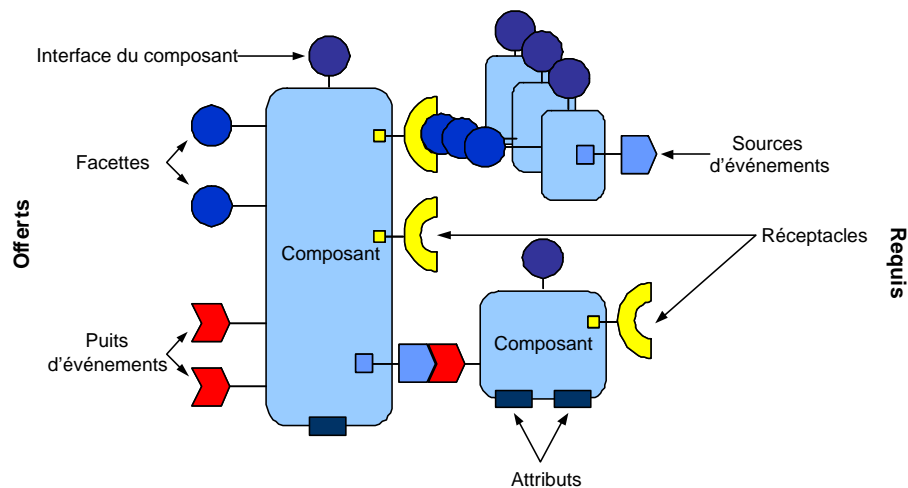


Figure 1.3 — Description d'une architecture CCM

Analyse. Concernant les connexions, CCM est aussi pauvre que EJB en matière d'abstraction de communication, seules les deux sémantiques de communication synchrones, entre *facettes* et *réceptacles*, et asynchrones, entre *source* et *puits* d'événements, sont disponibles.

Dans la figure 1.3, on remarque qu'un *réceptacle* peut contenir plusieurs *facettes*, c'est-à-dire qu'une interface requise peut obtenir un service de plusieurs interfaces fournies. Les communications ne sont plus restreintes aux connexions point-à-point comme dans EJB. L'interface `MULTIPLE` qui existe dans l'IDL3, permet de décrire de telles connexions. Cependant, la sémantique de cette communication multiple ne peut pas être décrite, elle est laissée aux soins des développeurs. En effet, il n'est pas possible de spécifier s'il y a une priorité de traitement de ces connexions ou une politique particulière à appliquer. Elle peut ainsi avoir plusieurs interprétations et laisse planer une ambiguïté sur la sémantique de ces connexions.

1.3 Les interactions comme entité réifiée

Après avoir recensé quelques approches centrées sur la définition des composants, nous décrivons dans cette section les approches qui définissent les interactions entre les composants comme des entités à part entière. Ces approches concernent principalement les langages de description d'architecture. Nous les avons regroupés suivant qu'ils offrent un nombre limité de connecteurs ou qu'ils offrent la possibilité à l'utilisateur de définir de nouveaux connecteurs.

1.3.1 Approches avec un nombre limité de connecteurs

Nous décrivons dans cette section les langages de description d'architecture UniCon et C2 qui réifient les connexions entre les composants logiciels dans une entité appelée « connecteur » mais ne définissent qu'un ensemble limité de ces connecteurs.

UniCon

UniCon (*Universal Connector Support*) est un langage de description d'architecture créé à l'Université de Carnegie Mellon en Pennsylvanie (USA) et dont les principaux acteurs sont Mary Shaw et Gregory Zelesnik [81, 82]. Ce langage est fondé sur les trois concepts de l'architecture logicielle : le composant, le connecteur et l'architecture.

Le composant dans ce langage est une unité d'encapsulation des fonctions d'un module logiciel ou de données. Il est caractérisé par une interface qui définit les services requis et fournis ainsi qu'une implémentation. L'interface d'un composant consiste en : le type du composant ; les propriétés qui spécialisent le type ; et une liste de points d'interaction avec le monde extérieur appelés *players*. Ces points d'interaction sont typés suivant le type du composant. L'implémentation d'un composant peut être primitive, qui consiste en des éléments extérieurs au domaine d'UniCon comme un fichier source dans un langage de programmation ; ou composite, qui consiste en un ensemble de composants et de connecteurs composés.

Le connecteur permet de spécifier les règles d'interconnexion de composants. Sa description suit la même approche que le composant, il est caractérisé par un protocole et par une implémentation. Le protocole consiste en : le type du connecteur, choisi dans la liste fournie par UniCon ; les propriétés qui spécialisent ce type ; et une liste de points d'interaction, appelés *roles*.

L'architecture d'une application est définie par un composant composite avec une interface de type général (*General*) qui ne spécifie rien. Sa partie implémentation instancie les composants à composer et connecte les *players* des composants aux *roles* des connecteurs.

Analyse. Pour ce qui est des connexions, UniCon fait partie des rares ADLs qui se penchent particulièrement sur l'implémentation des connecteurs. Cependant, il ne supporte que des connecteurs prédéfinis³ et non extensibles. De ce fait, l'implémentation des connecteurs est spécifiée comme intégrée (*built-in*). Ceci les rend propriétaires à ce langage, dans le sens où ces connecteurs ne sont utilisables que par des composants définis avec le langage UniCon, ce n'est pas un langage pivot comme l'est par exemple l'IDL. Les connecteurs sont bien spécifiés au niveau architecture mais ils correspondent à des connecteurs triviaux définis dans

³Canal (*pipe Connector*), appel (s) de procédure (*ProcedureCall Connector*), appels de procédure distants (*RemoteProcCall Connector*), accès de données (*DataAccess Connector*), ordonnancement de processus temps réel (*RTScheduler Connector*).

la plupart des systèmes. Ainsi, il n'est pas possible, pour l'instant, d'associer au connecteur UniCon une implémentation créée sans l'aide de cet ADL.

Les rôles sont fortement typés, chaque type de rôle n'autorise que le branchement d'un type *player*. Par exemple, le rôle *Definer* du connecteur *RemoteProcCall* n'autorise que le branchement du *player* de type *RPCDef*. Le langage fournit un compilateur qui effectue plusieurs vérifications. Il peut vérifier que l'interface d'un composant ou le protocole d'un connecteur est compatible avec son type. Il peut également vérifier la compatibilité des types à la connexion. Cependant, Il n'est pas possible de changer de sémantique de communication en changeant simplement de connecteur sans toucher aux composants puisque le type du connecteur à utiliser est spécifié dans le code du composant par le type du *player*. Ce langage manque ainsi de flexibilité. Il n'est pas suffisamment abstrait et reste dépendant de l'implémentation.

C2

C2 est un langage de description d'architecture conçu par l'Université UCI en Californie [54, 55, 57]. Il fut conçu à l'origine pour la conception d'interfaces graphiques et s'étend aujourd'hui à la conception d'autres applications. Il propose une manière de spécifier une architecture en définissant trois abstractions principales qui sont le composant, le connecteur et la configuration d'une architecture. L'idée de cet ADL est de définir une application sous forme de réseau hiérarchique de composants s'exécutant de manière concurrente, liés par des connecteurs et communiquant de manière asynchrone par envoi de message.

Un composant défini dans une architecture C2 est à l'écoute des composants qui sont situés au-dessus de lui et produit des événements pour les composants situés en dessous de lui. Chaque composant et chaque connecteur possèdent un haut (*top*) et un bas (*bottom*). Le haut d'un composant doit être connecté au bas d'un seul connecteur et le bas d'un composant est connecté au haut d'un connecteur unique. Ainsi, un composant est rattaché à l'architecture d'un système grâce à deux connecteurs dont l'un lui permet d'envoyer des événements à d'autres composants à travers sa partie *bottom*, et l'autre de recevoir des événements d'autres composants à travers sa partie *top*. Il n'y a pas de limites au nombre de composants liés à un connecteur.

Les connecteurs servent ainsi dans C2 à faire communiquer des composants en échangeant des messages. Ils permettent, d'une part de diffuser les messages (demande de requêtes ou événements) destinés aux composants, et d'autre part de filtrer certains messages. Un connecteur offre un nombre limité de politiques de filtrage parmi celles citées ci-dessous :

- Aucun filtrage : chaque message est envoyé à tous les composants qui se sont reliés au connecteur ;
- Filtrage notifié : chaque événement est envoyé à tous les composants qui se sont abonnés à cet événement ;
- Filtrage conditionnel : l'événement est envoyé aux composants selon certaines conditions données.

La spécification d'une application est composée de deux parties : la spécification de l'architecture logicielle et la spécification de l'implémentation. La spécification de l'architecture logicielle dresse la liste des composants de l'application, la liste des connecteurs qui assurent les interactions entre ces composants, et la topologie qui décrit l'assemblage statique des connecteurs avec les composants : pour chaque connecteur sont décrits les composants qui viennent au-dessus (via *top*) et au-dessous (via *bottom*). La spécification de l'implémentation

est formée de la liste des composants, où chacun est associé à une seule implémentation. Il est à noter qu'aucune description d'implémentation n'est associée aux connecteurs.

Analyse. C2 présente des caractéristiques intéressantes sur les interconnexions entre composants notamment par la définition de connexion n-aires. En effet, les connecteurs définis permettent de connecter plusieurs composants qui accèdent par le dessus du connecteur. Celui-ci peut envoyer des messages à plusieurs composants qui sont en dessous en appliquant les différentes techniques de filtrage. Il permet ainsi de décharger les composants de la responsabilité de gérer les communications avec d'autres composants. De plus, il permet la séparation entre l'architecture et l'implémentation tout en visant leur rapprochement [26]. Cependant, C2 n'offre pas une liberté de choix de sémantique de communication. Celle-ci est limitée à l'envoi de messages entre les composants. Par exemple, il n'est pas possible de spécifier l'utilisation d'un RPC au niveau architecture. Cette sémantique simple à besoin d'une combinaison de deux connecteurs pour être assurée. Ceci est dû au style hiérarchique contraignant. De plus, les connecteurs n'offrent que deux rôles possibles (*top* et *bottom*). Il n'est donc pas possible de spécifier au niveau architecture une coordination entre les composants ou des propriétés non fonctionnelles.

1.3.2 Approches avec des connecteurs définis par l'utilisateur

Nous décrivons dans cette section les langages de description d'architecture Wright et ArchJava qui définissent les connexions entre les composants logiciels comme des connecteurs. Ces langages permettent la définition de nouveaux connecteurs définis par l'utilisateur.

Wright

Wright [5, 4] est un langage de description d'architecture développé à l'Université de Carnegie Mellon par Allen *et al.* Il se focalise sur les spécifications formelles de comportement des composants et des connecteurs en utilisant la notation de l'algèbre de processus séquentiels communicants (*Communicating Sequential Processes* - CSP) [39]. Il repose sur les notions de composant, connecteur et configuration.

La description d'un composant possède deux parties importantes : l'interface et le calcul (*computation*). Les interfaces, à base d'événements des composants, sont composées d'un certain nombre de ports où chacun des ports représente une interaction dans laquelle le composant peut participer. Une interface peut contenir des événements émis et reçus, il n'y a pas de distinction entre les interfaces fournies et requises. La partie calcul décrit ce que le composant fait réellement. Elle met en œuvre les interactions décrites par les ports et montre comment ils sont attachés ensemble pour former le comportement du composant, elle décrit comment les ports interagissent entre eux.

Le connecteur représente une interaction parmi une collection de composants. Sa description se résume à un ensemble de rôles et une glu. Chaque rôle spécifie le comportement d'un participant dans l'interaction. Les rôles spécifient ce qui est attendu des composants qui participent à cette interaction. La glu d'un connecteur décrit comment les participants (c'est-à-dire les rôles) interagissent entre eux pour créer une interaction

Une configuration est une collection d'instances de composants liés par des connecteurs. Une fois les instances déclarées, une configuration est complétée en décrivant les attachements. Ces derniers définissent la topologie de la configuration en montrant quels composants

participent à quelles interactions. Un exemple d'une spécification d'architecture d'une application client/serveur (où seuls les protocoles du connecteur sont décrits) est présenté dans la figure 1.4.

```

System ClientServerSystem
  component Server =
    port provide [provide protocol]
    spec [Server specification]
  component Client =
    port request [request protocol]
    spec [Client specification]
  connector C-S-connector
    role client = (request !x -> result ?y -> client)
    role server = (invoke ?x -> return !y -> Server)
    glue = (Client.request ?x -> Service.invoke !x ->
            Service.return ?y -> Client.result !y-> glue)

Instances
  s : Server
  c : Client
  cs : C-S-connector
Attachments
  s.provide as cs.server
  c.request as cs.client
end ClientServerSystem

```

Figure 1.4 — Description d'une architecture Wright

Wright a apporté une avancée considérable pour la formalisation des interactions entre les composants en les décrivant avec les protocoles. Dans le code Wright précédent, la description du connecteur client/serveur évoque le comportement de chaque élément du connecteur :

- Le client doit recevoir une requête et renvoyer un résultat ;
- Le serveur doit envoyer une invocation et recevoir un retour ;
- La glu renvoie la requête du client pour invoquer le serveur puis renvoie le retour du serveur vers le résultat du client.

Analyse. Ce langage permet de vérifier la compatibilité entre les ports et les rôles et de détecter des anomalies dans les comportements des composants et des connecteurs, comme par exemple des éventuels interblocages entre les protocoles. De plus, cet ADL offre le moyen aux utilisateurs de Wright, qui maîtrisent l'outil CSP, de décrire leurs propres types de connecteurs (*user-defined*) qui permettent de capturer une variété d'interactions complexes entre les composants. Il fournit des descriptions des composants et des connecteurs abstraites et indépendantes de l'implémentation. Cependant, l'outil CSP n'est pas facile à assimiler par un débutant. Ceci pourrait altérer les communications à propos de l'architecture d'une application entre les diverses personnes impliquées dans un projet. De plus, Wright supporte la description des styles d'architecture et des instances d'architecture sous forme de texte seulement. Il permet la spécification et la description d'interactions complexes mais ne fournit aucune notion sur comment réaliser cette interaction. Les connecteurs et les composants ne correspondent pas à des éléments exécutables, et pour l'instant aucun environnement ou outil d'exécution n'est associé spécifiquement à ce langage. L'indépendance entre la spécification et l'implémentation des connecteurs ainsi que l'absence d'outils pour cette projection implique que la réalisation des connecteurs se fait à l'aide des outils existants en utilisant des moyens d'interaction simples fournis par les plates-formes. Cette réalisation fait perdre à l'interaction sa caractéristique d'entité de première classe, et elle se retrouve ainsi dispersée avec les autres composants.

ArchJava

ArchJava est un langage de description d'architecture conçu à l'Université de Washington [2]. Il s'agit d'une extension de Java qui permet à des programmeurs d'exprimer la structure architecturale d'une application dans le code source. À cet effet, ArchJava a ajouté de nouvelles constructions de langage pour supporter les composants, les connexions et les ports. Il permet d'explicitier les connexions entre les composants dans le code source contrairement aux langages orientés objet comme Java dont les connexions n'apparaissent qu'à l'exécution. De plus, les concepteurs d'ArchJava proposent, dans [3], d'étendre le langage avec des connecteurs définis par l'utilisateur. Concernant les configurations, le langage repose sur les composants composites pour les décrire. En effet, ce sont les composants composites qui contiennent la description des composants et leurs interactions. Une application est ainsi considérée comme un composant composite.

Un composant dans ArchJava est une sorte spéciale d'objet qui communique avec les autres composants d'une manière structurée à travers des ports. Un port représente un canal de communication logique entre un composant et un ou plusieurs connecteurs auxquels il est connecté. Les ports déclarent deux ensembles de méthodes : requises et fournies. Une méthode fournie est implémentée par le composant. Elle est disponible pour être appelée par d'autres composants connectés à ce port. Inversement, chaque méthode requise est fournie par un autre composant connecté à ce port. Les connexions entre les ports se font par des patterns de connexion. Le système de vérification de type assure que pour chaque méthode requise par un ou plusieurs des ports connectés, il existe exactement une seule méthode fournie correspondante avec le même nom et signature. Donc la principale sémantique offerte est l'invocation.

Analyse. À la base, ArchJava offre des connexions simples. Il a été étendu pour pouvoir exprimer des abstractions de connecteurs définies par l'utilisateur pour ne pas se contenter et être contraint par les sémantiques de communication offertes par la plate-forme (Java), qui ne font que relier les méthodes requises aux méthodes fournies. La syntaxe du langage a été étendue pour pouvoir exprimer l'abstraction de connecteur qui doit être utilisée. De nouveaux connecteurs peuvent être écrits en utilisant la librairie `archjava.reflect` qui réifie les connexions. L'idée consiste à indiquer explicitement l'endroit où il faut utiliser une abstraction de connecteur. À ce moment-là, dès que l'interpréteur ArchJava conventionnel rencontre une instruction utilisant une abstraction de connecteur, l'interpréteur ArchJava est remplacé par l'interpréteur de l'abstraction de connecteur pour la vérification du typage. À la fin, l'interpréteur principal d'ArchJava reprend son travail. Donc contrairement à Wright, ArchJava offre la possibilité de définir de nouvelles abstractions et propose, de plus, une solution pour les implémenter. Il permet de garder l'abstraction de communication jusqu'à un niveau tardif. Celle-ci n'est pas perdue car elle est réalisée avec des connexions simples et point-à-point. Cependant, comme nous le verrons dans le prochain chapitre, les communications point-à-point engendrent des ambiguïtés. En effet, il n'est pas mentionné comment les communications faisant intervenir plusieurs composants sont traitées ni comment gérer les communications lorsque plusieurs interpréteurs sont sollicités. Ceci laisse sous-entendre que ces problèmes sont gérés par les composants eux-mêmes. De plus, les abstractions sont décrites dans ce langage spécifique, il y a alors dépendance à la plate-forme sous-jacente.

1.4 Synthèse

Nous avons présenté dans ce premier chapitre un panel de modèles de composants et de langages de description d'architecture qui permettent de décrire des applications comme un ensemble de composants interconnectés. Il s'agit d'un domaine très vaste et varié et il peut y avoir différentes façons de l'aborder. Comme nous attachons une attention plus particulière aux interconnexions entre les composants, nous avons appuyé notre étude sur la considération et la représentation des interactions entre les composants selon que les approches les réifient dans des entités explicites ou non. Pour chaque approche, nous avons donné les principales caractéristiques des entités manipulées et nous avons discuté les interactions.

Les premières approches que nous avons étudiées concentrent les efforts de descriptions des architectures sur les composants logiciels et leurs configurations. Elles regroupent les langages et modèles qui ne réifient pas les interactions entre les composants dans des entités à part entières. La configuration d'une application est représentée comme un ensemble de composants reliés par des connexions. Ces approches offrent une sémantique de communication restreinte et très limitée. Certains langages (comme Darwin) offrent une sémantique de communication très basique par envoi de messages. Les autres langages et modèles (Rapide, Fractal, EJB, CCM) augmentent cette sémantique en définissant des communications par événements ainsi que des communications synchrones et asynchrones.

Étant donné que les composants contiennent souvent des fonctionnalités complexes, il est raisonnable de s'attendre à ce que leurs interconnexions le soient également. Cependant, avec les sémantiques restreintes qu'offrent ces langages qui permettent de relier que des services fournis à des services requis (communications point-à-point), il n'est pas possible de décrire ou de réaliser des interactions complexes et de les réutiliser. Cette complexité est réalisée par les composants qui se trouvent obligés de réaliser des propriétés qui ne sont pas liées à leur fonctionnalité. Ceci limite la séparation des responsabilités et peut même restreindre la réutilisabilité des composants. Certains langages et modèles à composants (comme Fractal et CCM) offrent des moyens pour assurer certaines interactions complexes qui font intervenir plusieurs composants (multiples). Cependant, comme elles ne sont pas explicites, elles ne permettent pas de préciser la propriété de la communication. Ces interactions engendrent des ambiguïtés d'interprétation ce qui peut aboutir à des divergences d'implémentations. En revanche, comme ils offrent des interactions simples, la plupart de ces langages et modèles offrent un support d'implémentation.

Les secondes approches que nous avons étudiées considèrent explicitement les interactions et les réifient dans des entités appelées « connecteurs ». Certaines approches offrent un nombre limité des connecteurs (Unicon et C2) et d'autres offrent le moyen d'en créer de nouveaux (Wright et ArchJava). Ces approches permettent ainsi la réutilisation de certaines interactions. Cependant, ces approches sont confrontées à des problèmes d'implémentation. En effet les approches qui définissent un nombre limité de connecteurs offrent les implémentations de connecteurs simples. D'un autre côté, les approches qui permettent de décrire des interactions complexes ne fournissent pas de moyens pour les réaliser, ce qui revient à les implémenter avec les moyens disponibles et simples. Ceci cause la perte de l'abstraction de communication durant le processus de développement en la mélangeant avec les composants fonctionnels.

L'étude de ces modèles a montré le caractère instable de la prise en compte des interactions entre les composants logiciels ainsi qu'une insuffisance relative à l'implémentation des connexions complexes. Nous présentons dans le chapitre suivant notre modèle de connecteur qui permet de remédier à ces différents problèmes.

Nous avons présenté dans le chapitre précédent quelques approches d'architecture et de génie logiciel en nous intéressant à leur considération des interconnexions entre les composants. Dans ce chapitre, nous donnons notre point de vue sur les connexions et comment nous définissons les connecteurs. Nous considérons un connecteur comme une entité à part entière au niveau architecture afin d'assurer la séparation entre les responsabilités fonctionnelles et de communication dans une architecture d'un côté, et d'assurer la séparation entre la description d'une abstraction et son implémentation d'un autre côté. De plus, le connecteur évolue d'une manière spécifique dans le cycle de vie et se transforme pour correspondre à des entités différentes.

Pour commencer, nous discutons les principaux problèmes qui ont motivé la reconsidération des connexions telles qu'elles se présentent actuellement. Nous présenterons quelques éléments de solution à ces problèmes afin d'amener progressivement le concept de connecteur et le besoin d'un modèle de connecteur. Ensuite, nous donnons notre définition des connecteurs et nous présentons succinctement leur évolution dans le cycle de vie. Enfin, nous détaillons le processus de conception et d'utilisation des connecteurs qui décrit les phases et les étapes qu'emprunte un connecteur depuis sa description au niveau architecture jusqu'à son déploiement.

2.1 Problématique et motivation

L'architecture logicielle consiste à décrire le modèle abstrait d'un système, en précisant les interdépendances qui existent entre les « constituants » de ce système. Elle représente, désormais, une étape à part entière dans le processus de développement et s'impose de plus en plus en amont de chaque développement d'application. L'architecture logicielle permet une description grossière de l'application sans rentrer dans les détails de programmation. Elle a pour principales tâches :

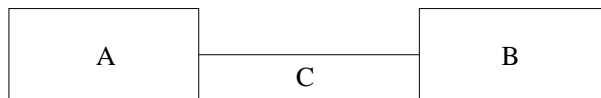
- D'identifier les principaux blocs de construction de l'application, c'est-à-dire les composants et leurs propriétés visibles ;
- De les organiser en les interconnectant afin d'accomplir une fonction ou un ensemble spécifique de fonctions.

Les architectes logiciels représentent les architectures des applications comme un ensemble de composants interagissant. Ainsi, deux entités principales y sont manipulées : les composants et leurs connexions. Les composants représentent les principales unités de calcul ou de

stockage du système ; ils réalisent les propriétés fonctionnelles de l'application. Les connexions, souvent désignées comme connecteurs, représentent des abstractions de communication de haut niveau qui regroupent la communication et la coordination entre les composants. Ces connexions décrivent la sémantique de l'interaction, et participent à la réalisation des propriétés non fonctionnelles de l'application. Elles deviennent ainsi une clé déterminante dans la qualité du logiciel. En effet, c'est sur la connexion que reposent désormais des propriétés importantes dans la qualité du logiciel [58] comme la performance, l'utilisation des ressources, les débits globaux, la capacité d'extension (ou *scalability*), la fiabilité, la sécurité, l'évolution, etc.

Une application est construite par l'assemblage des composants et des connexions, en interconnectant les ports des composants aux extrémités des connecteurs. Et pour simplifier la communication entre architectes et développeurs, une approche informelle mais pratique consiste à décrire les architectures des applications dans un style *boxes-and-lines* [1]. Il s'agit de représenter l'architecture d'une application en utilisant une combinaison de boîtes et de traits représentant respectivement les composants et leurs connexions. Cette combinaison de composants et de connexions (boîtes et traits) constitue l'organisation générale de l'application ou sa configuration.

Pour l'interprétation d'une configuration, il existe quelques conventions de style, définies dans [1], qui dictent la nature des entités qui composent l'architecture de l'application, et qui participent à poser des contraintes simples. Par exemple, si on se positionne dans un contexte où on manipule des filtres et des canaux (*pipe & filter*), alors les composants (les boîtes) représentent les filtres et les connexions (les traits) représentent les canaux, comme illustré dans la figure 2.1 (cas 1). Un canal réalise un transfert unidirectionnel de données. Une connexion bidirectionnelle nécessiterait alors sa représentation avec deux traits. Par ailleurs, si on se positionne dans un contexte client/serveur où les composants clients appellent les services des composants serveurs, alors les composants appelants et appelés sont représentés par des boîtes, et les interactions sont représentées par des traits (figure 2.1 (cas 2)). Par convention dans ce dernier cas, ces interactions sont assimilées à des appels de procédure, c'est-à-dire des interactions simples.



Cas 1 : A = B = Filtre
C = Canal (Pipe)

Cas 2 : A = Appelant	Cas 3 : A = Client
B = Appelé	B = Serveur
C = Appel de procédure	C = RPC

Figure 2.1 — Différents types de connexions dans le modèle *boxes-and-lines*

Ces représentations d'architecture en boîtes et traits ainsi que les conventions qui les accompagnent ne sont pas suffisantes en soi ; on ne peut naturellement pas réduire qu'à cela l'architecture d'une application. En revanche, elles constituent une base de départ et représentent une bonne intuition sur l'organisation globale de l'application avant d'entreprendre les détails des spécifications formelles. Ceci dit, l'intuition de représenter les connexions par de simples traits, malgré leurs responsabilités relatives à la qualité du logiciel, reflète le caractère simpliste et réducteur avec lequel elles sont considérées actuellement.

Le but de cette section est de montrer les problèmes liés à la considération et la représentation actuelle des interconnexions entre composants. À partir de la représentation des interactions avec de simples traits sans sémantique, nous allons montrer le besoin ressenti d'apporter de nouvelles notations, de nouveaux concepts et de nouvelles abstractions de communication¹ en architecture logicielle. Nous présenterons également des éléments de solution pour chacun des besoins exprimés.

2.1.1 Besoin de nouvelles notations et de nouveaux concepts

Le développement des applications avec les approches se basant sur les composants, par opposition aux approches orientées objet, procédurales ou fonctionnelles, apporte des solutions pour améliorer la qualité du logiciel. Ces approches offrent plusieurs avantages qui couvrent tout le cycle de vie de développement d'une application, et plus particulièrement pour la mise en œuvre de l'assemblage et du déploiement. D'abord, elles considèrent une vue globale de l'application en identifiant ses principaux constituants assez tôt dans le cycle de vie. Elles offrent ainsi une représentation préliminaire de l'application pour exprimer les propriétés essentielles de l'application. Ceci permet de garder la trace des composants et de les retrouver facilement pour une meilleure évolution du logiciel. Ensuite, dans ces approches, les composants cachent leurs implémentations et ne communiquent qu'à travers leurs interfaces, appelées *ports*, qui regroupent des services offerts et requis. Ceci permet de disposer ces composants sur étagère comme un ensemble de binaires, avec des interfaces bien définies, afin de les utiliser et les réutiliser sans se préoccuper de comment ils sont implémentés. Enfin, l'approche de développement à base de composants fournit des outils de déploiement qui permettent d'installer les composants et d'établir les connexions entre ces composants.

Notation non appropriée

Le développement d'application à base de composants est différent de l'architecture logicielle. Nous allons faire une correspondance entre les éléments du développement à base de composant, et en particulier les connexions entre les composants, et leurs représentants au niveau architecture. Nous allons passer en revue les différents types de communication : locale, distante et multipoint, afin d'illustrer les insuffisances du trait comme représentation d'une connexion.

Le trait dans une communication locale point-à-point

La représentation au niveau architecture d'une application impliquant un composant appelant et un composant appelé dans un environnement local peut être illustrée comme en figure 2.1 (cas 2). Les boîtes représentent les composants interagissant et le trait la sémantique de la communication. Dans un environnement local, les connexions sont souvent réalisées par un appel de procédure. Les composants sont accédés par déréférencement d'un pointeur et l'interaction est obtenue par un saut vers les instructions correspondantes de la procédure considérée. Ce sont les connexions les plus répandues dans les langages de programmation, elles y sont implicites et primitives. Cette communication est fiable,

¹Nous appelons une abstraction de communication toute communication, interaction ou mode de communication entre des composants considérée indépendamment de son contexte et dont les détails de mise en œuvre ne sont pas pris en compte au niveau architecture. Une abstraction de communication doit durer et rester comme une entité intègre le long du cycle de vie, de sa description à son déploiement pour avoir une implantation propre.

lorsqu'un message est envoyé du côté d'un appelant sa réception est toujours garantie du côté de l'appelé. C'est une communication simple, point-à-point, sa représentation par un trait reste tolérable. Nous désignerons ce type d'interaction comme une connexion simple.

Le trait dans une communication distante point-à-point

En se positionnant dans le contexte d'une application client/serveur (figure 2.1 (cas 3)), l'interaction entre le client et le serveur se fait généralement comme un appel de procédure à distance (ou RPC - *Remote Procedure Call*). Le RPC est habituellement utilisé dans des environnements répartis, en substitution aux techniques statiques basées sur les adresses mémoire utilisées dans les environnements locaux. Il permet à *un* client d'appeler une procédure sur *un* serveur distant, comme s'il s'agissait d'une procédure locale point-à-point. Sa fonction principale consiste à rendre transparentes les opérations de sérialisation, de codage et de décodage des informations, pour leur transmission à travers le réseau d'interconnexion. Il est l'un des moyens de connexion les plus connus et les plus utilisés par les développeurs dans les environnements distribués (CORBA [67], DCE [20], SOAP [37]).

Le RPC permet d'offrir les services du composant serveur au composant client sur le site de ce dernier. Au déploiement, il est concrétisé par l'introduction d'une souche de procédure (*stub*) qui effectue la transformation de l'appel de procédure en un envoi de message depuis le site du client vers le site du serveur. Sur ce dernier, une souche de procédure symétrique (*skeleton*) reçoit le message, effectue la transformation inverse, et réalise l'appel effectif du client par débranchement comme une communication locale. Les paramètres de retour transitent par le chemin inverse via le réseau d'interconnexion. Pour accéder au réseau, le message doit franchir une pile de protocoles de communication, comme le protocole TCP/IP (*Transmission Control Protocol/Internet Protocol*) par exemple. La figure 2.2 schématise cette description du déploiement de l'application client/serveur en utilisant le modèle *boxes-and-lines*, détaillant particulièrement le RPC en explicitant les éléments le constituant².

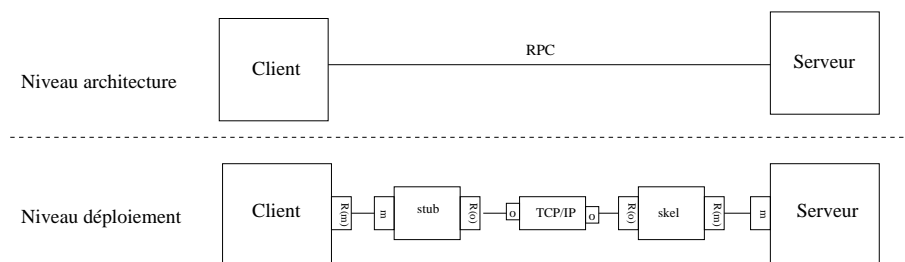


Figure 2.2 — Détails d'une connexion RPC au déploiement

Ainsi, à l'exécution, une requête émise par le client traverse principalement trois entités : les deux souches (*stub* et *skel*) et le protocole de communication. Nous désignons ce type de communication comme une connexion complexe car elle transite par un protocole.

À ce stade du cycle de vie, tous les éléments du RPC, à savoir les deux souches et le protocole TCP/IP, sont des composants binaires accessibles par leurs interfaces. Leur représentation par des boîtes est alors appropriée comme pour les composants client et serveur. Elle est aussi appropriée pour la représentation de leurs communications par des traits puisqu'il s'agit à chaque fois de communications locales simples. Chaque opération

²Cette description ne présente qu'un premier niveau de détails, qui nous permet de ne mettre en évidence que les éléments dont nous avons besoin pour nos explications.

requis d'un composant utilise une opération offerte d'un autre composant :

- L'opération requise $R(m)$ du client utilise l'opération (m) offerte par la souche du client (*stub*) ;
- L'opération $R(o)$ de la souche du client utilise l'opération (o) offerte par la pile de protocole TCP/IP ;
- Le protocole TCP/IP utilise l'opération offerte par la souche du serveur (*skel*) ;
- Enfin, l'opération requise $R(m)$ de la souche du serveur utilise l'opération offerte (m) de ce serveur.

La projection de la description du déploiement de binaires de cette application vers le niveau architecture avec le modèle boîte et trait, comme illustré dans la figure 2.2, indiquerait que deux composants logiciels, un client et un serveur représentés par les boîtes, sont interconnectés par un trait représentant une communication distante, lorsque celle-ci est précisée sur le trait. Cependant, en comparant cette représentation architecturale avec la représentation de l'application au niveau déploiement, dans cette même figure, on retrouve les composants client et serveur mais on remarque que le trait représentant la connexion entre les composants « gagne en épaisseur » lors de sa représentation au niveau déploiement. Ainsi, le RPC est une entité intérieurement complexe. N'ayant que des boîtes et des traits comme outils de représentation dans les architectures, on serait tenté de le représenter par une boîte car, intuitivement, celle-ci laisse présager qu'elle englobe des entités complexes contrairement au trait. Mais cette représentation ne serait pas adéquate car le RPC n'est pas un composant comme nous l'expliquerons un peu plus loin dans cette section. En revanche, le représenter par un trait est réducteur car le trait ne reflète pas et ne donne pas une interprétation intuitive de la complexité intérieure du RPC. Ceci reste aussi vrai pour des communications avec des propriétés plus complexes comme la fiabilité, ou l'asynchronisme. Ainsi, la représentation de cette connexion complexe par un trait n'est pas appropriée.

Le trait dans une communication multipoint

En restant toujours dans le cadre du client/serveur et en multipliant le nombre de clients et de serveurs on obtient une connexion complexe d'une autre nature : une connexion multipoint impliquant plus de deux participants. En effet, pour assurer certaines propriétés complexes dans une architecture client/serveur, on peut être amené à augmenter le nombre de serveurs offrant le même service — changer sa multiplicité. Par exemple, pour assurer la propriété de fiabilité, on peut utiliser un ensemble de serveurs gérés par un algorithme de consensus dont la tâche consiste à élire une seule réponse fiable à transmettre au client.

Bien qu'elle ne soit pas très appropriée, la représentation du RPC au niveau architecture avec un trait reste relativement facile à interpréter lorsque celui-ci est précisé dans la description de l'application. Ceci est dû au fait que c'est une abstraction de communication bien connue qui laisse apparaître la propriété de répartition. La situation n'est pas aussi évidente dans une communication complexe qui fait intervenir plusieurs participants. La figure 2.3 (a) représente l'architecture d'une application impliquant plusieurs clients reliés à un seul serveur avec des RPC (ou appels de procédure). À première vue, cette interconnexion ne suscite aucune interrogation. Il est naturel qu'un composant (le serveur) gère et réponde à plusieurs requêtes à la fois en offrant des services différents, ou puisse fournir un même service

à plusieurs composants différents (les clients) en même temps. Les requêtes sont considérées indépendantes les unes des autres, chacune gérée dans une session différente. Ceci dit, nous pouvons tout de même nous demander si les composants appelants requièrent le service en exclusion mutuelle ou non ? s'ils obtiennent les réponses d'une manière synchronisée ou pas ? ou s'il y a un ordre à respecter pour traiter les requêtes ?

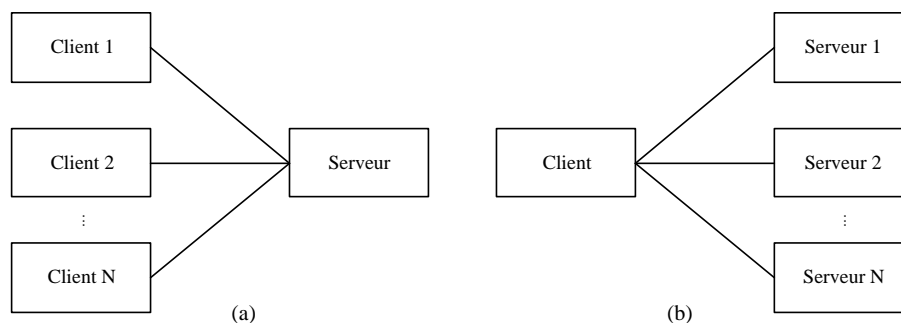


Figure 2.3 — Connexion complexe. (a) plusieurs clients - un serveur, (b) un client - plusieurs serveurs

La figure 2.3 (b) montre une autre communication complexe multipoint avec cette fois-ci un seul client et plusieurs serveurs. Une interprétation intuitive de cette architecture serait qu'un composant client requiert plusieurs services de différents composants serveurs. Mais est-ce la vraie intention ? Il est possible que le composant appelant utilise les autres composants pour équilibrer la charge, ou bien les utilise pour assurer la fiabilité à l'aide d'une technique de redondance ou de consensus. Cette représentation ne donne aucune indication sur la façon de faire interagir les composants. Même s'il est précisé que les communications sont distantes, il n'y a aucune information sur la coordination entre les composants, comme par exemple l'ordre d'utilisation des services.

Avec cette notation, la sémantique de la communication et la coordination entre les composants ne peut pas être exprimée car elle ne peut pas être contenue sur l'ensemble des traits ou sur leurs extrémités communes. Cette description graphique montre l'ambiguïté qui sera engendrée lors des spécifications et qui aura une conséquence directe sur le développement de l'application. En effet, les spécifications formelles reflèteront ces représentations puisqu'il n'existe pas d'entité à part qui regroupe la sémantique de la communication multipoint. Ceci se répercute sur le développement des applications :

- La sémantique de la communication et de la coordination sera mélangée au code des composants interagissant ;
- Ceci limitera la séparation des responsabilités ainsi que la réutilisation à la fois des composants et de la sémantique de communication.

Comme représentée dans la figure 2.3, cette description manque de précision et laisse plusieurs interprétations ouvertes. Elle est, de ce fait, ambiguë. Donc cette représentation des interactions de la sémantique de communication par un ensemble de traits n'est pas appropriée au cas des communications complexes. Nous proposons de l'enrichir.

Concepts non appropriés

Après avoir vu dans la section précédente les contraintes de notation, nous allons voir dans cette section les contraintes sémantiques qui existent entre une application réalisée à base de composants et son architecture correspondante, c'est-à-dire la correspondance entre

les éléments réels utilisés lors de la réalisation et leurs correspondants architecturaux. Nous avons vu que les notations architecturales actuelles ne sont pas appropriées, qu'en est-il des concepts ?

Revenons au résultat du déploiement de l'application client/serveur illustré en figure 2.2 (page 24). Cette figure représente l'assemblage final des composants client et serveur concrétisé par l'apparition de nouveaux composants intermédiaires pour la réalisation de la communication effective. À ce stade du processus de développement tous ces composants sont des binaires. À partir de là, nous proposons d'employer une méthode d'ingénierie inverse pour revenir en arrière dans le processus de développement et retrouver les éléments architecturaux qui correspondent à ces composants binaires. Pour cela, nous allons étudier une série de découpages en mettant des frontières entre les composants de cet assemblage, comme dans la figure 2.4.

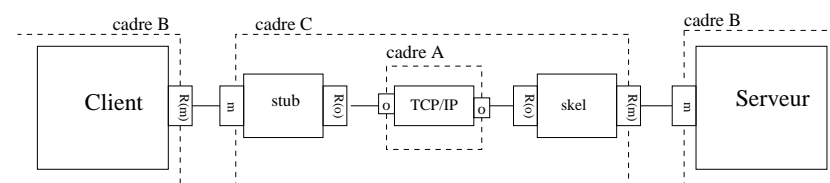


Figure 2.4 — Détails d'une connexion RPC au déploiement avec découpage

Le premier découpage en pointillé A permet d'isoler le protocole TCP/IP comme une entité à part. Ce protocole possède trois caractéristiques particulières :

- Il peut être considéré au niveau architecture comme une boîte noire qui possède ses *propres interfaces*. En effet, en prenant exemple sur la pile de protocole qui compose les couches du modèle OSI (*Open Systems Inteconnection*), chaque protocole d'une couche donnée offre et utilise les interfaces de la couche du dessus ainsi que les interfaces de la couche du dessous ;
- Il est déployable ;
- Il est déposé sur étagère pour être utilisé par des tiers.

Avec ces caractéristiques, on peut considérer que le représentant du protocole TCP/IP existe au niveau architecture. Il s'agit d'un composant logiciel qui *explícite ces interfaces* et cache son implémentation. Sa définition correspond parfaitement à la définition des composants de C. Szyperski [87] qui le décrit comme : « Une entité de composition avec des interfaces bien spécifiées. Un composant logiciel peut être déployé indépendamment et peut être sujet à composition par des parties tierces³ ». Par ailleurs, le fait que TCP/IP soit déployable sur plusieurs sites différents fait de lui un composant particulier. Conformément aux travaux de thèse de E. Cariou [17], il est considéré comme un composant dédié à la communication appelé « médium ». Nous reviendrons plus en détails sur ces travaux dans la section 4.1.

Dans le second découpage, l'encadrement en pointillé B dans la figure 2.4, on retrouve les composants fonctionnels, le client et le serveur. Ces composants possèdent des interfaces bien

³«A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.»

définies qui ont une *existence propre* au niveau architecture. Ils sont indépendants, destinés à être composés et déployables. Ils correspondent aussi parfaitement à la définition de C. Szyperski. Leurs représentants existent aussi au niveau architecture, ce sont les composants logiciels « conventionnels ».

Le dernier découpage regroupe le protocole TCP/IP avec les composants souches du côté client et serveur pour n'en former qu'une seule entité (encadrement C dans la figure 2.4). Ces composants forment le composant qui correspond à la concrétisation du RPC au niveau déploiement. À ce niveau, ce composant explicite comme interfaces celles des composants souches du côté client et serveur. Ces interfaces sont concrètes⁴ et possèdent une existence propre au niveau déploiement puisqu'elles représentent les premières interfaces avec lesquelles interagissent le client et le serveur pour émettre et recevoir les requêtes et les réponses. En revenant en amont dans le cycle de vie, si on recherche ces interfaces au niveau architecture, il se trouve qu'elles correspondent exactement aux interfaces des composants interagissant, c'est-à-dire le client et le serveur. En effet, les interfaces actuelles du composant RPC (qui sont celle des souches) sont *identiques à celle des composants*, ce qui sert à simuler un appel local du service. Le rôle du composant RPC à l'exécution consiste ensuite en l'acheminement des requêtes et des réponses à travers le réseau d'interconnexion. Au déploiement, les interfaces du composant RPC sont *obtenues* par un processus de génération ou de compilation, *elles ne représentent pas ses propres interfaces au niveau architecture*.

Ainsi, l'abstraction RPC existe au niveau architecture et elle est destinée à être utilisée par d'autres composants, ce qui satisfait une partie de la définition de C. Szyperski. En revanche, elle n'est pas indépendante, puisqu'elle ne définit pas d'interfaces au niveau architecture et a besoin des interfaces des composants interagissant pour obtenir les siennes au niveau déploiement. De ce fait, elle n'est pas déployable en tant que tel (elle ne correspond pas à un élément binaire déployable), elle doit être assemblée pour être déployée. Donc, cette entité ne correspond pas à la définition d'un composant, *son représentant au niveau architecture est différent du composant logiciel standard*. Lorsqu'elle est attachée, cette entité représente la connexion entre les composants. Cependant, à quoi correspond t-elle lorsqu'elle est dépourvue de toute utilisation ou lorsqu'elle est considérée en dehors de son contexte d'utilisation, isolée sur étagère ? L'abstraction RPC existe, nous savons que c'est une propriété qui sert à réaliser les communications distantes, mais quel statut possède-t-elle au niveau architecture si elle ne correspond ni à un composant ni à une connexion ?

Ainsi, les interactions entre les composants ou leurs connexions n'ont pas de statut stable en architecture logicielle. Il existe des éléments de connexion réifiés dans des composants avec des interfaces bien définies qui peuvent être représentés en architecture logicielle par des composants, mais il existe des éléments de connexion, tel que le RPC, qui ne possèdent pas d'interfaces définies et ne peuvent pas être représentés en architecture logicielle.

Éléments de solution

Les connexions entre les composants logiciels sont potentiellement complexes. Leur représentation avec de simples traits au niveau architecture n'est pas appropriée. Cette représentation n'est pas adaptée pour exprimer cette complexité et engendre des ambiguïtés d'interprétation. La description d'une architecture a pour but de représenter la structure générale des applications avec un niveau faible de détails. Les composants ont une représentation bien adaptée avec des boîtes qui laisse présager leur complexité en ne

⁴Par concrètes nous désignons le fait qu'elles sont implémentées.

décrivant que leurs propriétés principales, tout en évitant de se préoccuper des détails de l'implémentation. Par contre, la représentation actuelle des connexions par des traits sans sémantique réduit la possibilité de préciser leurs propriétés et ainsi les propriétés globales de l'application.

Des spécifications formelles peuvent étendre ces descriptions d'architecture afin de permettre leur compréhension⁵. Cependant, les descriptions d'architecture graphiques sont sensées déjà donner une interprétation rapide et intuitive de la structure et des propriétés générales de l'application. Telles quelles sont représentées actuellement, les architectures des applications manquent de précision concernant la sémantique de la communication ce qui a une conséquence directe sur le processus de développement. Comme nous le verrons plus en détails dans la section suivante, seules les interactions simples sont réellement implémentées pour subsister en tant qu'entité entière tout le long du processus de développement.

En plus d'une expression de sémantique de communication limitée, cette abstraction de la communication dans un simple trait manque de détails importants par rapport aux interfaces des connexions. Autant les interfaces des composants sont expressives, autant il reste une ambiguïté sur la nature des interfaces des connexions. Nous avons vu que nous pouvions assimiler certains éléments de la connexion à des composants logiciels un peu particuliers. Ces composants possèdent des interfaces bien définies et explicites qui existent depuis la description jusqu'au déploiement. En revanche, il existe d'autres éléments de la connexion qui n'ont pas d'existence au niveau architecture car ils ne possèdent pas d'interfaces propres à eux qui soient définies. Ces éléments existent par la sémantique de communication ou la propriété qu'ils assurent, mais leurs interfaces ne prennent forme que plus tard dans le cycle de vie, après l'assemblage. Il s'agit ici d'entités qui possèdent des interfaces *implicites*.

Pour remédier à ces problèmes, nous proposons d'introduire de nouvelles notations et d'enrichir l'architecture logicielle avec des concepts mieux définis. Nous proposons de passer du modèle *boxes-and-lines* au modèle *boxes, lines and ellipses* pour mieux expliciter les interactions. Les boîtes et les traits ne sont pas suffisants. Les boîtes dictent un peu la complexité intérieure des composants. En revanche les traits informent seulement qu'il existe une communication entre les boîtes mais ne permettent pas de préciser quelle type de communication ou de coordination existe entre ces boîtes. Le trait est ainsi simpliste et réducteur. Introduire l'ellipse pour exprimer la sémantique de communication entre les composants permet de donner plus de précision sur la coordination et laisse apparaître la complexité lors de son implémentation. En effet, le fait de prévoir explicitement une entité complexe depuis l'architecture permet de maintenir cette complexité pendant tout le cycle de vie. Les ellipses sont plus explicites pour déterminer la sémantique de la communication que les traits. Elles permettent d'explicitement la complexité que peut exprimer une connexion et par conséquent celle-ci est mieux ressentie et prise en charge au niveau de l'implémentation.

Par exemple, une communication distante entre un client et un serveur sera présentée comme une ellipse portant la mention ou la propriété RPC comme illustré dans la figure 2.5 (a). Si la sémantique de la communication entre le client et les serveurs dans l'exemple de la communication multiple de la figure 2.3 est l'équilibrage de charge, on la représenterait dans une ellipse explicitant cette sémantique comme dans la figure 2.5 (b). Ainsi, toute l'ambiguïté est levée.

Cette ellipse désigne ainsi l'entité que nous avons identifiée en architecture pour expliciter les propriétés générales de la communication et de l'application. Cette entité possède des interfaces implicites et nous la désignons comme étant le *connecteur*. Le connecteur n'est pas

⁵Comme c'est le cas dans Wright [4].

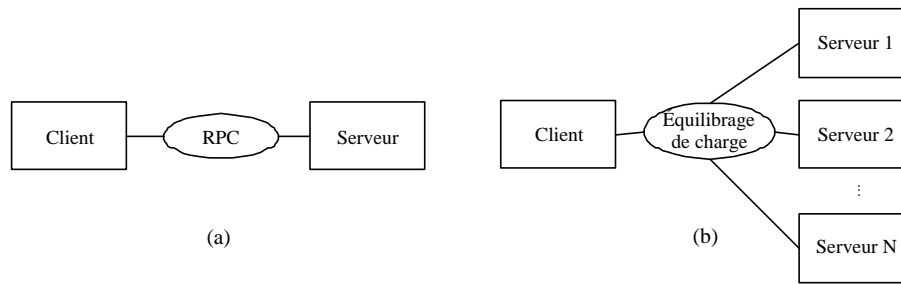


Figure 2.5 — Nouvelle forme d'un connecteur : une ellipse

un nouveau concept, il est apparu avec les premières définitions de l'architecture logicielle et ce terme existe dans plusieurs langages de description d'architecture logicielle [56] (cf. section 1.3.2). Cependant, son utilisation est confuse, il est utilisé pour désigner plusieurs entités de plusieurs niveaux que nous considérons comme étant très différentes. Il peut être utilisé pour désigner des interactions et des connexions très abstraites, ou pour désigner des interactions au niveau du déploiement, qui représentent des moyens de communication très concrets. Le premier objectif de cette thèse est d'éclaircir ce concept et l'usage de ce mot.

La différence mise en avant entre un composant et un connecteur dans les approches d'architecture logicielle est orientée fonctionnalité ; c'est-à-dire que les composants s'occupent des propriétés fonctionnelles des applications, et les connecteurs s'occupent de la communication et participent à la réalisation des propriétés non fonctionnelles. Cette différence est illustrée par le choix de deux formes graphiques pour représenter les composants et les connecteurs : les boîtes et les traits. La différence des entités que nous défendons dans cette thèse rajoute une différence de nature des entités en plus de cette différence de fonctionnalité. Cette différence est marquée par le fait que les composants possèdent des interfaces explicites au niveau architecture, alors que les connecteurs possèdent des interfaces implicites. Ceci implique une différence de fabrication et d'utilisation que nous allons expliquer tout au long de notre travail.

Ainsi, dans le cadre de cette thèse, nous nous basons sur les affirmations suivantes :

- Les entités qui existent au niveau architecture ne correspondent pas toutes à des composants avec des interfaces explicites ;
- Les connexions entre composants représentées par des traits sont complexes. Cette notation par des traits n'est pas appropriée pour exprimer les communications complexes et multipoint mais nous la gardons pour la représentation de communications simples et locales ;
- Nous précisons le concept de connecteur : un connecteur réifie une abstraction de communication dans une entité à part entière au niveau architecture. Il possède des interfaces *implicites* et il est représenté par une ellipse ;
- Nous faisons la différence entre un connecteur et une connexion. Un connecteur existe isolé sur une étagère, il est destiné à être assemblé. Une connexion existe une fois que les composants interagissant et le connecteur sont sélectionnés et assemblés formant ainsi la connexion.

2.1.2 Besoin de nouvelles abstractions de communication et de leurs implémentations

L'architecture client/serveur est un des modèles d'architecture les plus communs pour la réalisation des applications. Elle permet la répartition de ces applications sur différents sites afin de partager des données et des traitements. Les modèles de composants s'appuient sur cette architecture puisque les composants offrent et requièrent des services. Un composant peut être client et serveur à la fois. Deux modes de communication dominent dans ce style d'architecture : la communication synchrone et asynchrone. La communication synchrone bloque le client jusqu'à la réception d'une réponse du serveur. En revanche, la communication asynchrone permet au client d'envoyer un message et de poursuivre son travail sans interruption ; il peut recevoir une réponse du serveur quand celui-ci aura terminé de traiter son message. Nous nous servons de ces deux abstractions pour les explications qui suivent.

Abstractions de communication implémentées non suffisantes

Pour réaliser des applications, les développeurs ont souvent recours à ces moyens de communication synchrone et asynchrone de base pour assurer des propriétés de communication plus complexes, comme par exemple la propriété d'équilibrage de charge⁶. Supposons que cette dernière soit la sémantique de communication de la figure 2.3 (b) page 26. Dans cette application, un client a besoin d'un service offert par le serveur. Comme les clients émettent un nombre important de requêtes, le serveur a été dupliqué afin de répartir les requêtes équitablement sur tous les serveurs et éviter la surcharge d'un seul serveur et le ralentissement de tout le système.

L'équilibrage de charge peut être réalisé sur plusieurs niveaux d'abstraction suivant des politiques différentes. Nous nous basons dans cet exemple sur l'équilibrage de charge niveau application afin de pouvoir prendre en compte des métriques de charge liées à l'application. À notre connaissance, il n'existe aucune spécification de l'équilibrage de charge. Il existe des spécifications qui indiquent comment réaliser l'équilibrage de charge (des algorithmes que les auteurs se recommandent), mais pas de spécification générale de l'intention. L'inexistence d'une telle spécification rend son intégration dans des applications non formalisées. Ceci justifie donc l'inexistence d'une méthode standard pour la réalisation de cette abstraction et justifie aussi son inexistence en tant qu'entité implémentée. Analysons à présent comment cette propriété d'équilibrage de charge est susceptible d'être réalisée avec les moyens existants.

Dans notre exemple, les clients et les serveurs étant distants, une manière intuitive de les relier serait d'utiliser les abstractions de communication les plus courantes, la communication synchrone tel que le RPC (comme illustré dans la figure 2.6 (a)). Lors de la réalisation, une implémentation particulière de RPC peut être utilisée, RMI (*Remote Method Invocation*) par exemple. Dans ce cas, le processus de compilation des programmes mène à la génération des souches de procédure du côté des serveurs : les *stubs* qui représentent en fait les références distantes des serveurs, et les *skeletons*. Pour répartir les requêtes sur l'ensemble des serveurs afin d'équilibrer sa charge, le client doit connaître les noms de tous les serveurs participant à l'interaction. Ces noms doivent être inscrits dans le code du client et il doit lui-même décider du serveur qui exécutera chacune des requêtes en appliquant une politique d'équilibrage de charge. À l'exécution, le client doit télécharger l'ensemble des *stubs* qui sont du côté des serveurs avec une opération *lookup()*. Une fois installé localement du côté du client, le client

⁶Cette application avec une sémantique d'équilibrage de charge nous servira d'application exemple pour illustrer et argumenter nos contributions tout au long de notre travail.

communiquent directement avec les *stubs* qui se chargent d'envoyer la requête à travers le réseau vers le serveur choisi.

Ainsi conçu, le client possède la responsabilité de réaliser une propriété qui sort du cadre de sa fonctionnalité, c'est-à-dire l'amélioration du temps de réponse qui est réalisé à l'aide des mécanismes d'équilibrage de charge. Dans cette application, le RPC sert seulement à rendre transparents les détails de la distribution. Les détails de l'équilibrage de charge sont pris en charge par les composants de l'application. Cette réalisation encombre les clients et les serveurs avec des détails indépendants de leur fonctionnalité, ce qui rend impossible la séparation des responsabilités. Cette approche maintient de la glu superflue dans les codes des serveurs et du client comme le montre la figure 2.6 (b).

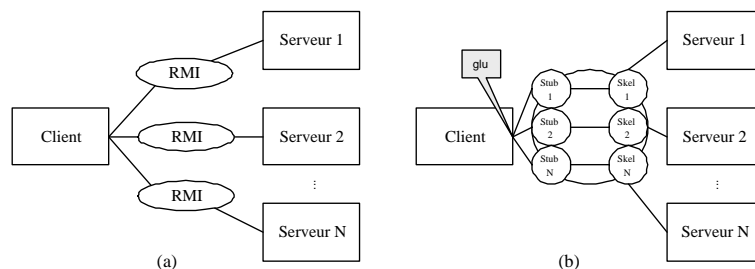


Figure 2.6 — Connexion complexe avec un connecteur RMI

L'abstraction « équilibrage de charge » n'existe pas au niveau architecture en tant qu'entité à part entière vu l'absence de spécifications. Elle l'est encore moins tout le long du cycle de vie. Ceci a pour conséquence que :

- Cette propriété n'est prise en charge que tardivement dans le cycle de vie de l'application ;
- Elle est conçue et réalisée avec des moyens de communication élémentaires (ceux qui sont implémentés) ;
- Elle est réalisée en même temps que les composants logiciels et devient une partie intégrante des composants de l'application ;
- Ni les composants logiciels, ni l'implémentation de l'équilibrage de charge ne sont réutilisables. Les composants ne sont plus réutilisables sans cette propriété d'équilibrage de charge, ce qui va à l'encontre des principes mêmes de la programmation à base de composants.

Ces points affectent directement la qualité et l'efficacité du développement logiciel. Les abstractions de communication sont trop dépendantes des applications ce qui les rend non réutilisables et rend difficile la maintenance et l'évolution des applications. Le fait de se limiter aux abstractions de communication et aux connecteurs existants dans les plates-formes courantes limite l'expressivité de la description de l'architecture. En effet, par habitude seuls les structures et blocs de construction supportés par un langage ou un système d'implémentation cible sont utilisés dans la pratique. La réalisation des abstractions de communication est donc à l'image de leur représentation. Le fait de les représenter actuellement par des traits indique qu'on n'utilise que des connecteurs simples lors de la réalisation de l'application. L'absence

de notation, que nous avons vu précédemment reflète l'absence d'abstraction. L'inexistence d'une spécification de l'équilibrage de charge implique l'inexistence d'un modèle de réalisation de cette abstraction. Ainsi, chaque développeur la réalise à sa façon, les efforts ne sont pas exploités et le savoir-faire n'est pas capitalisé.

Les abstractions de communication peuvent être décrites avec des langages de description d'architecture qui reconnaissent les connecteurs ou avec les langages de coordination (des langages formels). Cependant, comme ces langages ne fournissent pas un moyen d'implémentation pour garder cette abstraction, cette dernière est réalisée avec les moyens élémentaires. Ainsi, l'abstraction de communication se noie dans les composants lors de la réalisation. De plus, elle se perd pendant le cycle de vie puisque son implémentation revient à utiliser les moyens simples implémentés. Ainsi, le cycle de vie des abstractions de communication se trouve fortement lié à celui des composants. Il y a donc un *besoin de décrire des abstractions de communication nouvelles* pour plus d'indépendance par rapport aux composants et de les *disposer comme des entités implémentées à part entière*.

Implémentation des abstractions de communication sur plusieurs plates-formes

Les abstractions de communication client/serveur synchrones et asynchrones sont les plus faciles à utiliser car elles sont maîtrisées et disponibles à l'implémentation. Ce fort besoin de leur utilisation et le degré de maîtrise atteint par ces abstractions a conduit à étendre leur présence dans la plupart des modèles et plates-formes distribuées. En effet, on retrouve des implémentations très diverses sur les différentes plates-formes du marché, comme CORBA de l'OMG et JAVA de Sun. La figure 2.7 recense quelques réalisations possibles de ces deux propriétés dominantes qui existent sur des protocoles différents.

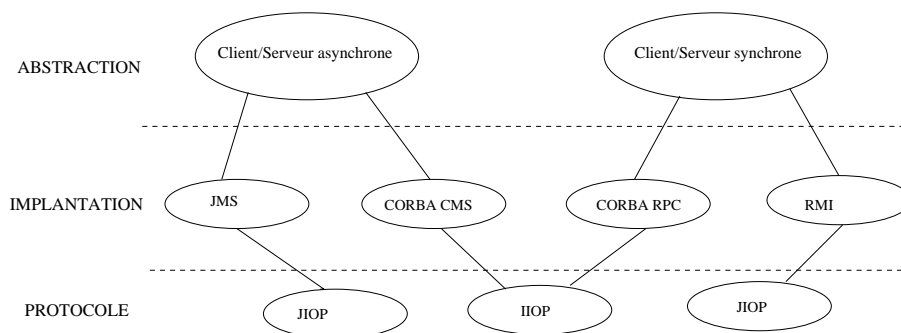


Figure 2.7 — Les propriétés synchrone et asynchrone et quelques réalisations

Comme nous l'avons évoqué précédemment, l'abstraction RPC est connue pour assurer la communication synchrone à distance. Comme l'indique cette figure, il existe des réalisations de RPC sur différentes plates-formes dont CORBA de l'OMG qui repose sur le protocole IIOP (*Internet Inter Orb Protocol*) et RMI de Sun qui repose sur le protocole JIOP (*Java Inter Orb Protocol*). Il existe aussi des réalisations de la communication asynchrone :

- Chez l'OMG, CORBA définit le service d'événement ou de message (CMS ou *CORBA Message Service*);
- Chez Sun, le service de message de Java (JMS ou *Java Message Service*) est apparu avec les EJB.

Ces services de réalisation de la propriété de communication asynchrone n'ont pas toujours été des entités à part entière, ils sont arrivés tardivement, bien après le RPC. En effet, certaines solutions se servaient de la communication synchrone pour réaliser la communication asynchrone. Par exemple, dans CORBA on utilisait le RPC en précisant le mot clé « *oneway* », et dans RMI on utilisait la communication synchrone combinée avec des threads [31]. C'est en affirmant son utilité que cette abstraction s'est imposée comme une entité à part entière.

De la même manière, disposant de cette palette de moyens de communication simples, implémentés et disponibles sur plusieurs plates-formes, les développeurs ont le choix pour réaliser des abstractions de communication complexes entre utiliser ces moyens simples, ou développer d'autres moyens sur mesure. Cette deuxième méthode consiste à développer les abstractions suivant les spécifications mais sur mesure comme des composants et de manière ad hoc sur les plates-formes. Le plus souvent les développeurs optent pour la première solution comme nous l'avons vu dans la section précédente. Bien que la deuxième méthode offre une bonne séparation des responsabilités, elle n'est pas très attrayante car elle reste difficile à réaliser et peu réutilisable. En effet, la réalisation d'abstractions de communication complexe nécessite une bonne maîtrise de la plate-forme, et les interfaces explicites des composants les obligent à utiliser des adaptateurs pour pouvoir les réutiliser. Cette solution rend les composants très dépendants et attachés à ces réalisations ad hoc.

Ainsi, dans la pratique les développeurs se réfèrent principalement à ces deux modes de communication simples car leurs implémentations existent. Ils utilisent une combinaison de ces abstractions, même si elle n'est pas toujours adéquate, et ceci est malheureusement visible au niveau de l'application. Ils ramènent la réalisation des abstractions complexes à une combinaison de ces moyens simples même si ces derniers ne sont pas destinés à cela. De la même manière que la propriété asynchrone était réalisée à l'aide de la propriété synchrone, d'autres propriétés complexes, comme l'équilibrage de charge, sont actuellement réalisées à l'aide de ces propriétés. Ceci est dû au fait que ces propriétés se sont affirmées et pour lesquelles on dispose d'un large choix d'implémentation sur des plates-formes diverses. Par exemple, les développeurs utilisateurs des modèles CCM et EJB avec leurs plates-formes sous-jacentes n'ont pas d'autres choix que d'utiliser ces deux sémantiques pour réaliser des sémantiques de communication complexes. En se reposant respectivement sur les intergiciels CORBA et RMI, ils n'offrent que les sémantiques disponibles sur ces plates-formes.

L'idée est de réaliser une abstraction de communication sur différentes plates-formes est très attractive. Elle est très utile puisqu'elle permet d'étendre des abstractions utiles sur les plates-formes disponibles pour les rendre à la portée de tous et ne pas pénaliser des utilisateurs d'une plate-forme particulière par rapport à une autre. Cependant, nous considérons que les abstractions actuellement implémentées ne sont pas suffisantes. En effet, pourquoi restreindre cette variété des plates-formes uniquement aux abstractions simples ? On continue à utiliser ces moyens de communication élémentaires de base sans penser à étendre la plate-forme avec de nouvelles abstractions. Donc l'idée est de penser à de nouvelles sémantiques de communication complexes et de prévoir leur intégration dans plusieurs plates-formes.

Éléments de solution

Les abstractions de communication simples sont très dominantes dans les développements logiciels actuels mais elles ne sont plus suffisantes. Leur utilisation limite la qualité du logiciel en terme de réutilisation, d'évolution et de maintenance. Malgré cela, les développeurs s'y

réfèrent automatiquement dans leurs applications sans développer de nouvelles abstractions de communication. Ceci est compréhensible car l'implémentation des abstractions simples existe, par contre il n'existe pas de méthodologie ou des outils pour réaliser des abstractions de communication complexes, génériques, indépendantes et donc réutilisables. Même si elles peuvent être spécifiées, ces abstractions de communication complexes ne sont pas implémentées, elles sont perdues ou noyées dans la suite du cycle de vie. Il y a donc un fort besoin de spécifier de nouvelles abstractions de communication fortes et complexes mais de ne pas se contenter que de ces spécifications. Il est souhaitable de définir une entité dédiée à la communication et de pouvoir la garder jusqu'à l'implémentation. Cette solution évitera de réaliser une abstraction de communication complexe avec une combinaison très compliquée d'abstractions simples disponibles à l'implémentation, et dont la mise en œuvre sera mélangée avec les composants de l'application.

Nous voudrions parvenir à créer des abstractions de communication complexes, comme l'équilibrage de charge par exemple, au niveau architecture et de les préserver jusqu'à l'implémentation. Nous proposons d'inciter à la spécification de nouvelles abstractions de communication comme des connecteurs. Ceci n'est pas une idée nouvelle, il existe beaucoup de travaux qui défendent la spécification de nouvelles abstractions. Cependant, ce que nous apportons ici c'est une solution pour réaliser des implémentations pour ces spécifications. Nous proposons le mécanisme de génération de code comme notre solution pour l'implémentation des connecteurs, car nous considérons qu'il représente un bon mécanisme de réutilisation. Ce mécanisme de génération (inspiré par le mécanisme de génération des connecteurs simples comme RMI) permet de fournir une alternative à la solution des abstractions de communication sur mesure. Ceci est possible car les interfaces implicites du connecteur ne deviennent sur mesure que tardivement et il n'y plus besoin de fournir des adaptateurs. Ainsi, ces abstractions de communication seront plus facilement réutilisables puisque leurs implémentations, sous forme de générateurs, existent.

En utilisant la notation proposée, la figure 2.8 (a) montre l'utilisation au niveau architecture d'un connecteur d'équilibrage de charge qui indique que cette propriété est appliquée aux composants interagissants. La figure 2.8 (b) montre son évolution dans la suite du cycle de vie, après l'utilisation du mécanisme de génération. En comparaison avec la description de la réalisation avec des connecteurs simples (figure 2.6 (b) page 32), on remarque que maintenant la glu superflue liée à la communication n'est plus imbriquée dans la réalisation des composants mais incluse dans la réalisation de l'abstraction de communication. Les interfaces des connecteurs sont adaptables, elles ne sont pas spécifiées « en dur » comme les abstractions faites sur mesure. Elles le deviennent après le mécanisme de génération. Ce connecteur, dont nous verrons la mise en œuvre plus en détails par la suite, réalise ainsi la propriété d'amélioration du temps de traitement des requêtes du client en utilisant les mécanismes d'équilibrage de charge entre les serveurs. Il décharge ainsi les composants de détails supplémentaires, non liés à leur fonctionnalité.

Comme déjà réalisé pour les propriétés de communication synchrone et asynchrone, nous proposons que chacun des connecteurs identifiés soit implémenté sur différentes plates-formes pour généraliser leur utilisation et ne pas être restreint à une seule plate-forme. Au niveau architecture, on se concentre sur la description du connecteur sans se soucier de ce qui existe ou n'existe pas sur les plates-formes. Nous encourageons l'implémentation de ces connecteurs sur différentes plates-formes existantes comme des générateurs afin de pouvoir préserver l'abstraction de communication comme une entité à part entière et indépendante pendant tout le cycle de vie. Nous généralisons ainsi l'approche d'avoir plusieurs abstractions sur plusieurs plates-formes différentes.

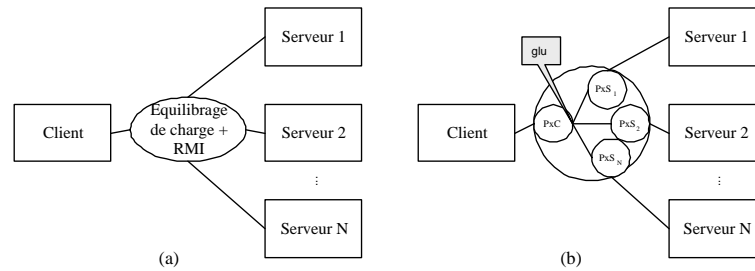


Figure 2.8 — Connexion complexe avec un connecteur d'équilibrage de charge

Pour cela, nous avons besoin de spécifier un cycle de vie des connecteurs qui soit indépendant de celui des composants. Un cycle de vie qui comporte la spécification, l'assemblage, l'implémentation et le déploiement des connecteurs. Ces derniers prennent ainsi des formes différentes et nous avons besoin de préciser à la fois la nature et le vocabulaire pour désigner les transformations des connecteurs dans ce cycle de vie.

2.1.3 Synthèse

Nous avons mis en avant dans cette section les problèmes liés au niveau de la compréhension actuelle des connecteurs, connexions et abstractions de communication par la communauté du développement logiciel. Nous avons commencé par les problèmes de notation et de représentation des architectures. La représentation actuelle en boîtes et traits restreint l'expressivité des architectures logicielles. Non seulement elle n'exprime pas la complexité réelle des connecteurs mais elle ne convient pas pour exprimer les architectures complexes, comme les architecture contenant des interactions multipoints par exemple. Elle n'offre pas d'interprétations intuitives et laisse planer des doutes et des ambiguïtés sur la sémantique de la communication.

Il ne s'agit pas que d'une question de notation, cette représentation se répercute sur la suite du processus de développement des applications. En effet, ces représentations sont à l'image de ce qui est réellement utilisé pour les réalisations. Le fait d'utiliser des connecteurs simples en architecture implique l'utilisation de connecteurs simples à la réalisation des applications. En effet, pour implémenter leurs applications, les développeurs utilisent principalement des combinaisons de connexions simples car leurs implémentations existent, ils reviennent toujours à utiliser les « bons vieux connecteurs ». En outre, certains éléments de connexion clés implémentés, comme le RPC par exemple, ne possèdent pas de représentant au niveau architecture. Il existe un réel écart entre les éléments d'architecture et les éléments d'implémentation. D'un côté il existe des langages et des méthodes pour la spécification d'interactions complexes qui ne fournissent pas des outils pour leur implémentation, d'un autre côté certains éléments de connexion clés implémentés, qui ne sont pas des composants, n'ont pas de représentant au niveau architecture !

Pour remédier à ces différents problèmes, nous avons proposé de changer de notation, de définir de nouveaux concepts et de fournir une solution d'implémentation. Nous avons proposé de représenter les interactions complexes par des ellipses qui permettent d'exprimer explicitement la sémantique de communication. Cette nouvelle notation instaure un nouveau concept : *le connecteur*. Celui-ci permet de contenir la sémantique de communication complexe et possède des interfaces implicites et génériques. Nous introduisons ainsi une différence de nature entre les composants conventionnels et les connecteurs, en plus de la différence de

fonctionnalité. Les composants fonctionnels explicitent leurs interfaces alors que les connecteurs les cachent. Nous définissons une méthodologie d'implémentation de ces connecteurs à l'aide du mécanisme de génération afin de pouvoir préserver son abstraction. Enfin, nous proposons de généraliser cette approche de génération de code pour de nouvelles abstractions de communication sur différentes plates-formes. Nous utiliserons ce mécanisme de génération pour réaliser notre connecteur complexe d'équilibrage de charge.

Nous décrivons dans la suite de cette thèse le moyen de réaliser et de mettre en œuvre ces propositions. La section suivante donne plus d'éléments de définitions et d'explications sur le concept de connecteur.

2.2 Définition et cycle de vie des connecteurs

Après avoir recensé les principaux problèmes liés à la définition et à la considération actuelle des moyens de communication en architecture logicielle, nous détaillons dans cette section notre proposition de connecteur en décrivant sa définition, ses caractéristiques et son cycle de vie. Mais avant cela, nous commençons par passer en revue quelques définitions standards des connecteurs qui existent dans la littérature. Nous discuterons leurs limitations et identifierons les éléments que nous augmentons pour une définition plus complète et rigoureuse des connecteurs.

2.2.1 Travaux connexes

Un travail de référence sur les connecteurs a été réalisé par Mehta *et al.* dans [58]. Dans cet article, les auteurs proposent un framework pour une classification des connecteurs visant une meilleure compréhension et prise en main des moyens de connexion. Ils se basent sur la définition classique et générique des connecteurs donnée par [83] :

« Les connecteurs servent d'intermédiaire pour les interactions entre les composants ; c'est-à-dire, ils établissent les règles qui régissent les interactions des composants et spécifient tout mécanisme auxiliaire requis⁷. »

Une contribution majeure de Mehta *et al.* [58] est l'identification de trois blocs fondamentaux de construction sur lesquels repose la réalisation d'un connecteur ou d'une interaction logicielle. Le premier bloc de construction est le conduit (tuyau), il s'agit d'*un* ou *plusieurs* canaux d'interaction sans aucun comportement associé. Le second bloc de construction regroupe les mécanismes de transfert de données. Le troisième bloc de construction regroupe les mécanismes de transfert de contrôle. Ainsi, tous les connecteurs, quelle que soit leur complexité, fournissent des mécanismes pour transférer du contrôle et des données le long des canaux. Sur cette base, nous considérons que si un connecteur comporte un seul canal, l'interaction est considérée comme une communication point-à-point. Si le connecteur comporte plusieurs canaux, l'interaction est considérée comme une communication multipoint. En suivant cette définition, une interaction reposant sur un seul canal aurait son représentant au niveau architecture (dans le modèle *boxes-and-lines*), c'est une communication point-à-point représentée par un trait. De même, une interaction reposant sur plusieurs canaux serait représentée comme un ensemble de communication point-à-point comme en figure 2.3 (page 26), c'est-à-dire comme un ensemble de traits sans sémantique explicite. Avec cette

⁷“Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.”

définition, nous retrouvons les problèmes des connexions multipoint que nous avons évoqué dans la section 2.1.1. Ne possédant pas de représentant explicite, la sémantique exprimée avec des connexions multipoint engendre une ambiguïté d'interprétation. C'est pour cette raison que nous considérons le connecteur comme l'ellipse qui englobe l'ensemble des interactions entre les composants en définissant une sémantique commune et qui cache les détails des connexions point-à-point qui peuvent exister à l'intérieur.

Une autre contribution de Mehta *et al.* [58] est la définition d'une classification étendue recensant les différents mécanismes d'interaction employés dans les systèmes logiciels. Ceci a pour but d'améliorer leur utilisation en apportant un support récapitulatif qui permettrait aux développeurs de connecteurs de concevoir et de réaliser de nouveaux connecteurs plus complexes en reposant sur cette classification. Cette classification des connecteurs est orientée fonctionnalité, la différence entre les différents connecteurs repose sur leur usage et non pas sur la nature des connecteurs. Les auteurs classent les éléments de communication en quatre catégories de services qui servent à assurer quatre fonctionnalités de communication indépendantes de l'application :

1. La communication : les connecteurs de communication assurent le transfert de données entre les composants. Ils constituent des blocs de construction primaires pour l'interaction des composants. En effet, les composants se passent habituellement des messages, échangent des données à traiter et se communiquent les résultats des calculs.
2. La coordination : les connecteurs de coordination assurent le transfert de contrôle entre les composants. Les composants interagissent en se passant des threads d'exécution. L'appel de fonction et l'invocation de méthode sont des exemples de ces connecteurs.
3. La conversion : ces connecteurs convertissent l'interaction requise par un composant à celle fournie par un autre composant pour permettre à des composants hétérogènes d'interagir.
4. La facilitation : les connecteurs de facilitation permettent d'optimiser et de faciliter les interactions entre les composants. Des exemples sont les services de planification et le contrôle de concurrence.

Cette classification exprime clairement la volonté d'attribuer aux connecteurs un rôle important en architecture logicielle comportant des sémantiques riches, touchant une multitude de domaines, et non restreints à de simples canaux sans sémantique. Cependant, ce travail reste un travail qui répertorie les connecteurs usuels mais ne fournit pas d'éléments pour guider leur évolution.

Un autre travail sur les connecteurs a été fait dans la classification des ADLs par Medvidovic *et al.* [56] :

« Les *Connecteurs* sont des blocs de construction architecturaux utilisés pour modéliser les interactions entre les composants et les règles qui régissent ces interactions⁸ ».

En plus de cette définition générale, les auteurs donnent les grandes lignes sur les caractéristiques que doit avoir un connecteur. Ces caractéristiques et éléments descriptifs des connecteurs sont identiques à ceux donnés aux composants, à savoir : l'interface, le type, la sémantique, les contraintes, l'évolution et les propriétés non fonctionnelles. Ainsi, cette définition ne donne également qu'une différence de fonctionnalité entre les composants et les connecteurs. C'est une définition très générique et abstraite ; nous allons la préciser.

⁸“*Connectors* are architectural building blocks used to model interactions among components and rules that govern those interactions.”

2.2.2 Notre Approche

Notre définition des connecteurs adhère tout à fait à ces définitions. Nous soutenons et maintenons la différenciation de fonctionnalité entre les composants et les connecteurs. Nous adhérons à la définition précisant qu'à la base un connecteur est un concept d'architecture qui fait du transfert de données mais qu'il ne doit pas être réduit qu'à cela. Les connecteurs peuvent être augmentés avec du contrôle pour pouvoir prendre des décisions. Ces décisions doivent être en relation avec la communication ou la coordination, et non pas en relation avec les fonctionnalités de l'application. Les connecteurs peuvent faire intervenir plusieurs participants pour réaliser des interactions multipoint. En outre, nous supportons l'idée de maintenir cette interaction et sa sémantique lors de l'évolution dans le processus de développement comme entité à part entière. Ceci a pour but de ne pas perdre cette interaction et d'éviter son éparpillement dans le code des composants.

Pour atteindre ces objectifs, nous définissons des blocs de construction de connecteur additionnels et nous précisons les blocs de construction déjà définis, notamment par Mehta *et al.* En plus des canaux et des mécanismes de transfert de données et de contrôle, nous ajoutons les notions de : propriété, prise, et protocole. La notion de *propriété* correspond à l'intention de communication que doit assurer le connecteur. La communication est assurée à travers les *prises*, qui sont les points d'attache du connecteur, au-dessus d'un *protocole*, qui représente les règles de communication ou de coordination. Nous aborderons plus en détails chacune de ces notions dans les sections qui suivent. Nous définissons également une différence supplémentaire entre les connecteurs et les composants logiciels conventionnels : une différence de catégorie relative à la nature des interfaces. Cette différence de nature nous permet de séparer les connecteurs recensés par Mehta *et al.* en deux entités différentes pour un meilleur usage et une meilleure compréhension. Nous distinguons ainsi deux catégories d'abstraction de communication : les connecteurs, et les composants de communication ou les médiums [17]. Nous nous focalisons dans cette section sur les connecteurs, et nous traiterons la différence entre ces deux modes de communication dans le chapitre 4.

Nous considérons que le connecteur est une entité architecturale abstraite et nous réservons ce mot pour désigner uniquement cet élément d'architecture. Ce terme est généralement utilisé par les différents intervenants dans le processus de développement d'application pour faire référence à tout ce qui est relatif à l'interaction entre les composants quel que soit le niveau d'intervention. Ceci amène à désigner par le même nom des entités très différentes, et induit par conséquent à des ambiguïtés de dialogue entre les différents intervenants pour le développement des applications. Afin de palier à cela, nous incitons à distinguer la nature des interactions ou des communications entre les composants suivant le niveau du processus de développement dans lequel on se situe. Nous considérons que les communications ou les interactions entre les composants évoluent et prennent des formes différentes. Nous associons ainsi une dimension temporelle au connecteur où il se transforme pour correspondre, à chaque niveau d'abstraction du développement logiciel, à une entité différente. Nous désignons ce processus de transformation comme le *cycle de vie* du connecteur. Ce cycle de vie est différent de celui des composants de communication et indépendant de celui des composants conventionnels. Ainsi, nous définissons un nouveau vocabulaire pour désigner ces entités différentes dans chacune des étapes du cycle de vie :

- Le connecteur est une entité d'architecture abstraite ;
- Il s'agit d'un générateur de code lorsqu'il est implémenté et déposé sur étagère ;

Niveau Etat	Architecture (abstrait)	Implémentation (concret)
Isolé (sur étagère)	<i>Connecteur abstrait</i>	<i>Générateurs</i>
Lié (assemblé)	<i>Connexion</i>	<i>Composant de liaison</i>

Tableau 2.1 — Vocabulaire par niveau d’abstraction et raffinement

- Il forme une connexion lorsqu’il est assemblé avec d’autres composants ;
- Et il est concrétisé comme un composant de liaison lorsque le système est déployé.

Le tableau 2.1 résume ce nouveau vocabulaire. Certains blocs de construction que nous avons ajoutés au connecteur ont la particularité d’être abstraits. Ils ont la caractéristique de se concrétiser au fil de l’avancement dans ce cycle de vie, en fonction des transformations du connecteur.

2.2.3 Définition du connecteur

Nous définissons un connecteur comme une entité abstraite qui doit assurer une propriété de communication. Il s’agit d’une entité potentielle qui se concrétise au fil du temps, en avançant dans le processus de développement logiciel. Il existe uniquement pour exprimer la propriété ou la sémantique de la communication.

Un connecteur est un élément d’architecture abstrait. C’est la spécification de réification d’un système d’interaction, de communication, ou de coordination d’une application. Il existe pour servir les besoins de coordination et de communication des composants interagissant. À ce niveau, il exprime une interaction ou une communication entre deux ou plusieurs entités non spécifiées et fictives. C’est une entité à part entière qui existe indépendamment de toute composition avec d’autres composants. Il est destiné à être assemblé avec d’autres composants pour réaliser les propriétés générales d’une application. Il n’est ni compilable ni déployable en tant que tel, il doit être assemblé pour être compilé puis déployé.

Un connecteur n’offre pas de services explicites qui seront utilisés par d’autres composants. Il propose d’assurer une propriété de communication sans la réifier comme des services. Il offre des moyens de description pour assurer un service de communication qui est l’intention ou la sémantique de communication. Le connecteur est indépendant de tout modèle, plate-forme ou technologie. Il offre des mécanismes d’interaction indépendants de l’application et contribue à assurer des propriétés non fonctionnelles telles que la sécurité et la fiabilité. Il existe pour servir comme intermédiaire ou comme un médiateur entre les composants interagissant. Il n’offre pas d’interfaces explicites pour réaliser la communication lui-même, mais s’adapte aux besoins des composants pour réaliser leur communication. Le service de communication existe plus à travers de la propriété exprimée que par les interfaces. Ces dernières sont présentes mais ne sont pas expressément énoncées, ce sont des interfaces implicites.

Nous allons décrire les blocs de construction que nous ajoutons au connecteur qui lui permettent d’assurer ces fonctionnalités :

Propriété : Le connecteur met en œuvre la sémantique de communication ou de coordination entre les composants. Il exprime l’intention avec laquelle seront assemblés les composants de l’application. Nous appelons ceci la *propriété* du connecteur. La propriété

représente la fonctionnalité du connecteur qui doit être indépendante de la fonctionnalité des composants. Elle correspond à la propriété de communication ou à la propriété non fonctionnelle de l'application. Elle est décrite indépendamment des composants interagissant. Elle représente le critère ou l'élément de distinction entre les connecteurs au niveau architecture et ces derniers portent le nom de leur propriété. Par exemple, si on veut utiliser une communication par appel de procédure entre deux composants distants, on utilise la propriété de communication RPC. On peut citer comme d'autres exemples de propriété l'équilibrage de charge ou le consensus qui peuvent être mises en œuvre sous forme d'un connecteur.

Prise : Le connecteur interagit avec le monde extérieur pour lui assurer sa propriété via des points d'attache que nous appelons *prises*. Celles-ci correspondent aux extrémités des canaux auxquels nous attachons une importance particulière. Les prises sont des interfaces puisqu'elles permettent l'échange d'information entre le connecteur et les autres composants mais elles correspondent à des interfaces particulières. En effet, une des définitions des interfaces est qu'elles décrivent un service d'un composant indépendamment d'une implémentation [45]. Cependant, nous ne retenons pas cette définition d'une interface de connecteur car celui-ci n'est pas censé rendre un service de communication, il permet d'assurer une propriété de communication. Le connecteur n'est qu'un intermédiaire qui définit des interfaces pour faire communiquer des composants et non pas pour communiquer avec ces composants. Les prises ne sont pas des services auxquels les composants ont accès, elles décrivent seulement le statut ou la position d'un composant dans l'interaction. Elles sont associées à des contrats [11]. Ces contrats décrivent des contraintes sur les composants, plus particulièrement leurs ports, qui sont susceptibles de participer à l'interaction. Un exemple de contraintes dans le connecteur RPC est que les données soient sérialisables.

Un connecteur peut avoir plusieurs prises. Leur nombre correspond au nombre des différents participants à l'interaction, et chaque prise peut avoir une multiplicité. Par exemple, le connecteur RPC contient deux prises, une qui doit être connectée à un composant client qui émet des requêtes et l'autre prise doit être connectée à un serveur qui traite ces requêtes. Ce connecteur simple est de multiplicité (1,1), il connecte un client à un serveur. Le connecteur RPC existe pour assurer la propriété de distribution, et ses interfaces sont implicites. En effet, il possède des points d'interconnexion qui ne sont pas visibles au niveau architecture et qui ne sont pas directement utilisables par les deux composants qui veulent interagir. Il n'y a pas d'appel explicite des prises.

Protocole : Les prises du connecteur sont destinées à communiquer entre elles pour pouvoir assurer la propriété de communication ou de coordination aux composants qui y sont attachés. Ces prises communiquent à travers un *protocole*. Le protocole⁹ constitue un ensemble de règles qui régissent le fonctionnement du connecteur. Il exprime un début de solution pour la réalisation de la propriété du connecteur sur des plates-formes (futures à définir). Il représente « l'intelligence » qui permet de réaliser la propriété. Le protocole est à définir dans une autre phase de processus de développement logiciel. Le mentionner à ce niveau d'architecture permet de prévoir une solution abstraite pour la réalisation de la propriété sans se restreindre à des outils de réalisation d'une plate-forme particulière. Par exemple, le protocole du connecteur RPC décrit qu'il faut transférer les données reçues de la prise client vers la prise serveur, que le client doit attendre la réponse (appel synchrone), et qu'il faut transférer la réponse de la prise serveur vers la prise client et

⁹Le terme protocole ne désigne pas forcément un protocole réseau, nous le décrivons ici d'un point de vue architecture.

lui redonner le contrôle. Ce protocole est ensuite concrétisé au-dessus de protocoles de communication différents selon qu'il soit réalisé sur une plate-forme Java, CORBA, ou autre.

Nous avons proposé de représenter les connecteurs sous forme d'une ellipse pour les différencier des composants logiciels conventionnels représentés par des boîtes. Pour des raisons de clarté des explications, nous introduisons une nouvelle notation graphique non formelle pour représenter le connecteur ainsi que ses nouveaux blocs de construction comme illustré dans la figure 2.9. Le connecteur est représenté par deux ellipses imbriquées l'une dans l'autre, l'ellipse extérieure étant entourée par des cercles. Dans cette figure :

- L'ellipse extérieure représente la propriété que doit assurer le connecteur. Elle est représentée par une ligne continue car, à ce niveau d'architecture, elle représente le premier bloc de construction concret du connecteur qui doit être identifié et décrit. La propriété représente un invariant qui existe pendant tout le cycle de vie du connecteur que nous allons définir ;
- Les cercles autour de cette ellipse extérieure représentent le deuxième bloc de construction : les prises. Elles y sont représentées par une ligne en pointillés car elles ne sont pas encore spécifiées à ce niveau. Elles sont abstraites en attendant d'être concrétisées plus tard dans le cycle de vie. Ce qui est concret dans cette représentation, c'est leur nombre et leur multiplicité. La description du connecteur doit indiquer le nombre de participants que doit faire interagir le connecteur ainsi que la multiplicité des prises, à savoir le nombre autorisé de composants de même type ;
- Le troisième bloc de construction, le protocole, est représenté par l'ellipse intérieure avec une ligne discontinue. Ceci a pour but de préciser que le protocole est abstrait. Il existe intuitivement mais il doit être spécifié ultérieurement pour réaliser la propriété sur une plate-forme choisie.

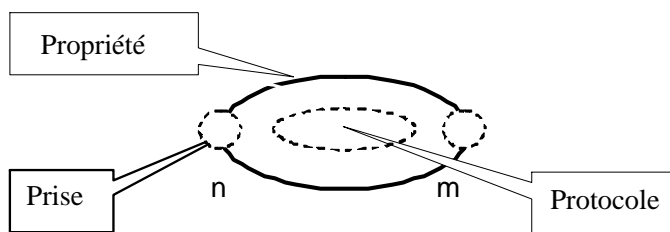


Figure 2.9 — Le connecteur

En plus des entités définies ci-dessus avec lesquels nous précisons la définition du connecteur, nous le considérons comme une entité qui évolue dans le processus de développement de logiciels pour concrétiser ses éléments abstraits. Ainsi, nous associons aux connecteurs un *cycle de vie* que nous présentons dans la section qui suit. En outre, nous proposons de ne plus utiliser le mot connecteur pour désigner toutes les autres entités qui représentent son évolution. En effet, toutes ces entités sont différentes du connecteur puisque chacune d'elles définit des éléments concrets différents et garde des éléments abstraits différents. Pour cela, nous introduisons un nouveau vocabulaire pour désigner les différentes transformations du connecteur durant son cycle de vie pour avoir des dénominations plus justes et plus précises.

2.2.4 Cycle de vie du connecteur

Dans la section précédente, nous avons défini le connecteur comme une entité abstraite. Beaucoup d'autres approches définissent ainsi les connecteurs mais s'arrêtent à ce niveau d'abstraction. Ceci justifie l'écart existant entre la définition de l'abstraction de communication au niveau architecture et son implémentation réelle sur les plates-formes. Les implémentations des interactions entre les composants sont réalisées d'une manière ad hoc sur les plates-formes et sont mélangés aux codes des composants causant ainsi leur disparition. Contrairement à ces approches, nous poussons notre définition plus loin en la considérant comme une entité qui évolue en parcourant les niveaux d'abstraction du processus de développement. Nous lui attribuons ainsi un cycle de vie propre, indépendant du cycle de vie des composants. Ce cycle de vie est défini sur 4 étapes, orthogonales deux par deux : deux phases et deux états. Les deux phases importantes dans le cycle de vie sont : la spécification et l'implémentation du connecteur. Les deux états du connecteur sont : isolé, et assemblé avec des composants fonctionnels. Le tableau 2.1 (page 40) résume cette classification.

Nous réservons le mot *connecteur* à l'entité abstraite, c'est-à-dire à la description d'un système de communication ou de coordination. Le connecteur correspond à des entités différentes en évoluant dans le cycle de vie, c'est pour cela que nous désignons ses différentes entités par un nom propre à chacune des transformations. Une nouvelle entité apparaît à chaque intersection entre une phase et un état. Le connecteur isolé est implémenté sur des plates-formes et des technologies différentes comme une famille de *générateurs de code*. L'assemblage au niveau architecture du connecteur avec d'autres composants forme la *connexion*. La concrétisation de cette connexion se fait en choisissant un générateur de code et en l'activant pour obtenir un *composant de liaison*.

Cette classification répond à deux objectifs nécessaires pour une bonne qualité du logiciel : la séparation des responsabilités ainsi que la séparation entre la spécification et l'implémentation. Le premier objectif consiste à séparer les responsabilités de communication de celle des fonctionnalités des composants. Il est assuré dans notre approche par l'isolation de l'abstraction de communication dans un connecteur à assembler avec les composants fonctionnels, c'est-à-dire le passage de l'état isolé à assemblé. Le second objectif consiste à séparer les deux niveaux de définition de connecteur, au niveau architecture en tant qu'entité abstraite, et au niveau implémentation en tant qu'entité implémentée au-dessus d'une plate-forme, à utiliser en vue d'un déploiement.

Ainsi, le cycle de vie d'un connecteur couvre les différentes phases du développement du logiciel à savoir : la spécification, l'implémentation, l'assemblage et le déploiement. Le connecteur décrit des éléments concrets au niveau architecture comme la propriété, les contraintes et le nombre des prises, et définit des éléments abstraits comme le protocole et le type des prises. Ces éléments abstraits se concrétisent dans les autres entités du cycle de vie que nous décrivons succinctement dans ce qui suit.

Le générateur

Au niveau architecture, le connecteur isolé doit être décrit d'une manière abstraite indépendamment de toute plate-forme pour éviter les éventuelles influences de celle-ci sur les descriptions. De plus, il définit des prises qui ne sont pas destinées à être appelées explicitement. Son implémentation en tant qu'entité isolée doit combler et satisfaire ces deux exigences. D'abord, elle consiste à choisir une plate-forme pour y intégrer la propriété du

connecteur et la déposer sur étagère, dans un but d'utilisation puis de déploiement. Ensuite, elle doit assurer l'indépendance vis-à-vis des composants interagissant puisque à ce stade ces derniers ne sont pas encore connus. Pour cela, nous avons choisi d'implémenter le connecteur sur une plate-forme donnée sous forme d'un générateur de code. En fait, un connecteur est implémenté sous forme d'une famille de générateurs de code, chacun d'entre eux est réalisé au-dessus d'une plate-forme différente. Ce code représente la réalisation de la propriété du connecteur sur la plate-forme. Le générateur prend en charge les détails de la plate-forme sans avoir des informations concernant les composants logiciels à connecter.

Cette réalisation sous la forme d'un générateur, nous permet de faire perdurer l'abstraction de communication en l'implémentant comme une entité à part entière. Elle l'est déjà au niveau architecture et elle le reste au niveau implémentation puisque cette abstraction est indépendante et n'est pas mélangée avec les composants interagissant. Elle permet d'assurer la séparation des responsabilités puisque la sémantique de communication n'est pas réalisée dans le code des composants interagissant. Elle est capitalisée dans une seule entité ce qui évite qu'elle soit perdue ou dispersée dans le code des composants. La flexibilité du mécanisme de génération permet d'éviter d'interconnecter les composants d'une manière fixe pour une seule sémantique de communication. Ce mécanisme de génération constitue aussi un moyen efficace de réutilisation car on réalise la propriété sur la plate-forme une seule fois comme un générateur, mais la réutilisation du générateur est illimitée.

L'élément à concrétiser dans cette étape est le protocole — niveau abstrait — pour la plate-forme cible — niveau concret. Il faut définir les algorithmes qui permettent de réaliser la sémantique de la communication, ainsi que la technologie sous-jacente. Ainsi, c'est l'ellipse intérieure dans la représentation non formelle du connecteur qui se concrétise comme illustré dans la figure 2.10. Elle représente le choix d'une plate-forme donnée pour y intégrer la propriété de communication ou de coordination du connecteur. Le connecteur peut être réalisé au-dessus de plusieurs technologies différentes, ce qui se traduit par la mise à disposition d'une famille de générateurs.

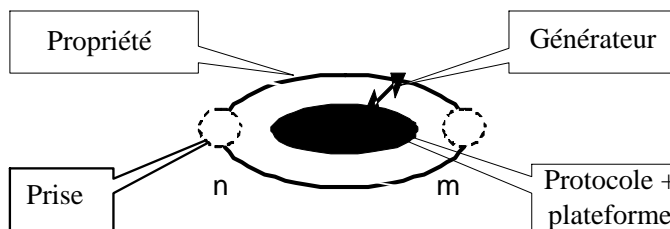


Figure 2.10 — Le générateur

Dans cette même figure, on remarque que les prises sont toujours représentées par des lignes discontinues. À ce stade du cycle de vie, elles sont toujours abstraites et génériques. Elles ne sont pas encore spécifiées, mais elles contiennent plus de contraintes, relatives à la plate-forme, sur les composants qui s'y connecteront.

Le générateur permet de garder une trace du connecteur depuis l'architecture jusqu'au déploiement. Il évolue comme une entité à part entière. La sémantique du connecteur (ou la propriété) ne se perd pas en avançant dans le processus de développement. La réalisation des générateurs doit remplacer les réalisations conventionnelles d'une abstraction de communication qui sont habituellement faites en combinant des moyens de communication simples dans le code des composants. Cette technique de génération permet une réutilisation massive

de ces abstractions puisqu'elles ne sont développées qu'une seule fois et réutilisées pour des milliers d'applications.

La connexion

Au niveau architecture le connecteur est décrit indépendamment de tout composant. C'est une entité abstraite destinée à être assemblée ou composée avec d'autres composants toujours à ce même niveau d'architecture. Pour ce faire, les ports des différents composants sont rattachés aux prises du connecteur et forment ainsi la connexion. C'est la représentation du connecteur « lié » au niveau architecture.

Dans une application, un composant offre des services qui seront utilisés par d'autres composants en suivant une certaine sémantique de communication. Il peut aussi avoir besoin des services offerts par d'autres composants en suivant une autre sémantique de communication. Le connecteur s'intercale entre ces composants pour assurer cette sémantique ou propriété de communication. Il agit comme un « médiateur » dans cette connexion. Il permet de trouver un accord entre les participants sans pour autant influencer sur les interfaces ou le fonctionnement interne des composants. Ces derniers n'utilisent pas des services offerts par le connecteur, mais l'utilisent comme un intermédiaire pour accéder aux services d'autres composants. Les services offerts et requis sont ceux des composants, le connecteur doit assurer la sémantique de communication avec transparence, d'où la nécessité de ses prises génériques. Ces prises sont spécialisées à la connexion en *adoptant* les services offerts et requis des composants interagissant pour les offrir et les utiliser en leur appliquant la sémantique de la communication avec transparence. Nous appelons ces prises transformées *les interfaces de la connexion*, elles sont identiques aux interfaces des composants. D'un côté, le connecteur adopte les interfaces du composant offrant le service, pour les offrir au composant qui requiert ce service tout en cachant la propriété de communication. D'un autre côté, il prend la place du composant qui requiert le service pour recevoir le service demandé et le remettre au vrai composant requérant en cachant la propriété de la communication.

À la connexion, les éléments qui se concrétisent dans le connecteur sont les prises. Nous les représentons avec des cercles remplis dans la représentation graphique informelle du connecteur comme l'illustre la figure 2.11. Désormais, on connaît les interfaces pour lesquels la propriété doit être assurée, l'ellipse extérieure se complète. Les interfaces représentent les éléments distinctifs des connexions à ce niveau.

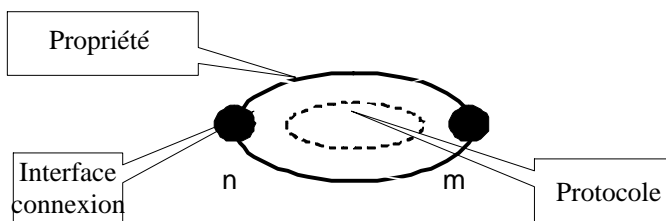


Figure 2.11 — La connexion

L'élément qui reste abstrait à ce stade du cycle de développement est le protocole ou la plate-forme sur laquelle va s'installer cet assemblage. Le protocole reste toujours représenté avec une ellipse en pointillés dans la figure 2.11. À l'assemblage on n'utilise que la propriété, on ne s'intéresse pas aux détails d'algorithmique et d'implémentation du connecteur. Ainsi, on réalise un assemblage en faisant abstraction de la technologie sous-jacente. La connexion

entre les composants n'est plus contrainte par les moyens de connexion offerts par la plate-forme. Cette abstraction nous fait gagner en réutilisabilité à la fois des connecteurs et des composants. On reste focalisé sur les services, sachant que les détails techniques des plates-formes seront pris en charge par les générateurs.

Il faut noter la différence entre un connecteur et une connexion. Le connecteur existe seul. La connexion apparaît lorsque les composants sont définis, la sémantique de communication ou le connecteur choisi, et l'ensemble connecté.

Le composant de liaison

À partir de la connexion logique décrite dans la section précédente, on obtient la connexion physique par l'application du générateur sur la connexion logique. Nous appelons ce résultat le composant de liaison. Le générateur et la connexion sont deux entités complémentaires. Chacune des entités définit des éléments abstraits qui se concrétisent dans l'autre et leur fusion génère une entité complètement concrète. D'un côté, le protocole abstrait de la connexion se concrétise grâce au protocole et la plate-forme définis par le générateur. D'un autre côté, les prises abstraites du générateur sont comblées avec les interfaces de la connexion. Ces prises sont concrétisées en se transformant en proxies. L'ensemble des proxies représente la réalisation de la propriété du connecteur pour les interfaces des composants interagissant au-dessus de la plate-forme choisie. La représentation graphique en ellipse se complète, tous les éléments se concrétisent. Comme l'illustre la figure 2.12, il n'y a plus de parties abstraites en pointillés. Le composant de liaison rattaché aux composants (la connexion physique) représente l'application à déployer.

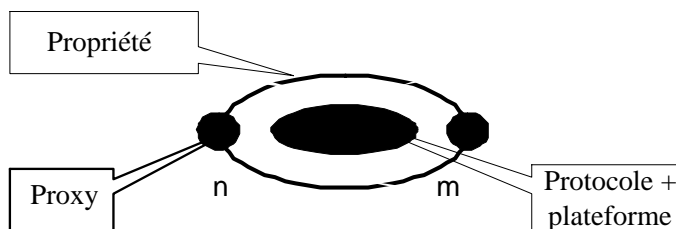


Figure 2.12 — Le composant de liaison

À ce niveau du cycle de vie, l'application déployée consiste en un ensemble de composants interconnectés. Le connecteur s'est transformé en un composant de liaison avec des interfaces explicites. En effet, il offre désormais les interfaces adoptées des composants interagissant. À l'exécution, les composants logiciels interagissent effectivement avec les interfaces du composant de liaison (les proxies) qui sont en réalité la réincarnation des services que ces composants offrent et requièrent. Les proxies empruntent ces services pour les offrir en cachant la propriété de communication ainsi que les détails des plates-formes. Le composant de liaison n'est plus transparent aux composants, il est appelé explicitement pour rendre le service dont le composant a besoin. Il ne rend pas le service lui-même car il ne l'implémente pas — c'est le composant fonctionnel qui l'implémente et le rend réellement — mais il donne une réalisation de l'interface afin de permettre l'accès au service en lui appliquant la propriété du connecteur. Cette propriété reste transparente au composant puisque le service lui parvient sans qu'il prenne connaissance de l'intention de communication prévue initialement.

2.2.5 Synthèse

Nous définissons donc un *connecteur* comme la réification d'une abstraction d'interaction, de communication ou de coordination. Le connecteur est une entité d'architecture potentielle, non déployable ni compilable, qui assure une propriété ou une intention de communication. Il existe comme une entité à part entière, sans aucun attachement à des composants. Il exprime dans un premier temps une interaction entre des intervenants fictifs qui se spécialisent lors de son utilisation. Pour ce faire, il fournit des interfaces implicites génériques et abstraites, appelées *prises*, qui lui permettent de s'adapter à tout composant l'utilisant. Ces prises sont abstraites et agissent comme des conteneurs à combler plus tard dans le cycle de vie. Cette notion de prise offre comme avantage de réaliser des interconnexions souples et facilement modifiables. Le connecteur définit un *protocole* abstrait qui spécifie son comportement. Ce protocole est séparé des prises du connecteur de la même façon que la description du fonctionnement d'un composant est séparée des interfaces de ses ports.

Avec ses interfaces implicites, le connecteur est différent d'un composant logiciel. Il possède ainsi un cycle de vie différent dans lequel il se transforme. Il parcourt les niveaux d'abstraction du processus de développement en concrétisant ses parties abstraites. Pour une meilleure maîtrise et compréhension de l'entité connecteur, nous introduisons un nouveau vocabulaire pour désigner ses transformations durant son cycle de vie. Le connecteur définit deux invariants qui existent tout le long du cycle de vie qui sont : la propriété et le nombre de prises différentes (leur statut). Les autres éléments du connecteur, tels que le protocole et le type des prises, restent abstraits. Ils se concrétisent plus loin dans le cycle de vie. Les prises seront spécifiées d'un point de vue service et plate-forme.

Lorsqu'au niveau architecture le connecteur est attaché aux composants ayant besoin de la propriété du connecteur pour communiquer, on appelle l'ensemble une *connexion*. Le connecteur prend connaissance des composants qu'il va interconnecter et adapte ses prises aux interfaces de ces composants. Les prises génériques du connecteur se spécialisent et on obtient les *interfaces de connexion*. Le protocole de communication quant à lui reste abstrait. On ne s'intéresse pas aux détails de la plate-forme sous-jacente ou à l'algorithmique de la connexion, seule la propriété est importante.

Le connecteur est réalisé comme un *générateur* de code. Le générateur utilise la plate-forme sous-jacente et choisit une solution pour la réalisation de la propriété du connecteur sur cette plate-forme. Il permet de garder l'abstraction de communication du connecteur intacte jusqu'au déploiement et de ne pas la perdre en la réalisant dans le code des composants interagissant. Ainsi on assure le raffinement du connecteur et la séparation des responsabilités. Les prises du générateur restent génériques, elles représentent des emplacements à combler avec les interfaces de la connexion.

Lorsque ces interfaces sont passées en paramètres au générateur, ce dernier génère le *composant de liaison* qui représente une entité entièrement concrétisée. Le connecteur est transformé en un composant de liaison avec des interfaces bien définies en adoptant les interfaces de la connexion et en effectuant la communication effective au-dessus de la plate-forme choisie.

Nous détaillons dans la suite le processus de conception et d'implémentation du connecteur.

2.3 Processus de conception et d'utilisation des connecteurs

Comme toute entité logicielle, le connecteur doit être spécifié pour pouvoir être implémenté sous forme de générateur, et utilisé pour former une connexion. Nous avons vu dans la section précédente une vision structurelle du connecteur avec une description non formelle des éléments abstraits et concrets de chacune de ses transformations ; c'est-à-dire le quoi ? Le but de cette section est de présenter, sous un angle plus formalisé, la notion de connecteur, de ses transformations, ainsi que le passage d'une entité à une autre ; c'est-à-dire le comment ? Pour cela, nous allons utiliser le langage semi-formel UML [66] pour décrire les principaux éléments d'un connecteur dans toutes les étapes qu'il traverse.

Associer un cycle de vie au connecteur implique l'existence de plusieurs intervenants qui agissent sur le connecteur pour le transformer d'un état à un autre dans ce cycle de vie. Le diagramme UML, illustré par la figure 2.13, résume les cas d'utilisation des phases parcourues par le connecteur, ainsi que les différents intervenants sur ces phases.

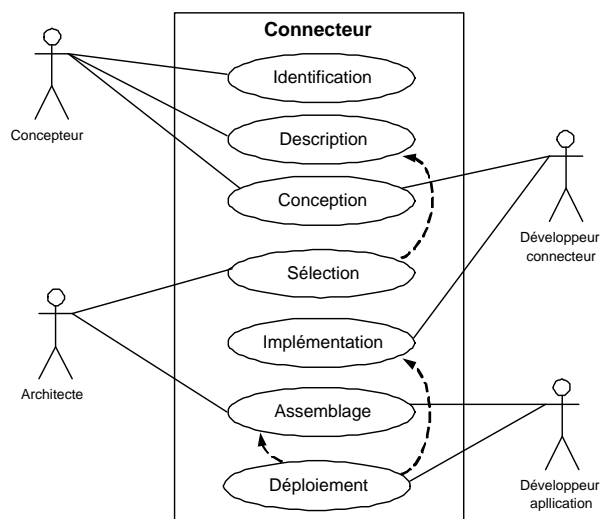


Figure 2.13 — Diagramme de cas d'utilisation du connecteur

Les phases d'identification et de description des connecteurs sont réalisées par un concepteur. Ce dernier participe aussi à la phase de conception du protocole du connecteur en vue de son implémentation comme des générateurs par des développeurs de connecteur. Les phases de sélection et d'assemblage du connecteur avec d'autres composants sont assurées par un architecte afin de former la connexion. Enfin, un développeur d'application participe à la réalisation de cet assemblage en sélectionnant le bon générateur pour obtenir le composant de liaison dans le but d'assurer la phase de déploiement. Dans les sections suivantes, nous allons détailler chacun de ces cas d'utilisation. Pour chacune des phases, nous décrivons les éléments qui la composent ainsi que les transformations de ces éléments d'une phase à une autre.

2.3.1 Identification et description

La phase d'identification et de description consiste à identifier des abstractions de communications utiles et de les réifier sous forme d'un connecteur. Il s'agit ainsi de définir et de

spécifier pour chaque connecteur les blocs de construction qu'il doit exprimer. On se positionne dans la case haute gauche du tableau 2.1 en page 40 (cf. figure 2.14).

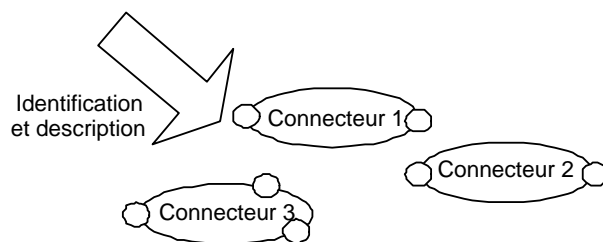


Figure 2.14 — Phases identification et description

Le but de cette phase est de chercher des solutions génériques à des problèmes de communication récurrents et de les réifier comme un connecteur. Il s'agit de capitaliser l'expertise de communication, habituellement prise en charge tardivement dans le processus de développement, dans une entité d'architecture pour plus de réutilisation. Il s'agit ainsi d'anticiper l'intégration de la communication au niveau d'architecture et ne pas la considérer comme un détail de développement. En effet, dans ce dernier cas les solutions sont divergentes, faites et refaites à chaque fois qu'elles sont nécessaires, sans quelles soient véritablement optimales. La finalité de ce processus d'identification est de constituer, à terme, un catalogue de connecteurs à mettre à la disposition des développeurs et des architectes afin de les réaliser et de les utiliser.

Une fois un connecteur identifié, il faut décrire les deux éléments qui le caractérisent ; c'est-à-dire la propriété, les prises (leur nombre, leur statut et leur multiplicité autorisée) et le protocole de communication ou de coordination. Une vision structurelle et statique du connecteur est représentée dans le diagramme de classe en figure 2.15. Ce diagramme définit les éléments qui constituent un connecteur au niveau architecture ainsi que les relations entre eux.

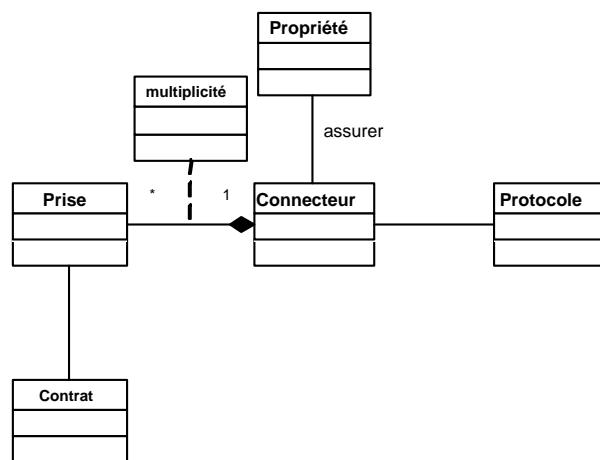


Figure 2.15 — Description d'un connecteur

L'intervenant principal dans cette phase est le concepteur. Il doit décrire et spécifier l'abstraction de communication ; c'est-à-dire décrire l'intention avec laquelle les composants interagiront indépendamment de la réalisation de cette interaction (la réalisation étant spécifiée

par le protocole). Nous avons désigné cette intention comme la propriété du connecteur, elle représente son élément invariant. En effet, contrairement aux autres éléments du connecteur elle est présente dans toutes les étapes du cycle de vie du connecteur, où elle n'y subit aucune transformation. Elle représente l'élément concret dans cette phase. À présent, la spécification de cette propriété peut se faire par un simple nom et en ajoutant éventuellement plus de détails avec le langage naturel.

En plus de la propriété, le concepteur doit décrire chacun des autres éléments du connecteur :

- Les prises du connecteur : leur description s'effectue en définissant leurs noms et les contraintes. Celles-ci constituent les conditions que doivent satisfaire les composants pour pouvoir s'y connecter et sont représentées sous forme de contrats. Il faut aussi préciser les multiplicités des prises afin de connaître, pour chacune d'elles, le nombre de prises identiques autorisé. Les contrats des prises ne spécifient pas des opérations requises ou offertes puisque le connecteur n'offre pas de services et n'utilise pas les services des composants. Ils peuvent spécifier des propriétés à récupérer des composants à connecter pour les besoins d'une coordination particulière (du protocole de coordination). Ces propriétés des prises ne doivent pas être décrites comme des signatures de méthodes. Elles doivent être assez génériques pour permettre une interaction avec tout composant satisfaisant les contraintes. Un exemple de contraintes peut être de spécifier que le composant est dans un environnement réparti et qu'il offre la possibilité de sérialiser ses données dans le but de les transférer par un réseau de communication. Un autre exemple peut être que la prise doit surveiller les informations sur la charge d'un serveur dans le cas d'un équilibrage de charge. Elles indiquent ainsi le minimum requis d'un composant pour pouvoir participer à la communication définie par le connecteur. Bien qu'elles ne définissent pas de services pour accomplir la propriété, les prises existent pour être connectées à des ports de composants afin de servir d'interface pour permettre l'échange d'information et assurer la propriété (c'est un intermédiaire qui laisse passer des données et du contrôle à qui sera appliquée la propriété). Les prises doivent donc comprendre les besoins et les services des ports pour pouvoir communiquer, échanger des informations et s'adapter. Si les interfaces des composants ou les ports sont décrits par un langage de description d'interfaces, alors les prises peuvent être spécifiées par la grammaire d'un langage de description d'interface comme l'IDL (*Interface Definition Language*) ou d'un langage de coordination comme FLO/c [30] ou ISL (*Interaction Specification Language*) [9].
- Le protocole de communication/coordination : sa description s'effectue en spécifiant les interactions entre les prises. Il s'agit de décrire pour la propriété du connecteur, la politique de communication ou de coordination *entre* les prises; c'est-à-dire à l'intérieur du connecteur. C'est un début de description de la solution de communication et de coordination pour la propriété et il peut exister plusieurs variantes de protocoles. Chaque propriété avec un protocole de communication ou de coordination différent constitue un connecteur différent. Il contribue à affiner la propriété du connecteur. Le protocole reste général et doit se spécialiser dans les phases suivantes.

Pour illustrer cette phase, nous avons regroupé dans les exemples ci-dessous quelques propriétés qui désignent un ensemble de connecteurs. Ces derniers possèdent les mêmes prises ainsi que les mêmes contraintes mais la différence entre ces connecteurs est décidée par rapport à la multiplicité des prises et au protocole de communication ou de coordination entre ces

prises.

Exemples

Les applications que nous utilisons manipulent des composants logiciels qui offrent et requièrent des services. Un composant suit ainsi le modèle client/serveur tel que chaque composant peut être client et serveur à la fois. Les propriétés que nous avons identifiées dans cette section s'appliquent principalement à ce modèle. Les connecteurs associés satisfont une propriété commune des prises mais diffèrent dans leur multiplicité ainsi que dans le protocole qui les relie. Comme nous nous positionnons dans un modèle client/serveur, ces connecteurs possèdent deux prises : une reliée à un client qui envoie des requêtes et une reliée à un serveur répondant à ces requêtes. La multiplicité des prises porte sur le nombre de composants susceptibles de se connecter, qui offrent et qui requièrent les mêmes services. Dans ce cas, le protocole de coordination est simple, il définit la politique à suivre pour connecter deux ensembles de composants de même service. On peut citer comme exemples :

1. Invocation : c'est la propriété la plus courante. Les connecteurs associés relient un ou n client à un serveur (multiplicité $(n,1)$). Un connecteur peut être augmenté avec des contraintes d'exclusion mutuelle par exemple : lorsqu'un client envoie une requête, le serveur la traite toute seule et n'accepte aucune autre requête jusqu'à la fin de la requête courante. Le connecteur peut également gérer un système de file d'attente pour gérer l'ordre d'arrivée des requêtes des clients. La plupart des connecteurs qui existent actuellement sur les plates-formes sont de ce type, comme le RPC par exemple, où les propriétés de la communication sont la communication synchrone et la répartition.
2. Choix : c'est une propriété qui s'appuie sur la propriété précédente. Un connecteur qui lui est associé offre à la base le moyen d'invoquer un même service offert par plusieurs serveurs. Ainsi, la multiplicité de la prise serveur est m . Une contrainte associée à cette prise serait qu'*un seul serveur parmi les m* réponde aux requêtes des clients. Le serveur élu est déterminé par le protocole de communication ou de coordination. Un exemple de connecteur de ce type est le connecteur d'équilibrage de charge. Dans cet exemple, le protocole de coordination peut choisir le serveur qui traitera les requêtes soit par une politique qui prend à chaque requête un serveur au hasard, soit par une politique qui choisit le serveur suivant la charge de chacun des serveurs présents dans l'interaction. Dans ce dernier cas, une contrainte supplémentaire est rajoutée à la prise serveur qui est de récupérer la charge du serveur. Ce connecteur peut être utilisé pour améliorer le temps de réponse et la performance des applications.
3. Fusion : cette propriété ressemble aux deux précédentes. Les connecteurs associés permettent d'invoquer un même service offert par m serveurs. Cependant, dans le cas de la fusion, le résultat de l'invocation à donner au client est défini en prenant en considération *toutes les réponses des m serveurs*. Prenons l'exemple d'un connecteur de consensus. Ce connecteur compare toutes les réponses des serveurs et ne les accepte pour répondre à l'invocation que si elles sont *toutes égales*. Dans ce cas une contrainte de la prise serveur est que tous les serveurs participant à l'interaction répondent aux requêtes et pas seulement un parmi les m serveurs. Ce connecteur peut être utilisé dans le cadre d'applications qui manipulent des calculs sensibles. Il doit collecter tous les résultats effectués par des calculateurs puissants, les comparer, et ne renvoie la réponse au client que si tous les résultats sont égaux afin de vérifier la fiabilité de la réponse.

Il peut y avoir plusieurs autres variantes de ces connecteurs, comme prendre les réponses de s serveurs parmi m , de calculer la moyenne des résultats, d'établir une priorité ou un tri, ... et la liste reste ouverte.

Il est possible aussi de définir des connecteurs qui contiennent plus de prises, et pas seulement deux prises comme dans les exemples que nous avons cités ci-dessus. Il s'agit dans ce cas de connecteurs qui impliquent plusieurs prises avec des statuts différents, ce qui signifie que les composants à connecter offrent des services différents et peuvent avoir aussi des multiplicités différentes. Ce genre de connecteur exprime une coordination plus évoluée entre ces différentes prises; ce qui nécessite un protocole de coordination plus complexe qu'on pourrait spécifier à l'aide d'un langage de coordination comme ISL [9]. Ce langage peut décrire des contraintes de causalité, des conditions de synchronisation, etc. Par exemple, on peut citer un connecteur de synchronisation deux par deux qui contient quatre prises, toutes d'une multiplicité maximum de 1. Deux des quatre prises possèdent un statut d'émetteur, et les deux autres possèdent le statut de récepteur. Le protocole de coordination spécifie que chaque deux prises de statut différent doivent être synchronisées.

Tous ces connecteurs seront utilisés par des architectes pour leurs propriétés afin d'assembler des composants, et seront implémentés par des développeurs dans les autres phases, comme nous allons le voir dans les sections suivantes.

2.3.2 Sélection et assemblage

Les architectures des applications sont décrites comme un ensemble de composants interconnectés par des connecteurs. Dans cette phase, un architecte est chargé de sélectionner les composants nécessaires à l'application, ainsi que de sélectionner les connecteurs qui répondent aux besoins de communication de ces composants. Après la sélection de ces éléments, l'architecte est chargé d'interconnecter et d'assembler l'ensemble des composants et des connecteurs choisis formant ainsi la configuration de l'application. Ce processus d'assemblage permet de passer du connecteur, qui est isolé, à la connexion qui comporte l'ensemble des composants et des connecteurs comme décrit dans la figure 2.16.

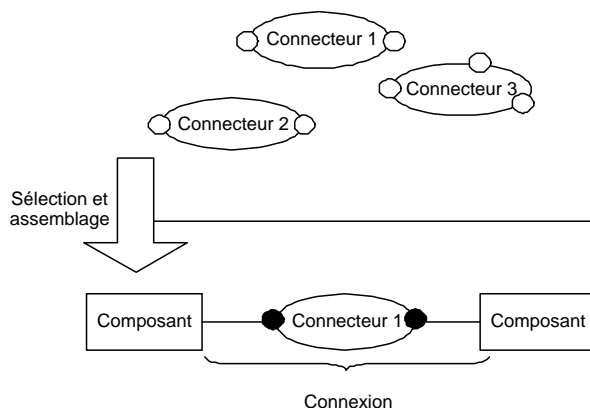


Figure 2.16 — Phases sélection et assemblage

L'assemblage des composants en utilisant des connecteurs reflète l'intention de l'architecte. Ce dernier est en charge de connecter un composant ayant besoin d'un service à un autre composant offrant ce service en choisissant la propriété de communication nécessaire pour l'application à développer. Pour cela, l'idéal serait que l'architecte choisisse une sémantique

de communication de façon à ce que les spécifications des interfaces des composants soient respectées et d'apporter le moins de changements possibles à ces composants. En effet, comme les composants et les connecteurs sont décrits et développés indépendamment, il est préférable de respecter le plus fidèlement possible les spécifications des composants et de ne pas apporter des modifications afin de retrouver facilement leurs correspondants exécutables. Dans le cas contraire, il sera nécessaire de modifier les spécifications des composants en fonction des besoins de l'application. Par conséquent, les nouvelles spécifications et celles des composants déposés sur étagère risquent d'être divergentes, et donc difficilement réutilisables.

L'utilisation du connecteur permet d'atteindre ces objectifs et de remédier à ces problèmes. Le connecteur définit la sémantique de communication (propriété) en permettant de l'introduire dans l'application avec transparence grâce à ses prises génériques. Lorsque les descriptions des composants offrants et requérants les services coïncident, les prises génériques du connecteur s'adaptent aux interfaces spécifiques des composants, et ainsi aucune modification de ces interfaces n'est nécessaire. Dans le cas contraire, le connecteur peut servir d'adaptateur et peut trouver un compromis entre ces interfaces. Ainsi, les composants sont reliés par l'intermédiaire du connecteur pour former une connexion et non pas reliés au connecteur. Ce dernier intervient pour assurer la transparence à la communication.

Le diagramme de classe illustré dans la figure 2.17 décrit la connexion. Il augmente le diagramme de classe décrivant le connecteur isolé avec les éléments nécessaires pour obtenir les *interfaces de connexion* (l'élément concret de cette phase).

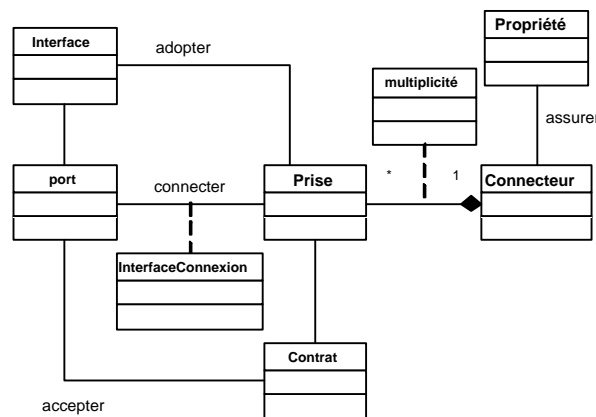


Figure 2.17 — Description d'une connexion

Dans ce diagramme, une connexion est obtenue par une liaison entre une prise du connecteur et un port d'un composant. Cette connexion se traduit comme une relation où une prise adopte les interfaces du composant liées au port. Il s'agit alors d'une spécialisation de la prise et on obtient une interface de connexion. Pour que cela puisse arriver, il est nécessaire que le port accepte le contrat de la prise et vice-versa. Il faut qu'ils soient « compatibles ». Ainsi, si un composant A offre une interface qui est utilisée par un composant B, pour les assembler, un architecte insère un connecteur qui permet d'assurer cette sémantique de communication, qui est l'invocation. Cette insertion est transparente aux composants puisque ces composants n'interagissent en aucun cas directement avec le connecteur. La connexion avec le connecteur se fait par la spécialisation des prises de ce connecteur en adoptant les interfaces des composants. Ainsi, ce n'est pas au composant de s'adapter ou d'adopter le rôle du connecteur comme c'est fait habituellement dans les langages de description d'architecture, mais le contraire grâce au processus de génération que nous décrivons dans la section suivante. C'est

la différence que nous attribuons à notre approche par rapport aux définitions courantes des connecteurs.

Dans cette phase, les types des prises se concrétisent et sont identiques aux types des interfaces des composants. Les interfaces de connexion représentent les contraintes de la sémantique de communication à appliquer sur les interfaces des composants. Le connecteur simule ainsi les services des composants tout en cachant la sémantique de communication. Dans cette phase, les composants sont reliés directement au service dont ils ont besoin. L'intention de la communication est assurée par le connecteur. Ce dernier est transparent aux composants, il représente juste un moyen de communication. On n'utilise pas les services du connecteur, on utilise le moyen connecteur pour accéder aux services d'un autre composant.

Cette séparation des entités au niveau architecture entre entité de communication et entité fonctionnelle permet la séparation des responsabilités pour les développeurs. Elle offre une meilleure réutilisation, évolution et maintenance des composants et des connecteurs. Le développeur de composant n'aura qu'à suivre la spécification fonctionnelle du composant et ne pas se préoccuper des détails de communication. Cette fonction est laissée à la charge d'un spécialiste : le développeur du générateur.

À ce stade du cycle de vie, l'architecte ne prend en considération aucun détail technique concernant la plate-forme. L'assemblage est décrit au niveau architecture en fonction des spécifications et pas en fonction de l'implémentation. Ainsi, l'architecte possède un choix large de connecteur, et n'est pas contraint de se limiter à l'utilisation des moyens de communication offerts par une plate-forme donnée. Il ne s'intéresse qu'à la sémantique de communication sans se préoccuper de comment celle-ci est réalisée. Il focalise les efforts sur les aspects de communication abstraits pour décrire son application et non pas sur la technologie et ses contraintes. Les détails d'implémentation de la sémantique de communication sont pris en charge par la phase suivante. Ainsi, même si l'abstraction n'existe pas sur une plate-forme il faut faire en sorte de la réaliser indépendamment de l'application, car une des motivations de notre travail est d'offrir le moyen de réaliser des sémantiques de communication complexes qui n'existent pas sur les plates-formes.

Exemple

Dans un environnement local, lorsque l'architecte désire connecter un composant qui requiert un service à un autre composant qui offre ce service, il utilise un connecteur de type appel de procédure qui est présent dans la plupart des langages de programmation. Si le composant offrant le service est transféré à un environnement distant, l'architecte utilisera un connecteur de type RPC qui va s'occuper d'assurer l'invocation de service à distance en s'occupant de la transformation des données pour qu'elles puissent traverser le réseau. Ceci ne gêne en aucun cas les composants, puisqu'un composant utilise dans son code un appel de procédure, comme si c'était un appel local, pour appeler le service offert par un autre composant. Pour simuler cet appel local, la prise client du connecteur RPC adopte le service offert par le composant appelé, et l'offre localement au composant appelant. Les détails de comment ce connecteur peut offrir localement ce service ne sont pas importants à ce niveau. L'architecte sait qu'ils existent et qu'ils sont traités ailleurs, donc il ne se préoccupe pas de le décrire ni de savoir si le langage de programmation qui sera utilisé offre cette caractéristique de la communication.

Le connecteur est transparent au composant, c'est l'architecte qui doit définir le comportement global de l'application et choisir le connecteur qui répond aux besoins en communica-

tion. Le connecteur s'occupe de rajouter et de cacher des traitements intermédiaires au service demandé, dans le cas de cette application : la synchronisation, l'accès au réseau, et la transformation de données. Ainsi, les développeurs des composants omettront de considérer ces détails ce qui les décharge d'un travail complexe et répétitif et qui demande des connaissances techniques poussées des couches basses. Ainsi, le niveau physique est caché.

Le comportement de l'application peut changer en changeant l'intention de la communication. Par exemple, si l'architecte veut être sûr que la réponse reçue par le composant appelant est fiable, il doit juste répliquer le composant appelé et remplacer le connecteur RPC par un connecteur de consensus. Ce dernier se charge de tester si toutes les réponses reçues des composants appelés sont égales. Comme les prises sont génériques, la prise client de ce nouveau connecteur adopte le service offert par les composants appelés et l'offre localement au client en lui ajoutant cette propriété de consensus. Ainsi, l'assemblage des composants en utilisant des connecteurs permet une reconfiguration facile de l'architecture de l'application. Il n'est pas nécessaire d'apporter des modifications aux composants puisque les interfaces requises et fournies sont les mêmes, celles des composants, quelle que soit la sémantique de communication utilisée.

2.3.3 Conception et implémentation

Les phases conception et implémentation caractérisent le passage de la description de l'abstraction de communication à sa mise en œuvre. Il s'agit de passer d'une sémantique de communication identifiée et réifiée sous forme de connecteur, à sa mise en œuvre au-dessus d'une plate-forme cible, sous forme de générateurs de code prêt à l'utilisation (figure 2.18). Le choix d'une telle entité de mise en œuvre répond aux soucis :

- De préserver l'abstraction de communication comme une entité à part entière jusqu'à l'implémentation ;
- De l'implémenter indépendamment des composants de l'application ;
- Et de séparer les aspects fonctionnels et architecturaux des choix techniques.

Le générateur permet de garder l'abstraction de communication comme une entité à part entière de l'architecture au déploiement, c'est-à-dire durant tout le cycle de vie. Il représente l'intégration des détails techniques de la plate-forme à cette abstraction. Il reste complètement indépendant de toute application, et l'abstraction indépendante de tout composant.

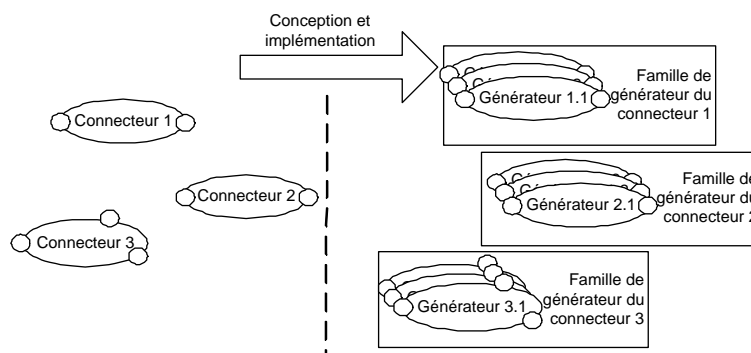


Figure 2.18 — Phases conception et implémentation

Nous avons défini dans la section 2.2.3 les éléments qui constituent un connecteur à savoir : la propriété (qui est l'invariant), les prises, et le protocole de coordination entre les prises. Ce dernier représente l'espace de travail de ces phases de conception et d'implémentation. Le

concepteur doit décrire un raffinement de ce protocole concernant la politique d'interactions entre les prises. Le processus de raffinement permet d'introduire des détails sur des propriétés non fonctionnelles non spécifiées au niveau du connecteur, qu'il n'est pas nécessaire de voir apparaître au niveau architecture. Ainsi, il peut découler plusieurs protocoles de mise en œuvre différents de ce processus de raffinement suivant les détails introduits. Le développeur implémente chaque protocole de mise en œuvre au-dessus d'une plate-forme cible. L'implémentation du connecteur est réalisée comme une famille de générateurs comme représenté en figure 2.18. Pour un même connecteur, ces générateurs peuvent être différents soit d'un point de vue protocole, soit d'un point de vue plate-forme de mise en œuvre.

La réalisation du connecteur comme un générateur représente le moyen idéal pour préserver la généricité des prises puisqu'elles y constituent les paramètres. Le générateur permet ainsi d'assurer la transparence à la communication avec ces interfaces implicites. En effet, le composant ne doit pas poser de contraintes sur l'environnement ; c'est à l'architecte de le faire en choisissant la sémantique de communication entre les composants, ce qui justifie la nécessité de la transparence de cette communication par rapport aux composants communicants.

La phase de conception constitue une étape intermédiaire entre la description et l'implémentation. Elle permet d'introduire des détails non nécessaires au niveau architecture mais nécessaires pour la mise en œuvre du connecteur. Par exemple, le connecteur RPC, décrit au niveau architecture, pose des contraintes sur les prises en indiquant uniquement qu'elles doivent être sérialisables. Par contre dans cette phase de conception, le concepteur doit décrire comment se font les opérations de codage de l'information pour traverser le réseau de communication ainsi que les opérations de décodage des informations après leur récupération du réseau de communication. Le concepteur raffine le protocole de coordination en décrivant quelques algorithmes d'interaction, c'est à dire « l'intelligence » qui régit les prises du connecteur. Ceci aboutit à plusieurs protocoles de mise en œuvre. Ainsi, chaque protocole de mise en œuvre correspond à une solution de communication différente du protocole de coordination. Le diagramme de classe en figure 2.19 illustre les relations entre les éléments du connecteur et ceux du générateur.

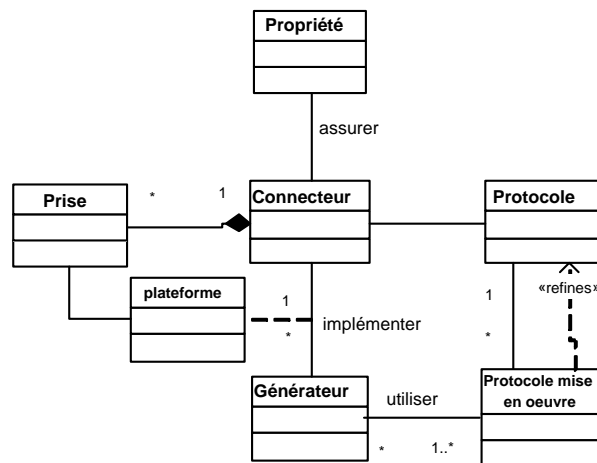


Figure 2.19 — Description du générateur

Les protocoles de mise en œuvre issus du processus de raffinement restent génériques puisque les prises ne sont toujours pas spécifiées. En effet, les composants qui interagissent ne

sont pas encore connus à cet instant. Il faut alors décrire les interactions entre les prises en fonction de leur statut, ou leur rôle dans l'application. Le protocole de mise en œuvre peut être très simple, une prise envoie une requête à une autre prise par exemple ; ou très complexe, en utilisant des composants qui réaliseraient des traitements en fonction du contenu des requêtes. Cependant, bien que les prises restent génériques, elles possèdent plus de contraintes liées au protocole de mise en œuvre et à la plate-forme.

La phase d'implémentation constitue la réalisation effective du générateur de code. C'est à ce niveau qu'il faut prendre en compte les contraintes de la plate-forme, non pas à l'assemblage, ni à la spécification. Le développeur de connecteur a pour tâche d'implémenter les protocoles de mise en œuvre résultant du processus de raffinement au-dessus d'une plate-forme cible existante¹⁰. La réalisation peut se baser sur les moyens de connexion élémentaires offerts par les plates-formes. Il n'y a pas besoin de tout reconstruire, il s'agit juste d'augmenter ces plates-formes avec les propriétés du connecteur afin qu'elle soit réalisée en dehors des composants. Ainsi, le développeur peut utiliser toutes les stratégies et combinaisons possibles pour construire les générateurs puisque, dorénavant, leur évolution et leur maintenance n'affecteront plus l'application. Le générateur répondra à la spécification du connecteur et il n'importe peu comment il est réalisé à l'intérieur.

L'idéal est de réaliser ces générateurs sur plusieurs plates-formes et de les mettre à disposition, afin d'offrir un choix d'utilisation des générateurs. En effet, un développeur qui manipule des composants logiciels réalisés avec une technologie particulière va choisir d'utiliser un générateur qui se rapproche le plus à la technologie utilisée par ses composants. Ainsi, plus il y a de générateurs sur différentes plates-formes, plus il y a le choix. Une fois qu'une plate-forme est précisée pour réaliser ou intégrer le générateur, le développeur peut soit l'étendre si elle est ouverte, soit se baser dessus pour construire le générateur en utilisant ses interfaces offertes. Par exemple, les générateurs existant RMI et CORBA sont construits au-dessus de TCP/IP qui est fermé. Dans la suite de ce travail (section 3.4.1), nous allons montrer comment nous avons réalisé les générateurs pour le connecteur d'équilibrage de charge sur la plate-forme ouverte Jonathan [64].

Ainsi, le générateur construit l'abstraction de communication définie par le connecteur sur une plate-forme indépendamment de toute application. C'est un moyen qui permet à l'architecte :

- De ne s'intéresser au niveau architecture qu'à la sémantique de communication en utilisant le connecteur pour assembler ses composants ;
- De ne pas se limiter aux sémantiques de communication simples offertes par les plates-formes pour décrire les communications complexes en les intégrant dans le code des composants. Désormais c'est la plate-forme qui s'adapte à l'application ;
- De réfléchir conceptuellement à l'intégration de la sémantique de communication en minimisant la conjoncture actuelle d'utiliser les contraintes technologiques à des fins conceptuelles. Il n'est plus obligatoire de se contenter des seuls connecteurs offerts par les plates-formes, il devient à présent possible d'en créer de nouveaux.

L'architecte garde à l'esprit qu'il existe des générateurs qui offrent une sémantique complexe au-dessus d'une plate-forme et épargne les développeurs d'application de s'en préoccuper. Il faut donc identifier de nouveaux connecteurs et les réaliser comme des générateurs. Par exemple, auparavant les communications se faisaient directement sur l'API *socket* des réseaux TCP/IP qui est une plate-forme de bas niveau. Ensuite, il y a eu une abstraction et sont apparues les plates-formes CORBA, RMI et SOAP. Ces abstractions ne

¹⁰Il serait judicieux de choisir les technologies courantes et les plus utilisées.

Connecteur	Equilibrage de charge non adaptatif	Equilibrage de charge adaptatif
Protocole de coordination	choix 1 parmi n sans considération de charge	choix 1 parmi n avec considération de charge
Protocoles de mise en œuvre (générateurs)	- Round Robin - Random	- Le moins chargé - Charge moyenne
Plates-formes	- CORBA - RMI	-CORBA -RMI

Tableau 2.2 — Exemples de générateurs pour les connecteurs d'équilibrage de charge

peuvent pas tout garantir, il faut alors abstraire plus, comme nous avons fait avec l'équilibrage de charge.

À ce niveau du cycle de vie, le générateur offre une base commune générique, à qui sera infligé un traitement spécifique en lui attribuant les interfaces de connexion que les prises sont prêtes à accueillir comme des paramètres dans la phase suivante. Le générateur permet d'appliquer la sémantique du connecteur sur les interfaces spécifiées à la connexion au-dessus d'une plate-forme. Le générateur agit comme un compilateur, il vérifie la compatibilité des interfaces des composants avec ses prises. Le générateur est plus qu'un élément de génération, c'est un élément qui permet la vérification de la compatibilité, au sens du typage, des interfaces des composants reçues en paramètre afin que la connexion puisse être définie entre ces composants.

Exemple

Nous avons réalisé deux des connecteurs que nous avons décrits dans la phase d'identification et de description : le connecteur d'équilibrage de charge avec une politique adaptative et le connecteur d'équilibrage de charge avec une politique non adaptative. Nous les prenons comme exemple ici et nous détaillerons les implémentations dans le chapitre suivant.

Dans le cas du connecteur d'équilibrage de charge non adaptatif, le connecteur spécifie que les requêtes du client doivent être équilibrées entre les serveurs sans imposer de conditions sur leur charge de travail. C'est un connecteur de choix. Son protocole de coordination décrit que ce connecteur doit élire le serveur qui doit traiter les requêtes du client. Après le processus de raffinement de la phase de conception, il découle du protocole de coordination plusieurs protocoles de mise en œuvre qui correspondent aux politiques de répartition de charge. Parmi celles-ci nous pouvons citer les politiques *Round Robin* et aléatoire. Chacun des générateurs qui réalise un protocole de mise en œuvre représente un générateur différent dans la famille des générateurs du connecteur d'équilibrage de charge non adaptatif. Ils satisfont tous à la spécification du connecteur mais diffèrent dans le protocole de mise en œuvre. Ils peuvent également différer dans la plate-forme de mise en œuvre. Le même raffinement est applicable au connecteur d'équilibrage de charge adaptatif, qui est différent du premier car les prises possèdent une propriété en plus : elles doivent surveiller la charge des serveurs. Le tableau 2.2 résume ce raffinement.

Un autre exemple est le connecteur de consensus qui fait partie des connecteurs de fusion. Ce connecteur fusionne toutes les réponses des serveurs en une seule réponse pour ne la transmettre au client que si toutes les réponses sont équivalentes. Le consensus représente le protocole de coordination du connecteur. Le raffinement de ce dernier consiste à choisir un des algorithmes de consensus pour l'appliquer. L'algorithme choisi représente le protocole de

mise en œuvre qui sera réalisé au-dessus de la plate-forme de mise en œuvre cible choisie.

2.3.4 Génération et déploiement

Les deux phases d'assemblage (section 2.3.2) et d'implémentation (section 2.3.3), sont indépendantes l'une de l'autre. L'architecte ne doit pas attendre le concepteur pour accomplir ses tâches de choisir et d'assembler les composants avec les connecteurs, et le concepteur ne doit pas attendre l'architecte pour accomplir sa tâche de concevoir puis implémenter le générateur. Par contre, la phase que nous traitons dans cette section suppose que les deux phases précédentes ont été achevées. Elle constitue la phase finale dans le cycle de vie du connecteur.

Après avoir choisi le connecteur qui définit la sémantique de communication pour constituer la configuration de l'application, l'architecte passe la main au développeur de l'application qui se charge de réaliser la connexion physique. Le développeur de l'application doit chercher les éléments binaires exécutables correspondants aux éléments de l'architecture afin de réaliser l'assemblage effectif et déployer l'application. Il aura à utiliser les composants exécutables, des composants développés et compilés, prêts à être assemblés puis déployés. Ceci n'est pas vrai pour le connecteur. Ce dernier est réalisé comme un générateur qui ne représente pas les exécutables de l'abstraction de communication. Les exécutables de la connexion sont obtenus après le processus de génération en fournissant les interfaces de connexion au générateur. Ce processus de génération fabrique le composant de liaison qui, lui, constitue les exécutables de la communication (figure 2.20). Le générateur s'occupe ainsi d'automatiser l'intégration de la propriété de communication pour des composants interagissant au-dessus d'une plate-forme.

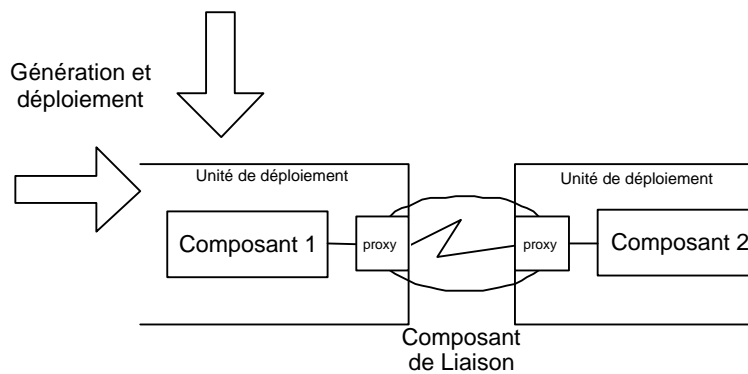


Figure 2.20 — Phases génération et déploiement

Le composant de liaison est la représentation physique de la sémantique de communication. Celle-ci est appliquée à un ensemble de composants au-dessus d'une plate-forme. De cette réification de la sémantique de communication résulte un composant de communication. En effet, le générateur crée un composant qui offre des interfaces explicites, avec lesquelles les composants interagissant communiquent effectivement. Lorsque le générateur qui répond au mieux aux exigences de l'application est choisi, le développeur lui fournit comme paramètres les interfaces de connexion pour lesquels il faut appliquer la sémantique de communication. L'activation du processus de génération transforme les prises génériques du générateur en proxies. Ces derniers sont les répliques des interfaces requises et offertes des composants interagissant adoptées par les prises. Ils augmentent ces interfaces avec la propriété de com-

munication et des détails techniques liés à la plate-forme. Les proxies constituent désormais les interfaces explicites du composant de liaison. Le connecteur compte sur les interfaces des composants interagissant pour fabriquer les siennes. Le diagramme d'états transitions présenté en figure 2.21 récapitule les états des transformations que subit une prise durant tout le cycle de vie. Dans un premier temps, la prise évolue séparément de deux manières différentes pour aboutir à deux entités qui répondent aux contraintes définies par la prise. D'une part elle adopte les interfaces des composants pour former les interfaces de connexion, et d'autre part elle est augmentée par les détails de la plate-forme. Les deux entités résultantes se rejoignent à la fin (jonction) pour devenir un proxy.

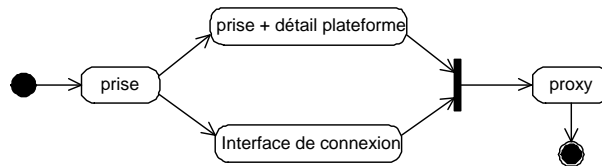


Figure 2.21 — Cycle de vie de la prise

Le proxy reprend ainsi les informations techniques ajoutées lors de l'implémentation sur la plate-forme ainsi que les signatures des interfaces de connexion. Le proxy implémente ces interfaces de connexion qui sont identiques aux interfaces des composants interagissant. Il ne les implémente pas effectivement, ce sont les composants offrant les services qui le font réellement. Il garde les mêmes signatures et il les implémente de manière à leur appliquer la propriété de communication avec transparence. Ainsi à ce niveau, la propriété de communication reste transparente aux composants mais le composant de liaison ne l'est plus puisque les composants communiquent directement avec lui. C'est une connexion physique car le composant de liaison est réellement relié aux composants pour remplir sa fonction de connexion. La vision structurelle est décrite dans le diagramme de classe de la figure 2.22.

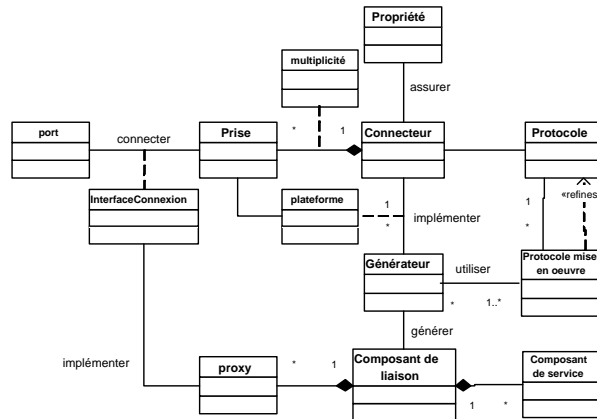


Figure 2.22 — Description du composant de liaison

Après le processus de génération et l'obtention du composant de liaison, l'application se retrouve finalement composée d'un ensemble de composants prêts à être déployés. En effet, le connecteur est transformé en un composant, il peut donc être déployé. Le composant de liaison est une entité virtuelle, il n'est pas représenté explicitement. Il n'est pas matérialisé par une seule entité, il englobe un ensemble d'éléments prêts à être déployés. Il englobe un

certain nombre de composants de service qui peuvent être des librairies, des composants, des modules, un middleware, ou le réseau. Ce composant de liaison, avec les entités qui le composent, doit garantir le transfert de données (transmission de messages), maintenir la liaison et assurer la propriété de communication. Ces entités du composant de liaison peuvent être déployées sur plusieurs sites. À ce niveau alors, on ne peut pas pointer une entité et dire que c'est le connecteur, l'entité connecteur disparaît pour se retrouver un peu dispersée dans plusieurs entités qui sont différentes des composants interagissant.

Le déploiement correspond à la phase de développement qui décrit la configuration du système en cours d'exécution dans un environnement effectif [80]. Il englobe les activités requises pour l'installation ou la mise à jour d'un système logiciel après son développement et sa mise à disposition. Pour le déploiement, des décisions concernant les paramètres de configuration, les performances, l'allocation des ressources, la distribution et la concurrence doivent être prises. Les résultats de cette phase sont capturés dans des fichiers de configuration ainsi que dans la vue de déploiement. Ainsi, dans cette phase, chaque entité du composant de liaison doit être installée à un endroit bien déterminé. Dans les applications distribuées, les proxies sont installés chacun du côté des composants avec qui ils sont destinés à communiquer (cf. figure 2.20) pour former l'unité de déploiement. Dans la terminologie UML, chaque nœud contiendra un artefact lié au connecteur. Ces artefacts sont les proxies qui implémentent l'interface de connexion. Cet artefact du connecteur sera celui qui interagit réellement avec l'artefact du composant et cette communication est locale.

Exemple

Imaginons une application avec un client et un serveur, où le serveur offre une opération *m*. En utilisant un connecteur RPC et un générateur RMI, après la génération, le stub généré offrira la même opération *m* du serveur. Ainsi, lorsque le client envoie une requête le stub intercepte la requête et la transmet au serveur qui implémente réellement cette opération. Les exigences des prises du connecteur sont que les données qui y transitent doivent être sérialisables. À l'exécution ce sont les proxies du composant de liaison qui doivent assurer les fonctions de codage et décodage de l'information (le *marshalling* et le *unmarshalling*). Chaque proxy est installé dans la même unité de déploiement que le composant avec qui il doit interagir. Ainsi, les communications sont locales.

Dans le cadre de notre exemple d'équilibrage de charge, on possède un proxy qui, en plus de réaliser le codage et décodage des informations pour assurer le transfert des données, applique une politique d'équilibrage de charge qui permet de sélectionner le serveur le moins chargé pour répondre à la requête du client.

2.4 Synthèse

Nous avons donné dans ce chapitre notre point de vue sur la notion de *connecteur*. Nous avons commencé par aborder les différents problèmes liés à la considération actuelle des connexions entre les composants. Nous avons montré que les notations actuelles pour représenter les connexions ainsi que les concepts utilisés n'étaient pas appropriés. Nous avons montré également le besoin de définir de nouvelles abstractions de communication accompagnées de leurs implémentations. Ensuite, nous avons posé notre définition du connecteur et de son cycle de vie. Finalement, nous avons décrit en détail les différentes phases que traverse une interaction entre les composants, à savoir : l'identification et la description, la sélection

et l'assemblage, la conception et l'implémentation, et la génération et le déploiement.

Nous définissons un *connecteur* comme la réification d'une abstraction d'interaction, de communication ou de coordination. C'est une entité d'architecture potentielle, non déployable ni compilable, qui assure une propriété ou une intention de communication. Elle existe pour servir les besoins en communication des composants interagissant tout en assurant leur transparence. Nous avons également défini un vocabulaire qui permet de désigner le connecteur dans différentes phases de son cycle de vie afin de ne pas référencer par ce mot tout ce qui est lié à l'interaction. Un connecteur est implémenté comme une famille de générateurs. Il est utilisé pour être assemblé avec des composants pour former une connexion. La fusion d'une connexion avec un générateur donne un composant de liaison. La figure 2.23 récapitule le cycle de vie du connecteur ainsi que les différentes phases parcourues.

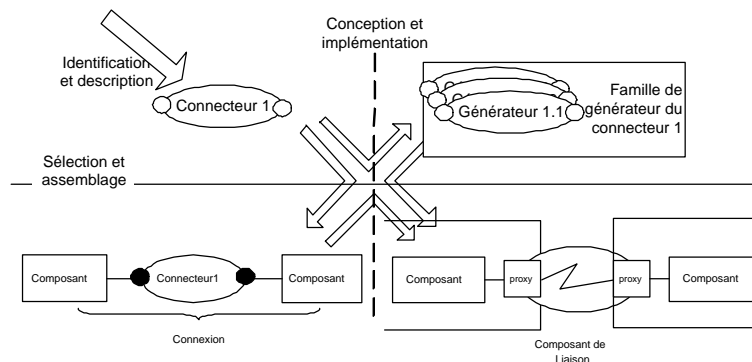


Figure 2.23 — Les phases du connecteur

Cette définition et ce cycle de vie donnent une meilleure compréhension du connecteur en distinguant les entités. Elle assure la séparation des responsabilités, le raffinement et la transparence. L'originalité de notre approche se manifeste par la possibilité de réaliser et de maintenir l'abstraction de communication jusqu'au déploiement par le biais du générateur. Ce qui fait sa force c'est l'indépendance et la dissociation de la description de l'abstraction de communication de son implémentation — et de la technologie qui la réalise — avec la distinction entre connecteur et générateur d'un côté ; et la séparation des responsabilités avec la distinction entre le connecteur et la connexion d'un autre côté.

La transformation tardive du connecteur en un composant possède un grand avantage : elle permet de garder la transparence des composants interagissant à la propriété de communication, et donc la séparation des responsabilités, le plus longtemps possible. Ainsi, la conception des composants logiciels se fait en fonction des services requis et fournis des autres composants et le connecteur n'intervient qu'en dernier lieu pour assurer les propriétés de communication. Vouloir changer un connecteur pour cause de changement de la propriété de connexion ou de la sémantique de communication n'affecte en rien la conception des composants. Cette approche offre ainsi une meilleure flexibilité et réutilisabilité des composants et des abstractions de communication. De ce fait, on gagne en abstraction, on ne se soucie pas du changement de l'intention dans l'application car ceci n'affecte pas les composants à connecter.

Nous nous sommes concentrés dans la suite sur la conception et la réalisation de différents générateurs pour un connecteur d'équilibrage de charge dont nous décrivons la mise en œuvre dans le chapitre suivant.

Nous avons présenté dans le chapitre précédent notre définition du connecteur ainsi qu'une méthodologie pour sa conception et son utilisation (son cycle de vie).

Dans ce chapitre nous allons illustrer ce concept de connecteur sur un exemple d'équilibrage de charge. Nous commençons par donner les principales caractéristiques de l'équilibrage de charge. Nous présenterons ensuite la plate-forme sur laquelle nous avons réalisé le générateur et nous décrivons cette réalisation. Enfin, pour illustrer la viabilité de notre approche, nous présenterons quelques résultats de tests de performance.

3.1 Description de l'équilibrage de charge

3.1.1 Description générale

L'expansion des technologies Internet durant cette dernière décennie a engendré l'accroissement d'applications et de services en ligne. Les applications distribuées comme les applications de e-commerce ou les systèmes de commerce courant en ligne (*Trading*) se sont multipliées. Elles impliquent des milliers de clients qui accèdent simultanément à des serveurs Internet. Cette forte demande provoque la surcharge des serveurs, ce qui engendre une lenteur des opérations de réponse aux requêtes. Ceci provoque un besoin croissant de développement de nouvelles solutions afin de rendre le système scalable en augmentant la capacité d'extension des applications. En effet, la présence d'un seul serveur ralentit le traitement des requêtes des clients existants et rend difficile, voir impossible, l'arrivée de nouveaux clients. Une solution efficace pour améliorer le temps de réponse et gérer cette grande demande consiste à augmenter le nombre de serveurs offrant le même service (des copies) et d'appliquer une technique d'*équilibrage de charge* entre les serveurs.

L'équilibrage de charge est une technique bien adaptée pour l'utilisation des ressources de calcul disponibles. C'est une technique qui consiste à répartir, aussi équitablement que possible, la charge de travail entre plusieurs serveurs afin d'améliorer la vitesse de réponse globale du système et sa performance. L'équilibrage de charge permet d'exploiter pleinement toutes les ressources d'un système réparti. Il existe des mécanismes logiciels/matériels qui permettent de déterminer quel serveur devrait exécuter chacune des requêtes des clients. Ces mécanismes varient suivant les stratégies de distribution de charge et les niveaux de leur mise en œuvre.

3.1.2 Caractéristiques de la propriété d'équilibrage de Charge

L'équilibrage de charge permet d'utiliser les ressources disponibles avec plus d'efficacité. Il existe différentes stratégies [40] d'équilibrage de charge qui peuvent être réalisées à différents niveaux d'abstraction. Ces stratégies dépendent à la fois de la politique d'équilibrage de charge et du niveau de mise en œuvre.

Politiques d'équilibrage de Charge

Il existe plusieurs techniques pour répartir équitablement la charge de travail sur les différentes ressources de calcul. Ces techniques se répartissent en deux catégories de politiques suivant que les conditions de charge de travail des serveurs à l'exécution soient prises en compte ou non. Ainsi, on distingue les politiques non adaptatives des politiques adaptatives. Contrairement à la politique non adaptative, la politique adaptative mesure la charge des serveurs et utilise le résultat de cette mesure pour le choix du serveur qui traitera les requêtes. Chacune des deux catégories possède plusieurs algorithmes. Voici quelques détails sur chacune d'entre elles :

Politiques non adaptatives : C'est un ensemble de politiques qui ne prennent pas en compte la charge de travail des serveurs pour équilibrer la charge. Elles sont généralement utilisées lorsque les requête génèrent des quantités de charge identiques ou uniformes.

On peut citer comme exemple d'algorithmes non adaptatifs la politique du Round Robin et la politique aléatoire. La première permet d'appliquer un algorithme simple : les serveurs participants à l'application s'inscrivent initialement dans une liste. Cette liste est parcourue séquentiellement de telle sorte qu'un serveur suivant est sélectionné à chaque nouvelle requête du client. La seconde politique permet de choisir aléatoirement, dans cette même liste, un serveur qui traitera chaque requête du client.

Politiques adaptatives : C'est un ensemble de politiques qui permettent de prendre des décisions de répartition des requêtes en fonction de la charge de travail des serveurs à l'exécution, et « s'adaptent » dynamiquement aux changements de cette charge. Elles peuvent prendre des informations disponibles sur chaque serveur pour sélectionner le serveur qui traitera la requête. L'intérêt de ces politiques est qu'elles offrent le moyen de s'appuyer sur des métriques d'équilibrage de charge très diverses. Elles peuvent considérer des métriques physiques, comme la charge d'usage ou la puissance du CPU, et le pourcentage d'utilisation de la mémoire ; ou logiques, comme le nombre de requêtes émises par seconde ou le contenu même des requêtes. Ces politiques peuvent être utilisées lorsque la charge générée par chaque requête n'est pas prédictible à l'avance et n'est pas uniforme. Elles nécessitent des algorithmes plus avancés et plus évolués pour éviter à un ou plusieurs serveurs d'être surchargés pendant que d'autres serveurs restent sous utilisés.

On peut citer deux exemples de stratégies d'équilibrage de charge adaptatives : la politique du serveur le moins chargé et celle de la charge minimum. Dans ces deux cas la distribution des requêtes s'adapte dynamiquement aux conditions de changement de charge de travail des serveurs. La politique du moins chargé permet à des serveurs de recevoir des requêtes jusqu'à ce qu'une valeur de seuil soit atteinte. Une fois le seuil atteint, les requêtes suivantes sont transférées à un autre serveur avec une charge de travail plus faible. La politique de la charge minimum calcule la moyenne des charges de travail sur les serveurs. Si la charge de travail sur un serveur particulier est supérieure à la charge de travail moyenne des serveurs et plus grande que la charge de travail du

serveur le moins chargé, par un pourcentage de seuil de migration, toutes les requêtes suivantes seront transférées au serveur le moins chargé.

Niveaux de mise en œuvre de l'équilibrage de charge

L'équilibrage de charge est une technique connue et efficace qui peut aider à satisfaire le besoin grandissant d'avoir des applications distribuées possédant une grande capacité d'extension (*scalability*). De plus, c'est une technique qui peut améliorer la performance de serveurs interconnectés par un réseau. Les stratégies que nous avons décrites ci-dessus peuvent être réalisées à différents niveaux des couches protocolaires : réseau, système d'exploitation, intergiciel et application. Nous allons passer en revue comment l'équilibrage de charge peut être réalisé sur ces différents niveaux :

Niveau réseau : L'équilibrage de charge à ce niveau [43, 22] est fourni par les routeurs IP et les serveurs de nom de domaine (*Domain Name System* - DNS) [14] qui servent un groupe de machines hôtes. Par exemple, lorsqu'un client résout le nom d'un hôte (d'une machine), le DNS peut assigner une adresse IP différente dynamiquement à chaque requête suivant les conditions de charge actuelles. Le client contacte ensuite le serveur désigné, ignorant qu'un serveur différent sera sélectionné pour sa prochaine résolution DNS. Les routeurs peuvent aussi être utilisés pour relier un flot (*flow*) TCP à n'importe quel serveur suivant les conditions de charge actuelles, puis utiliser cette liaison pour la durée du flot.

L'équilibrage de charge qui s'appuie sur les infrastructures réseaux est souvent utilisé par les sites web à gros volume et à grande échelle. Ils utilisent souvent l'équilibrage de charge au niveau de la couche *réseaux* et au niveau de la couche *transport*. Pour appliquer l'équilibrage de charge, ces couches utilisent respectivement l'adresse IP et le port pour déterminer où transmettre les paquets.

Niveau système d'exploitation (OS) : Ce type d'équilibrage de charge est fourni par des systèmes d'exploitation distribués à travers des mécanismes de *clustering* (groupage), de répartition de charge¹ et la migration de processus [24, 29]. Le clustering constitue un moyen efficace pour accomplir la haute disponibilité et la haute performance. Il s'agit de combiner un grand nombre d'ordinateurs afin d'améliorer la puissance de traitement du système entier. Les processus peuvent ainsi être distribués avec transparence parmi les ordinateurs dans le cluster (groupe). En général, les clusters emploient conjointement des techniques de répartition de charge et de migration de processus pour équilibrer la charge entre les processeurs — ou plus généralement les nœuds du réseau. L'utilisation de la migration de processus consiste à transférer ou déplacer l'état d'un processus surchargé s'exécutant sur un nœud vers les nœuds moins chargés. Cette opération de transfert d'état du processus requiert un support d'infrastructure de plate-forme important pour manipuler les différences de plates-formes entre les nœuds.

Niveau Intergiciel (Middleware) : À ce niveau, certaines implémentations d'intergiciel [46, 41, 42] intègrent la fonctionnalité d'équilibrage de charge au niveau du bus d'objet (*Object Request Broker* - ORB) lui-même, tandis que d'autres [70, 8] l'implémentent à un niveau plus haut comme le niveau des services de l'intergiciel. Une technique courante pour la réalisation de l'équilibrage de charge à l'intérieur de l'ORB est l'extension du service de nommage qui peut être parcouru pour appliquer une politique d'équilibrage

¹La répartition de charge ne doit pas être confondue avec l'équilibrage de charge. En effet, les ressources de traitement peuvent être réparties parmi des processus pas nécessairement équilibrés.

de charge non adaptative comme le Round Robin. Les autres intergiciels définissent une entité explicite appelée « équilibreur de charge » (*load balancer*) qui est spécialement dédiée à l'exécution de la politique d'équilibrage de charge.

L'équilibrage de charge au niveau intergiciel fournit une flexibilité significative en termes d'influence sur comment un service d'équilibrage prend ses décisions. Ceci est faisable notamment en définissant des métriques relatives à l'application et la possibilité de prendre en compte le contenu des requêtes. L'équilibrage de charge à base de middleware peut être utilisé conjointement avec l'équilibrage de charge à base de réseau et à base d'OS cités ci-dessus pour avoir des solutions efficaces. Il constitue une passerelle pour personnaliser la nature de l'équilibrage de charge.

Niveau application : Réaliser l'équilibrage de charge au niveau application implique la prise en compte de la politique d'équilibrage de charge par les applications. Ceci signifie qu'à ce niveau, les mécanismes d'équilibrage de charge sont mélangés avec le code des composants. C'est le composant client lui-même qui décide quel serveur devra traiter ses requêtes suivant des métriques propres à l'application.

Ce niveau offre un grand choix et une grande flexibilité pour le choix de la métrique et de la politique d'équilibrage. Il constitue le meilleur moyen pour prendre des décisions d'équilibrage de charge en fonction du contenu des requêtes. Cependant, il est complètement intégré dans l'application, les composants contiennent ainsi du code superflu qui est indépendant de leur fonctionnalité.

Il existe ainsi une multitude de façons et de niveaux pour la réalisation de cette propriété d'équilibrage de charge. Néanmoins, chacun des niveaux de réalisation cités ci-dessus présente un certain nombre d'inconvénients.

La réalisation de l'équilibrage de charge aux niveaux OS et réseaux est transparente à l'application mais manque de flexibilité et peut ne pas être appropriée à certains types d'applications distribuées. En effet, en l'appliquant à ces niveaux, il n'est pas possible d'utiliser des métriques qui sont liées à l'application lors de prises de décisions sur l'équilibrage de charge. De plus, ces niveaux manquent de contrôle sur les politiques et sont incapables de prendre en considération le contenu des requêtes lors de la distribution de la charge de travail. La principale métrique prise en compte au niveau réseau est la fréquence de réception des requêtes [43], et la prise de décision sur l'équilibrage de charge est effectuée uniquement en fonction de la destination des requêtes. De même, il n'est pas possible de prendre en compte le contenu des requêtes pour effectuer la migration des processus au niveau système d'exploitation. Dans ce dernier cas l'équilibrage de charge est de gros grain ; c'est le serveur lui-même qui est soumis à l'équilibrage plutôt que le composant ou l'objet résidant sur le serveur.

Appliqué sur les deux derniers niveaux, application et intergiciel, l'équilibrage de charge offre plus de possibilités pour le choix des métriques qui peuvent être liées à l'application. Il est aussi possible de prendre en compte le contenu des requêtes. En effet, en le réalisant au niveau application il constitue la meilleure solution pour distribuer la charge équitablement entre les serveurs en fonction des métriques liées à l'application ou au contenu des requêtes. Cependant, en l'appliquant à ce niveau une grande partie de la politique d'équilibrage est intégrée dans les composants. Il n'y a pas de transparence et les composants contiennent du code indépendant de leur fonctionnalité. Il est bien avantageux de pouvoir prévoir cette propriété au niveau architecture de l'application, car ceci peut aider pour sa maintenance et son évolution, mais malheureusement son implémentation est mêlée à celle des composants.

La réalisation de l'équilibrage de charge au niveau intergiciel semble être la solution idéale : elle offre plus de choix des métriques liées à l'application tout en étant transparente à l'application, ou presque². En revanche, les solutions qui existent sont des solutions dispersées et non structurées d'une part, et sont intégrées dans le processus de développement comme des solutions de dernière minute d'autre part. Les solutions actuelles manquent de flexibilité pour le changement de politique car chaque intergiciel assure une seule politique et il est difficile de changer d'intergiciel s'il y a besoin de changer de politique d'équilibrage de charge. La décision de faire de l'équilibrage de charge est prise tardivement dans le processus de développement, par le développeur et non pas par l'architecte de l'application. Ceci implique la possibilité d'apporter des modifications aux composants de cette application. En effet, la propriété n'apparaît pas dans l'architecture de l'application et ceci crée une divergence entre l'architecture de l'application et son implémentation. Il est alors difficile d'avoir un suivi de l'application pour sa maintenance et son évolution.

3.2 Connecteur d'équilibrage de charge

Dans les trois premiers niveaux de réalisation de la propriété d'équilibrage de charge que nous avons vu dans la section précédente ; à savoir les niveaux réseau, système d'exploitation et intergiciel ; l'intégration de cette propriété ne peut être décidée que tardivement dans le processus de développement, lors du développement des composants ou à l'installation. Dans le dernier niveau, le niveau application, l'intégration de la propriété d'équilibrage de charge est décidée au niveau architecture de l'application, cependant sa réalisation est mêlée à celle des composants et n'est pas transparente. Cette propriété d'équilibrage de charge est considérée dans notre terminologie comme complexe impliquant plusieurs participants, elle fait intervenir plusieurs clients et plusieurs serveurs. Nous proposons de la réifier en un connecteur et de la capitaliser dans un générateur au-dessus d'un intergiciel.

Nous décrivons dans cette section ce connecteur d'équilibrage de charge ainsi que les principales exigences pour la réalisation d'un bon équilibrage de charge.

3.2.1 Description du connecteur d'équilibrage de charge

Généralement, évoquer cette propriété d'équilibrage équivaut à évoquer des techniques de placement, des algorithmes et politiques d'équilibrage de charge, etc. Cette propriété est rarement évoquée indépendamment d'une réalisation, en tant qu'une abstraction. Ceci explique sa réalisation plus sur les couches protocolaires basses que sur les niveaux application et intergiciel. En effet, nous avons cherché une spécification de l'équilibrage de charge mais tout ce que nous avons obtenu ce sont des descriptions d'algorithmes où le principal objectif est d'améliorer les techniques de placement et de migration de processus [21, 85]. Il n'existe pas à notre connaissance de spécification standardisée. Par exemple, CORBA ne spécifie pas un service d'équilibrage de charge. Les constructeurs d'ORB ont la responsabilité d'utiliser des mécanismes offerts par GIOP (*General Inter-ORB Protocole*), le protocole standard de l'OMG, pour assurer la re-direction nécessaire pour l'équilibrage de charge [65]. Chaque constructeur d'ORB prend ses propres décisions d'implémentation pour cette propriété ce qui fait que son intégration dans l'application ne se fait qu'à un moment tardif, lors du choix

²Les solutions d'équilibrage de charge au niveau intergiciel utilisées actuellement n'offrent pas une transparence totale au niveau du serveur. Dans le cas des techniques d'équilibrage de charge adaptables par exemple, c'est le serveur qui doit s'occuper de faire connaître sa charge.

de l'ORB. De plus, il est libre à chaque constructeur de choisir une technique différente pour l'implémentation de l'équilibrage de charge (comme un service explicite, ou bien intégré dans l'ORB combiné avec un service de nommage, etc). À partir de là, il résulte des solutions divergentes et dispersées. De plus, aucune capitalisation du savoir-faire n'est possible.

Nous proposons donc de réifier la propriété d'équilibrage de charge comme un connecteur qui doit mettre en œuvre cette intention de communication. Nous la considérons comme une entité à part entière autonome et abstraite, et nous la réalisons sous la forme d'un générateur. De cette façon, nous détachons la propriété des détails algorithmiques et d'implémentation qui seront traités dans les phases de conception et d'implémentation. Ceci nous permet de remonter cette propriété au niveau architecture et de pouvoir garder sa trace jusqu'à l'installation, et ce afin d'assurer dans de bonnes conditions les tâches de réutilisation, de maintenance et d'évolution du logiciel. La propriété du connecteur étant décrite, il reste à décrire les deux autres blocs de construction de ce connecteur, c'est-à-dire ses prises et son protocole de coordination.

Étant dans un contexte d'applications à base de composants, le connecteur d'équilibrage de charge existe pour augmenter les performances et réduire le temps de réponse entre des composants qui émettent des requêtes (clients) et des composants qui reçoivent et traitent ses requêtes (serveurs). Ainsi, le connecteur définit deux prises différentes. La première prise possède un statut de « client » qui récupère les requêtes émises par les composants client. La seconde prise possède un statut « serveur » qui renvoie les requêtes vers les composants serveur choisis. Ce connecteur est à utiliser dans des environnements répartis ou distribués pour utiliser le maximum de ressources disponibles. Il faut alors poser des contraintes sur les prises : elles doivent accepter des données qui sont sérialisables afin que celles-ci puissent être transférées dans cet environnement réparti. Il peut y avoir plusieurs composants client reliés à plusieurs composants serveur, les multiplicités des prises sont alors m et n respectivement pour la prise client et la prise serveur.

Le protocole de coordination décrit comment les prises interagissent entre elles. Dans le cas de l'équilibrage de charge, il s'agit de choisir parmi les n serveurs celui qui va traiter les requêtes du client. C'est un protocole de type 1 parmi n . Ce choix de serveur s'effectue en fonction ou indépendamment de la charge des serveurs suivant que la stratégie utilisée est adaptative ou non. Ainsi, il résulte d'un début de processus de raffinement deux protocoles de coordination différents correspondants aux deux stratégies. On obtient alors deux connecteurs d'équilibrage différents avec chacune des deux stratégies puisque la stratégie adaptative ajoute une contrainte supplémentaire sur les prises. En effet, comme cette stratégie doit mesurer la charge de travail des serveurs pour équilibrer la charge, cette responsabilité est assignée à la prise « serveur » qui s'occupe de récupérer la charge de travail du serveur et assurer la transparence. Ainsi, ce n'est pas au composant serveur de fournir sa charge au connecteur. Les deux connecteurs ne posent aucune contrainte sur les types de données, leurs prises sont génériques. Le premier protocole de coordination décrit seulement que les requêtes des clients doivent être distribuées parmi les serveurs présents. Le second protocole de coordination décrit que les requêtes des clients doivent être distribuées sur les serveurs en fonction de la charge de ces derniers.

3.2.2 Exigences d'un service d'équilibrage de charge

Après la description d'un connecteur et de sa propriété, la phase qui suit consiste en la conception de ce connecteur et sa réalisation comme un générateur. Comme nous l'avons constaté dans la section 3.1.2, bien que la réalisation du service d'équilibrage de charge au

niveau intergiciel comporte quelques inconvénients, elle constitue un bon compromis entre la transparence de la propriété et la prise en compte de métriques de charge liées à l'application. Nous réalisons les connecteurs comme des générateurs au-dessus d'un intergiciel qui sert de plate-forme de mise en œuvre.

Afin de raffiner les contraintes pour la réalisation des générateurs des connecteurs, nous avons repris des exigences exprimées par Othman *et al.* dans [70] pour un *service* d'équilibrage de charge CORBA. Dans ce qui suit, nous allons décrire ces exigences pour voir comment nous les satisfaisons et les améliorons :

Granularité : comme le service d'équilibrage de charge est spécialement conçu pour CORBA, la granularité de l'équilibrage de charge est fixée à l'objet car il y constitue l'unité d'abstraction. Considérer l'objet comme l'unité d'équilibrage de charge affine la granularité par rapport aux approches considérant une granularité relative à la station de travail entière ou au serveur, cependant elle reste réductrice.

Avec les prises génériques du connecteur qui ne posent pas de condition sur le type de données qui y transitent, nous offrons un plus large choix sur la nature des unités à équilibrer.

Transparence : un service d'équilibrage de charge doit être aussi transparent que possible du côté du client comme du côté du serveur. En effet, le service d'équilibrage de charge développé par les auteurs est transparent du côté du client puisqu'ils utilisent un mécanisme de transmission de requêtes³ offert par CORBA qui permet de rediriger les requêtes avec transparence. Cependant, cette transparence n'est pas totalement assurée du côté du serveur puisque le serveur transmet sa charge explicitement au service d'équilibrage de charge. De plus, le service d'équilibrage de charge contacte le serveur explicitement pour lui demander de continuer d'accepter les requêtes ou de les refuser lorsque les conditions de charge changent.

Dynamicité des requêtes : un service d'équilibrage de charge doit pouvoir prendre en compte les requêtes du client qui arrivent avec un mode déterministe, par exemple des requêtes qui arrivent dans des périodes de temps fixes ; ou un mode non déterministe, c'est-à-dire des requêtes dynamiques dont la durée d'exécution ne peut être connue à l'avance. Ceci à pour objectif de pouvoir assurer les politiques adaptatives et non adaptatives. Nous supportons cette idée et nous avons effectivement identifié deux connecteurs afin d'assurer ces propriétés.

Augmenter la dépendance au système : cette caractéristique est introduite pour justifier la prise en charge des pannes par le système. Elle est légitime car il faut exploiter les capacités du système sous-jacent. Cependant, le service d'équilibrage de charge doit avoir une part de responsabilité dans la gestion des pannes. Il doit pouvoir gérer les pannes de serveurs (*crash*) afin de transférer les requêtes destinées à ce serveur vers un autre serveur en suivant les conditions de charge. Cette caractéristique est assurée par le connecteur d'équilibrage de charge.

Les auteurs définissent également trois autres exigences auxquelles nous adhérons et que nous satisfaisons parfaitement comme nous le verrons dans la section 3.5 :

- Reconfiguration : il faut pouvoir ajouter et supprimer des serveurs sans perturber le système ;

³Utilisant l'exception *LOCATION_FORWARD*.

- Surcharge minimum : il ne faut pas introduire une latence anormale ou la surcharge du réseau qui pourrait réduire au lieu d'augmenter la performance ;
- Support des métriques et politiques d'équilibrage de charge liées à l'application.

En satisfaisant ces exigences, ce *service* d'équilibrage de charge apporte bonne solution pour réaliser cette propriété. Cependant, il constitue une solution particulière pour réaliser cette propriété et il n'est pas assez générique. Il est conçu pour une plate-forme particulière, CORBA, et utilise plusieurs techniques liées à la plate-forme afin d'assurer les exigences décrites, comme la transparence, qui ne sont pas forcément disponibles dans d'autres plates-formes. Par exemple, le mécanisme de transmission de requêtes de CORBA n'est pas disponible sur RMI, ce qui rend ce service dépendant des plates-formes et non réutilisable avec des composants utilisant d'autres plates-formes. Ainsi, notre proposition d'abstraction de cette propriété de communication comme un *connecteur* permet de la considérer indépendamment de toute plate-forme. Sa réalisation avec la technique de génération permet d'utiliser n'importe quelle plate-forme sous-jacente puisque le générateur est en mesure d'utiliser les interfaces offertes par cette plate-forme. On peut ainsi choisir le générateur adéquat suivant sa disponibilité sur la plate-forme qu'utilisent les composants interagissant.

Dans la suite, nous allons illustrer notre solution de réalisation du connecteur d'équilibrage de charge. Cependant, avant d'entamer la description de la mise en œuvre du générateur pour ce connecteur, nous allons présenter la plate-forme que nous avons utilisée à cet effet : la plate-forme Jonathan [64].

3.3 Présentation de Jonathan

Les connecteurs d'équilibrage de charge sont destinés à répartir la charge équitablement entre des composants distants dans le cadre d'applications distribuées (il ne s'agit pas d'équilibrer la charge entre processus d'une même machine). Nous avons choisi le niveau intergiciel pour la réalisation des protocoles de mise en œuvre, et la plate-forme Jonathan pour l'implémentation des générateurs de code associés aux connecteurs.

Jonathan est une plate-forme d'exécution répartie (*Distributed Processing Environment - DPE*) créée par France Télécom R&D (groupe ObjectWeb) lors du projet européen ACTS RETINA. C'est une plate-forme de programmation répartie ouverte et libre⁴. En effet, contrairement aux plates-formes standards, comme CORBA et RMI, une partie de son implémentation peut être spécialisée et redéfinie pour satisfaire de nouvelles contraintes de qualité de service, comme la mise en place d'un système d'authentification ou la garantie d'intégrité sur des réseaux longue distance. En partant de la problématique liée aux invocations distantes, Jonathan utilise différents concepts dont ceux introduits par RM-ODP (*Reference Model of Open Distributed Processing*) [13, 76] dans le but de fournir une architecture générique pour les bus logiciels. Il est possible, grâce à cette architecture, de créer des bus logiciels conformes aux spécifications CORBA ou Java RMI, ou d'en créer de nouveaux.

3.3.1 Principaux concepts de Jonathan

Pour relier les objets distants, Jonathan utilise le concept des objets de liaison qui réifient la liaison informatique. Il définit le concept de personnalité pour désigner un nouveau bus logiciel.

⁴Jonathan est disponible et librement téléchargeable sur le site <http://www.objectweb.org>

Objet de liaison

Introduit par le modèle RM-ODP, un objet de liaison a pour rôle d'assurer la transmission des invocations entre objets distants et le retour d'un résultat si nécessaire. L'encapsulation de la liaison dans un objet permet d'abstraire des détails techniques de la sémantique de communication (par exemple, type de liaison utilisée, paramètres de la liaison) et de créer de nouveaux types de liaison sans avoir à modifier les applications réparties. Toutefois, les objets de liaison restent des abstractions et ne sont pas explicitement représentés : les objets de liaison ne sont pas des entités de plein droit dans Jonathan. Ce concept est matérialisé par un ensemble de modules reliés entre eux, qui doivent garantir la communication, maintenir la liaison et permettre la transmission des messages. RM-ODP est principalement conçu pour assurer ces trois derniers objectifs, il rend transparentes les tâches de gestion de la répartition et de l'hétérogénéité.

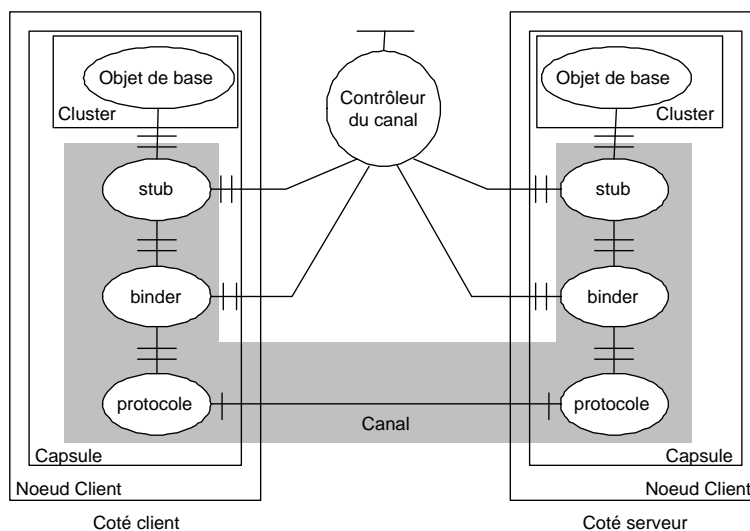


Figure 3.1 — Liaison client/serveur avec un objet de liaison

La figure 3.1 schématise l'objet de liaison qui constitue une abstraction du canal. Celui-ci apparaît en grisé et nous pouvons remarquer les différents modules qui le composent. Il est composé d'un ensemble de *stub*, de *binder*, et d'objets protocoles. Le *stub* est chargé d'interpréter les interactions et de réaliser les transformations et vérification nécessaires à cette interprétation. Le *binder* maintient la liaison entre objets distribués. Enfin, l'objet protocole communique avec d'autres objets protocoles du même canal pour permettre les interactions entre les objets de base. Néanmoins, pour obtenir un bus logiciel fonctionnel, il est nécessaire de spécialiser chacun de ces modules. Pour cela, Jonathan apporte la notion de personnalité.

Personnalité

Jonathan permet de créer des bus logiciels comme CORBA ou RMI. Il fournit un *framework* générique qui, une fois complété, donne un bus logiciel fonctionnel. L'ensemble des modules et outils ajoutés, qui viennent se greffer au-dessus, constitue la personnalité. Une

personnalité définit un système de type, un système de type de liaison⁵, une API (*Application Programming Interface*) et des règles de programmation. La personnalité est unique à chaque type de bus logiciel et elle permet d'assurer l'intégrité des communications. Le framework Jonathan fournit deux personnalités :

- David est une personnalité qui permet d'obtenir un bus logiciel conforme aux spécifications CORBA-IIOP. Il intègre le protocole d'interaction IIOP (*Internet Inter-ORB Protocol*) basé sur les protocoles GIOP (*General Inter-ORB Protocol*) et TCP/IP. Il comporte également le protocole de présentation CDR (*Common Data Representation*) qui permet de transporter les données sur un flot d'octets. David fournit tout un paquetage de classes et d'interfaces CORBA, un compilateur IDL (*Interface Definition Language*) pour générer les *stub* et les *skeleton*, et un serveur de noms ;
- Jeremie est une personnalité qui permet d'obtenir un bus logiciel conforme à Java RMI. Il intègre le protocole JIOP (*Jeremie Inter-ORB Protocol*) qui utilise le protocole IIOP et un protocole de présentation spécifique. Jeremie fournit une extension du paquetage RMI, un compilateur RMI qui crée des *stub*, et un serveur de noms.

3.3.2 Architecture de Jonathan

Organisation de l'API Jonathan

Afin d'aider les développeurs à modifier le comportement de Jonathan et à développer de nouvelles personnalités, Jonathan offre des interfaces (API) et des modules qui sont destinés à être spécialisés. Ces modules sont :

Noyau : il possède une vocation de configuration, similaire aux variables d'environnement de unix et windows.

Marshaller : il permet de transformer les données, voir des objets, en « paquet ». Le unmarshaller réalise l'opération inverse. Chaque personnalité doit créer sa propre composante.

Stub : le *stub* est le représentant local d'un objet distant et c'est une partie de la liaison entre les client et le serveur.

Protocole : il représente des protocoles comme TCP/IP ou GIOP.

Jonathan est également basé sur 4 frameworks : de liaison, de communication, de configuration, et de resource. Les modifications que nous avons apportées ne concernent que le premier, le framwork de liaison.

Fonctionnement de Jonathan

La communication entre objets distants se fait à travers les différents éléments qui constituent l'objet de liaison (*stub*, *binder*, protocole). Tout d'abord l'objet distribué doit être diffusé, ensuite l'application cliente doit récupérer la référence de l'objet distant pour pouvoir communiquer. La communication entre l'objet distribué et l'application cliente se déroule en 4 étapes :

⁵Le type de liaison représente les protocoles utilisés par la liaison.

1. La première étape se déroule du côté de l'objet distribué. Une fois celui-ci initialisé, un représentant local (stub serveur) est instancié, servant d'interface entre l'objet distribué et la pile de protocoles. Puis, lorsque l'objet distribué est enregistré dans un éventuel serveur de nom, la pile de protocoles est initialisée pour se mettre en attente d'une connexion avec un client.
2. La seconde étape concerne le client, il doit récupérer la référence de l'objet distant. Pour cela, il se connecte au serveur de nom, puis initialise sa pile de protocoles et instancie un stub client. À ce moment, le client et l'objet distant peuvent communiquer.
3. Lorsque le client envoie une invocation à l'objet distant, celle-ci est interceptée localement par le stub. Ce dernier la conserve dans un marshaller qui parcourt la pile de protocoles. Arrivé en bas de la pile, le message est transformé en paquets et est envoyé sur le réseau. Du côté de l'objet distribué, le message est récupéré dans un unmarshaller qui remonte la pile de protocoles jusqu'à trouver une session de requête (stub serveur). Cette session extrait le message du unmarshaller et l'envoie localement à l'objet destinataire.
4. La réponse est interceptée par le stub serveur et parcourt le chemin inverse jusqu'au client qui récupère la réponse.

Les étapes 1 et 2 constituent l'initialisation de la liaison entre le client et le serveur. Les étapes 3 et 4 concernent la communication entre le client et le serveur. Elles peuvent être répétées indéfiniment. La figure 3.2 illustre cette communication.

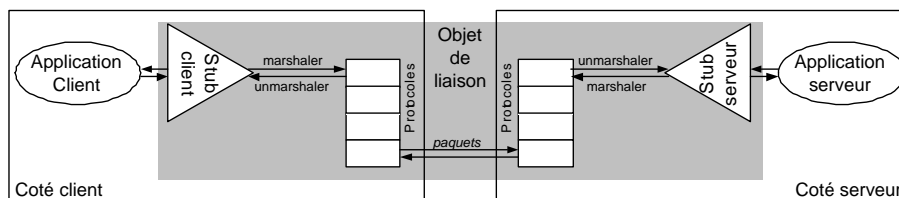


Figure 3.2 — Schéma d'une invocation entre une application cliente et un objet distribué dans Jonathan

3.3.3 Etude de Jeremie : la personnalité RMI de Jonathan

Pour la réalisation des générateurs du connecteur d'équilibrage de charge, nous nous sommes basés Jeremie, la personnalité RMI de Jonathan. Nous avons étendu cette personnalité sachant que cette plate-forme ne possède pas un service d'équilibrage de charge comme CORBA. Nous présentons dans cette section quelques spécificités de cette personnalité ainsi qu'un exemple de réalisation de la propriété d'équilibrage de charge avant l'extension de Jeremie avec les nouveaux générateurs.

Spécificités de Jeremie

Jeremie est un bus logiciel similaire à Java RMI. Son API fournit un ensemble de classes, d'interfaces et de programmes Java pour développer des applications Java réparties à la manière de Java RMI 1.2. Elle réutilise des interfaces comme `java.rmi.Remote`, ou des classes abstraites comme `java.rmi.server.RemoteServer`; redéfinit des classes existantes; apporte ses propres classes et interfaces; et fournit des programmes de génération de stub et de serveur de noms.

Le fonctionnement de Jeremie est conforme à la description de Jonathan à quelques exceptions près. Indépendamment de JDK (*Java Development Kit* 1.1 ou plus), le générateur de stub de Jeremie — appelé `JRMICompiler` — est un programme proche de `rmic` du JDK 1.2. Il ne produit que des stub et pas de skeleton car Jeremie se sert d'un code générique utilisant le paquetage `java.lang.reflect` pour accéder et interagir avec la bonne méthode de l'objet distant utilisé. Le stub client possède une référence sur l'objet distant, créée à la connexion avec ce dernier. Cette référence s'occupe de transmettre les messages à la pile de protocoles et de récupérer les réponses (c'est un intermédiaire entre le stub et la pile de protocole).

Fonctionnement de Jeremie avec la propriété d'équilibrage de charge avant l'intégration du connecteur

Pour illustrer les invocations à distance avec Jeremie, nous allons utiliser une application client/serveur avec la propriété d'équilibrage de charge réalisée au niveau application. Ainsi, il est possible d'utiliser des métriques de calcul de charge liée à l'application et d'agir sur les stratégies d'équilibrage de charge employées. Pour représenter cette application, nous utilisons la notation *boxes, lines-and-ellipses* comme illustré dans la figure 3.3 (a).

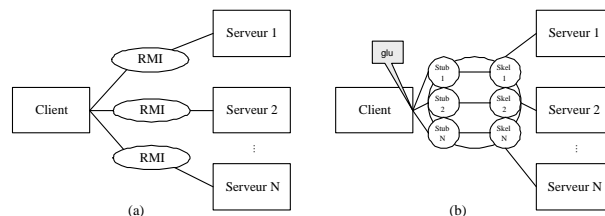


Figure 3.3 — Connexion complexe avec un connecteur RMI

L'utilisation de cette représentation au niveau architecture explicite le sens de la connexion. La figure 3.3 (a) représente une application répartie qui implique un client et plusieurs serveurs distants. Les connexions sont point à point et synchrones, réalisées comme des invocations de méthodes (ce sont les propriétés du RMI). La figure 3.3 (b) reflète l'architecture, après la compilation, c'est à dire à l'exécution. La communication se fait en plusieurs étapes :

- Du côté du client, pour répartir ses requêtes avec cette propriété d'équilibrage de charge, le client doit :
 - prendre connaissance de *tous* les serveurs participants à l'application. Ces serveurs sont explicitement connus dans son code,
 - il commence par résoudre tous les noms⁶ des serveurs avec une opération `lookup()`,
 - il télécharge les stubs générés du coté des serveurs. Cette procédure de téléchargement est répétée pour toutes les copies de serveurs,
 - il exécute la politique d'équilibrage de charge implémentée dans son propre code et adresse les requêtes aux stubs⁷ locaux qui les transfèrent aux serveurs.
- Du côté des serveurs, ces derniers doivent faire connaître leur charge au client — dans le cas d'une politique d'équilibrage de charge adaptative. Il n'y a pas de skeleton, c'est une session requête qui s'occupe de retrouver les bons objets.

⁶Dans RMI, à chaque objet est associé un nom dans un registre.

⁷Les stubs contiennent les références des objets qui, dans ce cas, sont manipulées par le client.

Comme décrit ci-dessus, Jeremie s'occupe de réaliser les invocations distantes afin de cacher les détails de la distribution. Ainsi, en l'utilisant tel qu'il existe actuellement les opérations liées à l'équilibrage de charge, comme l'application de la politique et la mesure de la charge, sont réalisées par les composants car Jeremie n'est pas adapté pour le faire.

La réalisation décrite ci-dessus est la conséquence de l'absence d'une abstraction liée à l'équilibrage de charge au niveau architecture. Les architectes utilisent alors les moyens disponibles pour décrire leurs applications, c'est-à-dire des connecteurs simples point-à-point. L'introduction de cette propriété se fait à un moment tardif par le développeur. Disposant de moyens de communication simples, ce dernier introduit la propriété dans les composants, une solution de dernière minute. De ce fait, il existe dans les programmes de glu qui est indépendante de leur fonctionnalité (figure 3.3 (b)). L'application résultante sera complètement différente de son architecture et les programmes deviennent difficiles à comprendre, à maintenir et à faire évoluer. De plus, le mécanisme d'équilibrage de charge conçu ne servira qu'à une seule application. Il n'est pas possible de le réutiliser malgré qu'il soit répétitif.

Nous confirmons ainsi que pour prendre en compte la propriété d'équilibrage de charge avec des métriques liées à l'application, les éléments de connexion actuels ne sont pas appropriés. Nous allons voir dans la section qui suit comment nous étendons Jeremie pour y introduire la propriété d'équilibrage de charge.

Jonathan offre la possibilité de modifier les générateurs de stub, c'est ce que nous avons utilisé pour réaliser notre générateur d'équilibrage de charge. Jonathan offre un objet de liaison ouvert que nous avons étendu pour obtenir notre composant de liaison. Il est à noter que nous n'avons pas comme objectif de créer une nouvelle personnalité à Jonathan puisque nous ne faisons qu'étendre Jeremie en lui introduisant la propriété d'équilibrage de charge. Nous n'avons pas touché aux types de liaison (les protocoles) et aux processus de transmission de données. Nous avons juste déporté l'intelligence qui doit se faire avant cette transmission du code du client vers le stub. Pour cela, nous avons utilisé deux méthodes pour réaliser deux générateurs. Dans le premier cas nous avons injecté de l'intelligence de la politique d'équilibrage de charge dans les proxies. Dans le second cas, nous avons utilisé, dans ces mêmes proxies, les interfaces d'un composant qui s'occupe de réaliser le protocole de mise en œuvre du connecteur.

3.4 Mise en œuvre du connecteur d'équilibrage de charge

Pour montrer la faisabilité de nos concepts et évaluer tout le cycle de vie du connecteur, nous avons réalisé les deux connecteurs d'équilibrage de charge, adaptatif et non adaptatif, décrits dans la section 3.2.1. Comme proposé précédemment, les deux connecteurs sont réalisés comme des générateurs au-dessus de la plate-forme Jonthan. Nous décrivons dans cette section le passage aux phases de conception et d'implémentation.

Dans la phase de conception, il résulte du processus de raffinement du connecteur non adaptatif deux protocoles de mise en œuvre. Le premier concerne la réalisation d'un générateur avec une politique Round Robin, le second concerne la réalisation d'un générateur avec une politique d'équilibrage de charge aléatoire. Ces politiques n'affectent en rien les propriétés du connecteur, seul l'algorithme interne change. Les deux générateurs sont implémentés et sont conformes à la description du connecteur, il n'y a pas besoin de récupérer la charge des serveurs.

Nous rappelons qu'à l'issue du processus de raffinement du connecteur d'équilibrage de charge adaptatif on obtient, là aussi, deux protocoles de mise en œuvre. Le premier avec une politique d'équilibrage de charge du serveur le moins chargé et le second avec la politique de la charge minimum. Ils sont aussi conformes à la description du connecteur. Dans ce cas, la prise en compte de la charge est nécessaire pour appliquer la politique.

Nous allons décrire dans la suite de cette section la réalisation de deux de ces générateurs : le générateur avec la politique d'équilibrage de charge Round Robin et le générateur avec la politique d'équilibrage de charge du serveur le moins chargé. Ces générateurs sont réalisés sur la plate-forme Jonathan avec la personnalité Jeremie.

3.4.1 Réalisation des générateurs

La personnalité Jeremie de Jonathan offre un générateur pour générer les stubs de l'objet de liaison. Il s'agit ici de l'implémentation du connecteur RPC. Ce générateur gère la partie communications distantes à laquelle nous avons rajouté la partie équilibrage de charge avec les deux politiques citées. Les générateurs que nous avons réalisés reposent sur une extension et une modification du générateur existant. Ils s'occupent de générer des composants de liaison dont les interfaces, c'est-à-dire les proxies, s'occupent de quelques propriétés des politiques d'équilibrage de charge.

Ainsi, avant les opérations de marshalling, le stub doit s'occuper de réaliser la politique d'équilibrage de charge pour choisir la référence du serveur qui traitera la requête. Une fois le choix de la référence effectué, le marshaller poursuit sa fonction pour transformer les données et les transmettre aux différents protocoles puis vers le réseau.

Réalisation du connecteur d'équilibrage de charge non adaptatif : politique Round Robin

Le premier générateur que nous avons réalisé applique la politique Round Robin. Pour cette politique non adaptative, il n'est pas nécessaire de surveiller la charge de travail des serveurs pour équilibrer la charge. À la description, le connecteur ne posait pas de contraintes particulières concernant les prises à part qu'un client qui se connecte à la prise client doit envoyer des requêtes et le serveur connecté à la prise serveur doit y répondre. Il n'y avait pas de contraintes concernant la charge des serveurs.

Pour la réalisation de ce générateur, la modification de la plate-forme a concerné uniquement le mécanisme de génération du proxy client. Ce dernier réalise un certain nombre d'opérations avant d'entamer ses tâches initiales. Il doit parcourir une liste dans laquelle les serveurs qui participent à l'interaction se seraient déjà inscrits. En la parcourant, il envoie chaque nouvelle requête au serveur suivant le dernier serveur utilisé dans la liste. En reprenant le schéma explicatif du fonctionnement de Jonathan en section 3.3.2 (page 73), la figure 3.4 montre le niveau de notre intervention dans le fonctionnement normal de Jeremie. Ainsi, avant d'envoyer la référence du serveur dans le marshaller, le stub fait un calcul pour prendre la référence du serveur qui suit le serveur dernièrement utilisé dans la liste des serveurs enregistrés.

Pour concrétiser cette liste, nous avons utilisé le même principe du registre de Jeremie le *JRMIRegistry* que nous avons étendu et que nous avons appelé *JRMIRegistryLB*⁸. Ce

⁸LB pour *Load Balancing*.

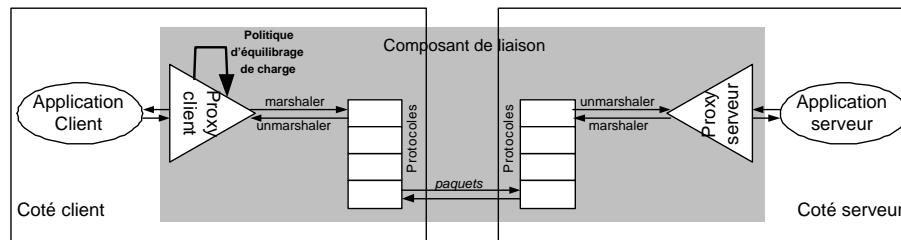


Figure 3.4 — Niveau d'intervention sur Jeremie pour la réalisation du générateur d'équilibrage de charge non adaptatif

registre agit comme un annuaire qui regroupe tous les serveurs participants. Il doit fournir les référence des serveurs au proxy côté client pour que ce dernier puisse appliquer la politique.

Le diagramme de séquence illustré dans la figure 3.5 représente l'enchaînement des événements à l'exécution entre les composants de l'application client/serveur et le composant de liaison obtenu après génération.

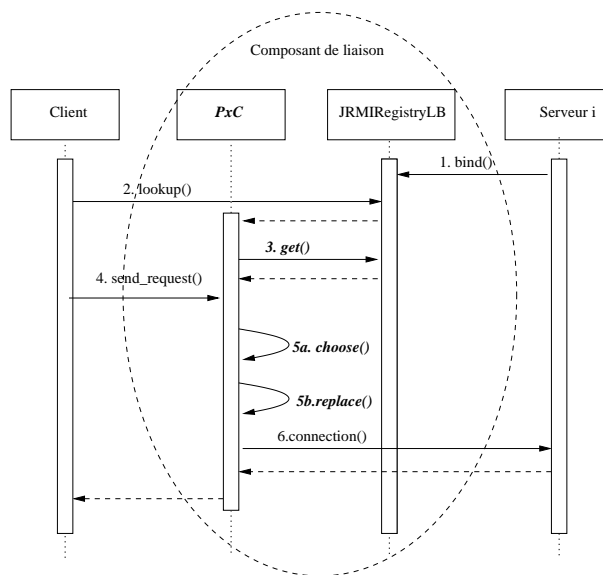


Figure 3.5 — Diagramme de séquence de la politique d'équilibrage de charge Round Robin

1. Les serveurs S_1, \dots, S_n commencent par s'enregistrer normalement auprès du registre *JRMIRegistryLB* avec une opération *bind()* ;
2. Le client récupère la référence d'un seul serveur en résolvant un nom par une opération *lookup()*, ce qui provoque le téléchargement du proxy *PxC* généré par le générateur. Initialement, le composant client n'établit le contacte qu'avec un seul serveur, il ne sait pas que n serveurs se cachent derrière cette communication contrairement au cas de la section 3.3.3 ;
3. Le proxy *PxC* généré contient la logique de la politique d'équilibrage de charge. Il doit prendre connaissance de tous les serveurs participants et enregistrés dans le registre *JRMIRegistryLB* avec l'opération *get()*. Ce n'est plus au client de s'occuper de cette tâche ;

4. Lorsque le client appelle le service offert par l'un des serveurs, il utilise le nom du service offert par le serveur. Il envoie sa requête localement à un seul proxy représentant tous les composants serveurs, comme si les serveurs n'étaient pas distants ;
5. Le proxy intercepte la requête et applique la politique du Round Robin :
 - a) Il commence par choisir le serveur suivant dans la liste pour servir la requête,
 - b) il remplace ensuite la référence actuelle par la référence du serveur nouvellement choisie ;
6. Le proxy reprend ses tâches initiales de codage et de décodage des paramètres, d'envoi vers et réception du réseau, et la réception du résultat pour le client.

Ainsi, contrairement à la réalisation de l'équilibrage de charge au niveau application, dans le cas de l'équilibrage de charge au niveau intergiciel le code de la politique d'équilibrage de charge est injecté dans le proxy généré (*PxC*) et il n'y a aucune référence à l'équilibrage de charge au niveau du code des composants client et serveur. Nous utilisons la même logique que lorsqu'il est fait dans le client mais au lieu de la mélanger avec le code du client, elle est injectée dans le proxy. Le proxy assure la même tâche qu'auparavant mais la référence est constamment modifiée. La politique est réalisée localement auprès de chaque client. Cette méthode n'améliore pas la performance par rapport à l'autre solution mais elle assure la séparation des responsabilités. Elle fournit ainsi du code plus propre et réutilisable.

Les proxies implémentent l'interface du serveur à leur manière. Dans ce cas, ils l'implémentent pour garantir le transfert et appliquer la sémantique de communication qui est la politique de l'équilibrage de charge. Pour ce faire, nous sommes intervenus sur le framework de liaison de Jonathan que nous avons étendu en enrichissant le compilateur.

Réalisation du connecteur d'équilibrage de charge adaptatif : politique du serveur le moins chargé

Le second générateur que nous avons réalisé applique la politique d'équilibrage de charge du serveur le moins chargé. Pour cette politique adaptative, il est nécessaire de calculer la charge des serveurs pour pouvoir décider de celui qui traitera les requêtes des clients. Ainsi, conformément à la spécification du connecteur, la prise du côté des serveurs doit accomplir la fonction de calculer, surveiller et récupérer la charge de travail des serveurs.

Pour réaliser ce générateur et concrétiser la fonction de la prise, nous avons apporté deux modifications au fonctionnement normal de Jeremie :

- D'un côté, nous avons réintroduit la notion de proxy côté serveur. Il s'agit de l'équivalent du skeleton dans RMI qui, rappelons le, était absent dans Jeremie. Le générateur génère ainsi un proxy du côté du serveur qui doit récupérer la charge quelle que soit la métrique utilisée. Pour cela, le registre du nouveau générateur, qui fait partie du composant de liaison généré, possède une structure différente. Il possède un champ en plus qui stocke la charge de chaque serveur inscrit ;
- D'un autre côté, pour pouvoir garder le contact et changer de serveur avec transparence pour le client, le proxy client n'est pas aussi simple que le précédent. Le proxy client utilise des processus (*threads*) Java pour surveiller le changement de charge des serveurs et pouvoir rediriger ses requêtes au serveur le moins chargé.

La figure 3.6 montre le niveau d'intervention et les modifications de l'actuel Jeremie pour la réalisation du générateur du connecteur d'équilibrage de charge adaptatif avec la politique du serveur le moins chargé.

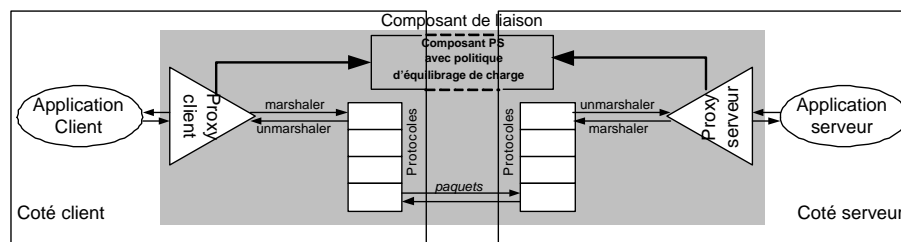


Figure 3.6 — Niveau d'intervention sur Jeremie pour la réalisation du générateur d'équilibrage de charge adaptatif

La réalisation de ce générateur s'appuie sur un composant logiciel (de communication) qui possède des interfaces explicites. Il s'agit d'un composant conforme au modèle *publish/subscribe* (ou fournisseur/consommateur) que nous appelons : le composant PS (*PS-Component*⁹). Ce composant regroupe les threads et le registre, utilisés pour les fonctions de surveillance de charge et de récupération de la bonne référence. Le composant de liaison sera désormais composé de ce composant plus les proxies client et serveur générés. Le composant est construit au-dessus de TCP/IP puisqu'il utilise des *sockets* d'écoute.

Le générateur utilise des classes de Jeremie pour manipuler et instancier des classes qui composent le composant de liaison. Nous allons à présent expliquer le rôle du composant et le rôle des proxies générés :

a) **Le composant PS** : ce composant offre des services explicites qui permettent aux serveurs de publier leur charge à l'aide du service *publish()* ; et aux clients de s'enregistrer auprès de ce composant pour récupérer la référence du serveur le moins chargé en cas de changement de charge à l'aide du service *subscribe()*. Il est responsable de l'application de la politique et de la réalisation des calculs. En fait, les clients et les serveurs n'interagissent pas directement avec ce composant, chacun d'eux possède un représentant pour réaliser cette tâche, ce sont les proxies. Le composant PS repose sur deux classes. La première classe, **LBRegistry**, implémente les services *publish()* et *subscribe()*. De plus, cette classe implémente les services de l'interface de Jeremie **Registry** avec ses méthodes *bind()* et *lookup()*. Elle gère le registre et offre également des fonctionnalités pour gérer les références des serveurs inscrits en fonction de leur charge. Le registre de ce générateur définit une structure de données différente et plus complexe que celle du registre de Jeremie. Il possède un champ en plus relatif à la charge des serveurs. Chaque serveur qui s'enregistre doit fournir avec lui sa charge et avoir le moyen d'annoncer son changement de charge ainsi que la nouvelle valeur quand nécessaire. Donc, en plus de mettre en correspondance chaque nom avec la référence de l'objet, il fait correspondre à chaque référence la charge du serveur. La seconde classe nommée **ThreadLBRegistry** sert d'interface avec les représentants des serveurs et des clients (les proxies). Elle permet de générer les threads qui communiquent avec les proxies client et serveur.

b) **Les proxies** : le générateur génère deux types de proxies qui correspondent aux deux prises du connecteur. Ces proxies permettent de cacher les détails de la communication distante entre le client et le serveur mais possèdent quelques fonctionnalités supplémentaires :

- Le proxy du côté client implémente les mêmes interfaces offertes que le serveur, ce sont les interfaces adoptées lors de la connexion entre les composants et le connecteur.

⁹PS Pour Publish/Subscribe

Ce proxy généré implémente les interfaces de façon à transformer les données pour les envoyer à travers le réseau d'interconnexion, et à envoyer les données au serveur approprié. À la différence du proxy généré par le générateur non adaptatif, ce proxy doit prendre en compte la charge des serveurs pour sélectionner le serveur qui traitera les requêtes. Pour cela, ce proxy crée un *thread* qui s'inscrit auprès du composant PS (à travers la classe `ThreadLBRegistry`). Dès qu'il y a un changement de charge, le composant PS communique la bonne référence au thread qui la retransmet au proxy pour remplacer la référence courante. Donc le thread a un rôle d'écoute. Le proxy établit la connexion et envoie les requêtes au même serveur tant que le thread ne lui a pas envoyé une nouvelle référence pour remplacer la référence courante.

- Le proxy du côté du serveur représente le client qui envoie les requêtes. D'un côté, ce proxy récupère la requête envoyée par le proxy du côté du client, la transforme (unmarshalling) et l'envoie au serveur pour que ce dernier la traite comme si la requête vient d'un objet local. D'un autre côté, il doit calculer la charge du serveur et la transmettre au composant PS pour le mettre à jour. Pour cela, il crée aussi un thread qui s'occupe d'envoyer la charge au composant PS (via la classe `ThreadLBRegistry`). Dans l'exemple qui suit, la métrique utilisée est le nombre de requêtes reçues. Ainsi, à chaque fois que le proxy reçoit une requête, le thread augmente le nombre. Le thread envoie le nombre de requêtes reçues périodiquement au composant PS comme indicatif de la charge. Ceci est réalisé uniquement si la charge a subi une variation minimale de 10%.

Ainsi ce générateur joue le rôle d'un adaptateur. Les proxies utilisent les interfaces d'un composant qui réalise la politique d'équilibrage de charge plutôt que ce soit le client et le serveur qui utilisent directement ces interfaces. La figure 3.7 montre un exemple d'utilisation du composant de liaison généré à l'exécution :

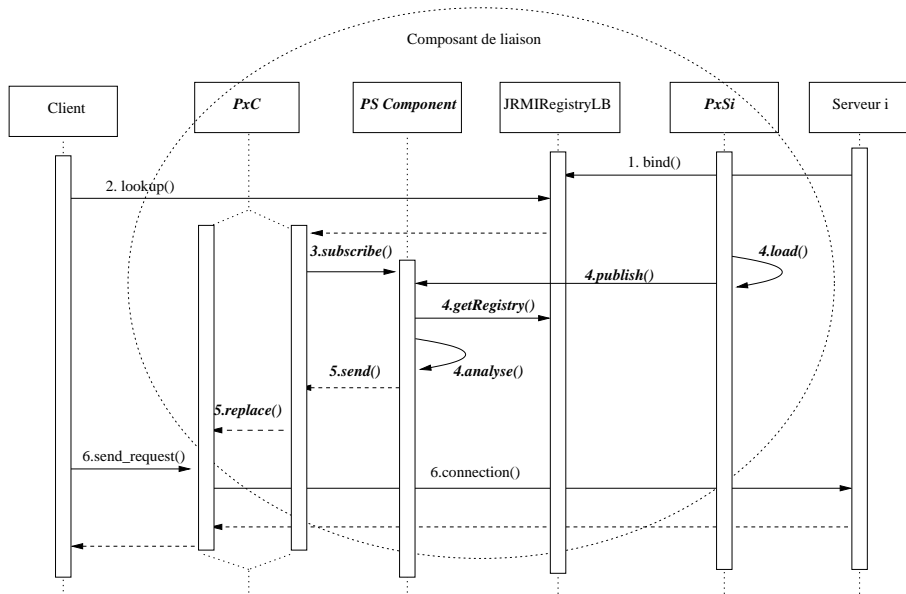


Figure 3.7 — Diagramme de séquence pour la politique d'équilibrage de charge du serveur le moins chargé

1. Les serveurs commencent par s'inscrire auprès du nouveau registre ;
2. Le client récupère la référence d'un seul serveur en résolvant un nom par une opération

lookup() ce qui permet de rapporter son proxy ;

3. Le proxy (*PxC*) lance un thread qui s'enregistre auprès du composant PS pour recevoir la bonne référence du serveur qui doit traiter les requêtes du client ;
4. De l'autre côté, les proxies serveur (*PxSi*) mesurent la charge et l'envoient au composant PS qui doit calculer la charge minimum après avoir pris connaissance de tous les serveurs enregistrés dans le registre avec leurs charge ;
5. Le composant PS renvoie au thread client la référence du serveur offrant la charge minimum, et le proxy client l'utilise pour remplacer la référence courante ;
6. À la réception d'une requête du client, le proxy poursuit ses tâches de transformation de données, d'établissement de la connexion avec le bon serveur pour le transfert des données.

Contrairement à la politique du Round Robin, la politique du serveur le moins chargé est une politique qui change de serveur à la demande. Au lieu que chaque requête de client soit servie par un serveur différent, le client continue d'envoyer ses requêtes au même serveur jusqu'au changement de décision du composant PS, lorsqu'il détecte une variation dans la charge des serveurs. Il existe une connexion permanente entre le composant PS et le serveur. Lorsqu'un serveur se déconnecte proprement ou subitement, le composant PS détecte cette anomalie. Il retire la référence de ce serveur du registre et réévalue les charges des autres serveurs pour rediriger les requêtes du client connecté vers d'autres serveurs en suivant le même calcul de charge.

Les interfaces du composant PS sont utilisées par les proxies et non par les composants eux-mêmes. L'avantage de cette approche est qu'elle assure aux composants la transparence à la communication. De plus, elle offre la flexibilité que nous souhaitons. Ainsi, si l'architecte désire changer de sémantique de communication, il ne doit remplacer que le connecteur sans avoir à toucher aux composants. Cette substitution ne nécessite pas de re-concevoir ou réécrire les composants fonctionnels puisque ces derniers ne possèdent que du code lié à leur fonctionnalité et ne possèdent pas de code lié à la communication. Le passage d'un connecteur non adaptatif à un autre connecteur adaptatif au niveau architecture se fait sans contraintes puisque les prises sont responsables de récupérer la charge des serveurs avec transparence.

3.4.2 Nouvelle architecture de l'application

Nous avons décrit les connecteurs d'équilibrage de charge dans la section 3.2.1 et nous avons décrit la réalisation des générateurs associés dans la section précédente. Maintenant que l'abstraction et sa réalisation existent, il est possible de les utiliser dans notre application.

Reprenons l'exemple de la section 3.3.3 où un client a besoin d'un service offert par un serveur. Dans cet exemple, nous avons utilisé une combinaison de connecteurs RPC où la politique d'équilibrage de charge est réalisée dans le code des composants clients et serveurs. Puisque le connecteur d'équilibrage de charge est spécifié, il est préférable de l'utiliser dans l'architecture de notre application. La figure 3.8 (a) illustre l'architecture de l'application en utilisant le connecteur d'équilibrage de charge. L'ensemble des connecteurs RPC est remplacé par le connecteur d'équilibrage de charge.

Dans cette architecture (figure 3.8 (a)) la sémantique de communication est plus claire. Cette représentation indique qu'il est de la responsabilité du connecteur de réaliser l'équilibrage de charge et les appels distants. Les composants et le connecteur sont développés indépendamment les uns des autres, ils sont ainsi réutilisables indépendamment les uns des

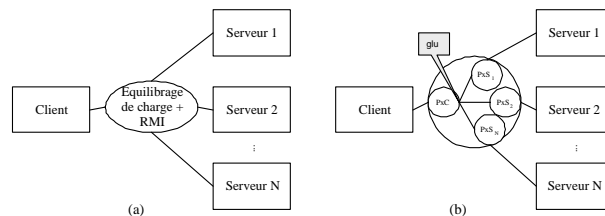


Figure 3.8 — Une application qui utilise un connecteur d'équilibrage de charge

autres. Un architecte choisit les composants de son application, décide de la sémantique de la communication entre ces composants en choisissant un connecteur, puis assemble le tout. Une fois le générateur correspondant au connecteur sélectionné, il est utilisé comme suit :

```
# JRMICompilerLB ApplicationInterface
```

Cette ligne de commande appelle le générateur par son nom (`JRMICompiler`) et lui fournit en paramètre les interfaces de connexion (`ApplicationInterface`). On obtient après ce processus de génération le composant de liaison qui est utilisé comme déjà décrit dans la figure 3.5 (page 77).

La figure 3.8 (b) montre que l'abstraction est préservée jusqu'à la fin puisque à l'exécution le client s'adresse à un seul proxy qui est la concrétisation de la prise à l'exécution et qui représente l'ensemble des serveurs. Cette solution nous permet de réaliser les mécanismes d'équilibrage de charge comme au niveau application pour plus de souplesse dans le choix des métriques, mais sans toucher aux codes du client et des serveurs. Ainsi, l'architecture peut servir de référence tout au long du cycle de vie.

Il est à noter que le générateur de Jonathan peut toujours être utilisé pour réaliser des connexions point à point RMI. De ce fait, l'introduction de l'équilibrage de charge, ou le changement de la stratégie d'équilibrage de charge ne demande qu'une simple re-génération de proxies à partir de la même description d'interfaces.

3.5 Tests et résultats

Nous avons utilisé les générateurs que nous avons développés pour faire une série de tests et des calculs de performance. Le but n'est pas de faire des comparaisons avec d'autres techniques d'équilibrage de charge mais d'illustrer la viabilité de l'approche en montrant la faisabilité de ces générateurs, et les améliorations obtenues en appliquant notre approche. Nous décrivons dans cette section la plate-forme matérielle de test utilisée ainsi que les tests et résultats obtenus.

3.5.1 Description du matériel et de la plate-forme de test

Plate-forme matérielle/logicielle

Nous avons réalisé des tests sur 32 machines équipées du système d'exploitation *Fedora Core* avec 1 à 8 serveurs et 1 à 22 clients. Une machine a été réservée pour le registre et une autre pour le contrôle du *benchmark* (jeu d'essai). Les machines serveurs et la machine

registre possédait les caractéristiques matérielles suivantes : des processeurs Intel Pentium IV, 3 GHz, et 512 MB de RAM. Les machines clientes, quant à elles, avaient la configuration suivante : 9 machines avec des processeurs Intel Pentium IV, 2.6 GHz et 512 MB de RAM ; 9 machines avec des processeurs Intel Pentium IV, 2GHz et 256 MB de RAM ; et 4 machines avec des processeurs Intel Pentium III, 600 MHz et 256 MB de RAM. Les machines étaient interconnectées avec un réseau LAN à 100 Mbps.

Description de l'application

Pour la réalisation des différents tests, nous avons utilisé une application simple où le serveur offre comme service le calcul de numéros aléatoires. Le client demande un nombre de numéros aléatoires, et le serveur les lui fournit dans un tableau. Le choix de ce service est relatif au temps de CPU nécessaire pour effectuer ce calcul qui est assez important. Ceci surcharge le serveur et augmente ainsi le temps de réponse. Pour réaliser les mesures, le client envoie 10 fois 200 000 requêtes avec une fréquence approximative de 1 500 requêtes par seconde, et mesure à chaque cycle le temps de réponse. La mesure de la charge des serveurs, effectuée par le proxy serveur, consiste à calculer le nombre de requêtes reçues par seconde.

Nous avons réalisé des tests avec plusieurs combinaisons de nombre de clients et de serveurs. Pour chaque politique d'équilibrage de charge, il s'agissait de lancer de 1, 2, 4, à 8 serveurs pour 1, 2, 4, 8, 16 et 22 clients.

3.5.2 Résultats et performances

Nous avons testé plusieurs configurations pour l'évaluation des générateurs qui comportent des politiques d'équilibrage de charge différentes¹⁰ :

- Tout d'abord nous avons commencé par tester l'utilisation de nos générateurs avec un seul serveur et plusieurs clients afin de comparer les résultats avec ceux d'une application n'utilisant aucun service d'équilibrage de charge ;
- Ensuite, nous avons augmenté le nombre de serveurs pour tester la capacité d'extension en utilisant les différents générateurs qui réalisent chacun une politique d'équilibrage de charge différente. Nous avons testé la productivité et le temps de réponse pour chaque générateur, c'est-à-dire pour chaque politique ;
- Enfin, nous avons comparé la performance des générateurs lorsque le nombre de serveurs est faible puis lorsqu'il augmente pour tester les politiques d'équilibrage de charge.

Dans ce qui suit, nous allons décrire les différentes configurations et interpréter les résultats obtenus.

Mesures avec un seul serveur

Cette première expérience a été réalisée sur une application où des clients envoient leurs requêtes sur un seul serveur. Le but de cette expérience est de montrer l'impact de l'utilisation des générateurs réalisés par rapport à l'utilisation du générateur Jérémie, c'est-à-dire sans

¹⁰Pour chacun des générateurs correspond une politique : Round Robin (RR), aléatoire (Random), ou le serveur moins chargé (LL pour *Least Loaded*).

appliquer la technique de l'équilibrage de charge. Nous avons mesuré la productivité du serveur en calculant sa charge en nombre de requêtes traitées par seconde pour un nombre de clients variant de 1 à 22 (cf. figure 3.9). Un nombre de requêtes traitées élevé est synonyme d'une bonne productivité. Nous avons également mesuré la latence en calculant le temps nécessaire au serveur pour traiter les requêtes des 1, 2, 4, jusqu'aux 22 clients (cf. figure 3.10). Dans ce dernier cas, un temps de réponse bas est synonyme d'une bonne performance.

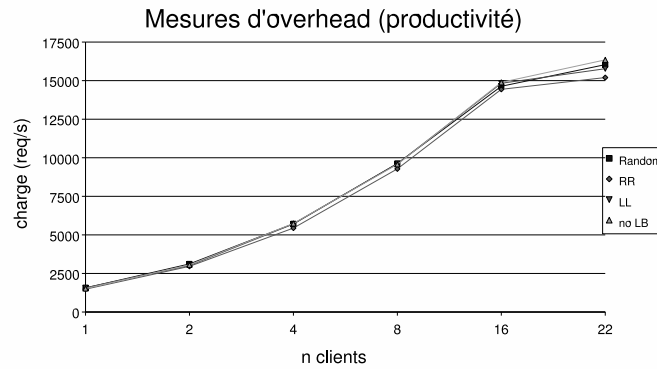


Figure 3.9 — Productivité avec un serveur

Dans la figure 3.9, on remarque que dans ce cas où on n'utilise qu'un seul serveur, la productivité du serveur lorsqu'on applique un générateur d'équilibrage de charge (Random, RR ou LL) est légèrement inférieure à celle du générateur Jeremie (no LB). Le serveur traite légèrement moins de requêtes lorsqu'on applique un générateur d'équilibrage car il y a un petit retard lors du premier changement de références¹¹. Par contre, en utilisant le générateur Jeremie les requêtes sont acheminées directement et dans ce cas le serveur reçoit plus de requêtes à traiter. Une fois que le client récupère la référence du serveur qui doit traiter la requête, le registre (*Registry*) n'est plus contacté car il n'y a plus besoin de changement de référence. Cependant, il est à noter que dans ce cas où le nombre de client maximum est atteint (22 clients), le serveur ne traite que la moitié du total des requêtes émises, ce qui prouve qu'il est surchargé.

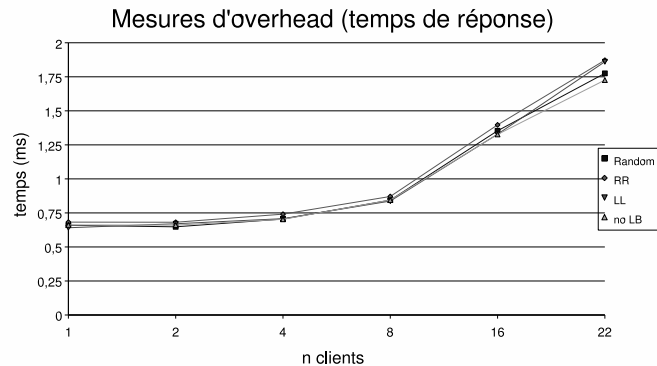


Figure 3.10 — Temps de réponse avec un serveur

¹¹Les générateurs utilisent une technique de remplacement de la référence actuelle du proxy client pour désigner le serveur qui doit traiter la requête.

De même, dans la figure 3.10 le temps de réponse du serveur dans le cas de l'application d'un générateur d'équilibrage de charge est légèrement supérieur pour la même raison : le temps perdu lors du changement de référence. Du moment qu'il y a un petit retard au début, ceci se répercute sur le traitement du reste des requêtes.

Dans cet exemple, les générateurs n'ont pas été utilisés pour accomplir leur fonction d'équilibrage de charge, ce qui justifie le léger avantage de l'utilisation du générateur Jérémie. Mais l'intérêt de cet exemple est de montrer la flexibilité de notre approche qui consiste à réaliser les connecteurs comme des générateurs. En effet, cet exemple montre qu'avec la même architecture d'application, c'est-à-dire un composant client demande un service à un composant serveur, il est possible d'utiliser différents connecteurs sans pour autant retoucher à ces composants. Ainsi, même si une propriété n'a pas été prévue dans l'architecture d'une application, elle peut être rajoutée sans affecter les composants interagissant. Ceci est d'autant plus vrai pour la reconfiguration. Comme nous allons le voir dans la suite, le passage d'un serveur à plusieurs se fait de façon complètement transparente par rapport aux composants clients. Il est possible de rajouter et retirer des serveurs sans toucher le connecteur ou les autres composants.

Mesure de la capacité d'extension

L'objectif de l'utilisation d'un service d'équilibrage de charge étant d'offrir une bonne capacité d'extension d'un système (*scalability*), nous allons montrer maintenant les résultats de productivité et de latence lors de l'utilisation des différents générateurs en augmentant le nombre de serveurs de 1, 2, 4 à 8 serveurs et avec le même nombre de clients que l'expérience précédente. Nous allons traiter chacune des politiques séparément : aléatoire, Round Robin et serveur le moins chargé.

a) Politique aléatoire :

La figure 3.11 montre les variations de la productivité du système en augmentant le nombre de clients et de serveurs en utilisant la politique d'équilibrage de charge aléatoire.

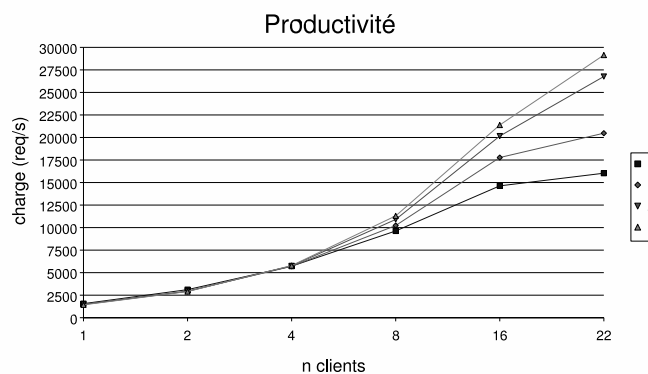


Figure 3.11 — Résultats de la productivité avec la politique aléatoire

Avec un nombre de clients inférieur à 4, la productivité est pratiquement la même pour les systèmes avec 1, 2, 4 et 8 serveurs. Par la suite, on remarque qu'au delà de 8 clients la productivité augmente jusqu'à atteindre son maximum lorsque les 22 clients envoient simul-

tanément leurs requêtes aux 8 serveurs présents¹². Cette expérience montre qu'en augmentant le nombre de serveurs non seulement on évite la surcharge du système, mais on peut supporter un plus grand nombre de requêtes. En effet, un système avec 8 clients produit 12 000 requêtes par seconde qui sont presque toutes traitées par les 8 serveurs. Avec le même nombre de clients, un seul serveur ne traite que moins de 10 000 requêtes et commence à saturer alors que les autres serveurs approchent plus des 12 000. De plus, dans le système avec 16 clients, alors qu'un système avec un seul serveur commence à atteindre sa limite¹³, un système avec 2 serveurs peut traiter un nombre supérieur de requêtes. Cette évolution est progressive, c'est-à-dire qu'un système avec 4 serveurs aura une productivité supérieure au système avec 2 serveurs et inférieure à un système avec 8 serveurs. Ce résultat montre que le générateur offre une bonne capacité d'extension au système.

Les résultats de la figure 3.12 montrent la variation de la latence en augmentant le nombre de clients et de serveurs. Ce graphique a un lien direct avec le précédent. Il est complètement inversé puisque les systèmes qui possèdent les meilleures productivités possèdent aussi les meilleurs temps de réponse et vice-versa. Par exemple, le temps de réponse avec 22 clients et un seul serveur est de 1.78 ms, avec 2 serveurs il est de 1.07 ms, avec 4 serveurs il est de 0.81, et avec 8 serveurs il est de 0.74 ms. Avec un temps de réponse au début légèrement inférieur dans le cas d'un seul serveur dû au changement de référence comme expliqué dans la section précédente. On remarque aussi que lorsque le système avec un seul serveur est sur le point de saturer (avec 8 clients), la latence augmente très rapidement.

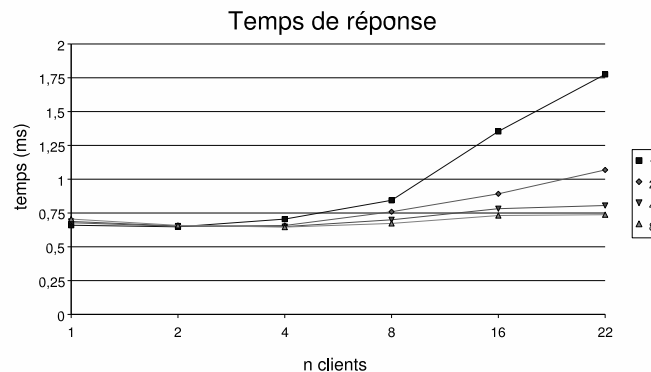


Figure 3.12 — Résultats de la latence avec la politique aléatoire

Ces données montrent que l'utilisation du générateur satisfait à ses tâches de diminuer le temps de réponse des serveurs en appliquant la politique d'équilibrage de charge aléatoire, et à satisfaire la montée en charge du système en étendant et en ajoutant au système de nouveaux serveurs. Ceci est conforme à notre objectif.

b) Politique Round Robin :

La figure 3.13 montre les variations de la productivité du système en augmentant le nombre de clients et de serveurs et en utilisant la politique d'équilibrage de charge Round Robin. Les résultats que nous pouvons observer sur cette figure sont équivalents à ceux de la politique aléatoire. Ceci reste logique puisque les deux politiques appartiennent à la

¹²Un client envoie 1 500 requêtes par seconde, les 22 clients en envoient 33 000 et on peut observer que les 8 serveurs traitent presque 30 000 requêtes.

¹³En augmentant le nombre de requêtes sa courbe commence à ressembler plus à une droite.

même famille, les politiques d'équilibrage de charge non adaptative, et les deux répartissent équitablement la charge.

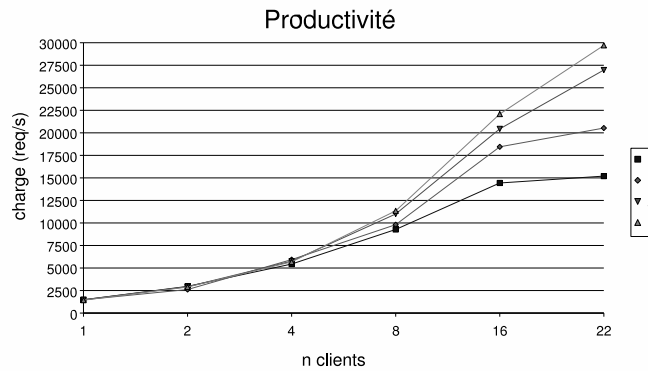


Figure 3.13 — Résultats de la productivité avec la politique Round Robin

Que les résultats soient équivalents ne signifie pas que les deux politiques possèdent la même productivité ni le même temps de réponse. Comme nous le verrons un peu plus tard, lorsque les requêtes arrivent de façon uniforme la politique Round Robin profite mieux de l'information disponible et effectue une répartition de charge adaptée.

Les résultats de latence de la figure 3.14 sont équivalents aux résultats de la politique aléatoire.

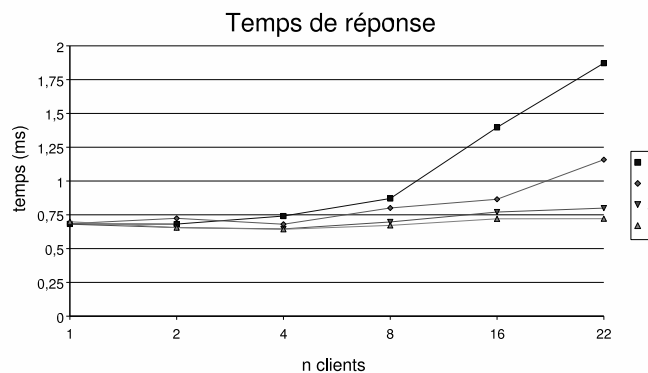


Figure 3.14 — Résultats de la latence avec la politique Round Robin

On peut déduire trois observations intéressantes :

- Avec un nombre fixe de clients, le temps de réponse diminue en augmentant le nombre de serveurs ;
- Avec un nombre élevé de serveurs, l'accroissement du temps de réponse se fait d'une façon modérée ;
- Lorsque le nombre de serveurs est faible, le temps de réponse augmente très rapidement.

Ainsi, nous pouvons affirmer que l'utilisation du générateur d'équilibrage de charge améliore la performance du système car plusieurs requêtes sont traitées potentiellement en parallèle. Le temps de calcul de chaque requête reste le même, c'est le temps d'attente qui est réduit.

c) Politique du « serveur moins chargé » :

Contrairement aux politiques précédentes, cette politique est adaptative. Le choix de la référence du serveur qui doit traiter les requêtes des clients est réalisé en fonction de la charge de travail réelle du serveur. Intuitivement, on peut penser que ce type de politique fait une meilleure répartition de charge. Cependant, il faut tenir compte du fait que le choix de l'élément le moins chargé nécessite la vérification de la charge de tous les serveurs et que la mesure de cette charge doit se faire régulièrement et avec des périodes de temps bien choisies. Tout ceci a un coût non négligeable (comme nous le verrons dans la section suivante).

La figure 3.15 montre l'évolution de la productivité du système en augmentant le nombre de clients et de serveurs. Même si les données ne sont pas égales à celles des politiques précédentes, les conclusions sont équivalentes. L'utilisation du connecteur avec cette politique assure une meilleure capacité d'extension (*scalability*) du système.

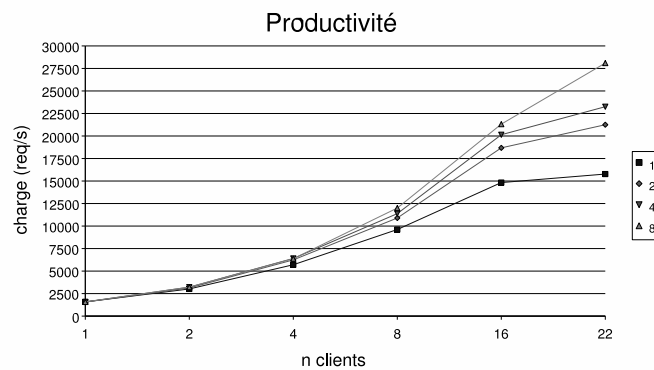


Figure 3.15 — Résultats de la productivité avec la politique du moins chargé

La figure 3.16 expose la variation du temps de réponse d'après le nombre de clients et serveurs présents dans le système. On peut observer que la latence diminue lorsque le nombre de serveurs augmente. Un temps de réponse plus petit permet aux serveurs de traiter plus de requêtes et d'augmenter la productivité globale du système.

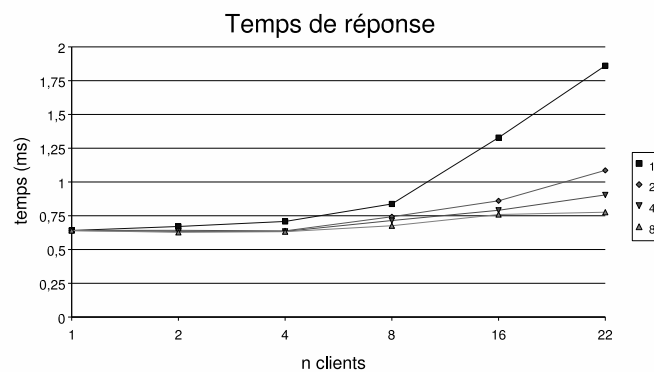


Figure 3.16 — Résultats de la latence avec la politique du moins chargé

Comparaison des performances des politiques

Nous avons étudié dans les points précédents le comportement des différentes politiques lorsqu'on augmente le nombre de clients et de serveurs. Toutes les politiques permettent d'accroître la performance du système en augmentant la productivité et en diminuant le temps de réponse. Nous allons comparer dans cette section les performances obtenues des différentes politiques et voir quelles conclusions on peut en déduire sur l'architecture. En effet, le fait que les résultats soient équivalents ne veut pas dire que les performances soient les mêmes. Afin de relever les différences, nous avons effectué les comparaisons pour deux cas : un nombre faible de serveurs et un nombre élevé de serveurs :

a) Nombre faible de serveurs :

Pour étudier ce cas, nous avons choisi de comparer les résultats obtenus avec deux serveurs. Ce cas possède deux caractéristiques. D'un côté, on remarque dans les courbes de productivité de chaque politique (figures 3.11, 3.13 et 3.15) que le système commence à être surchargé avec un nombre de clients égal à 22^{14} . D'un autre côté, étant donné que le nombre de serveurs est faible (2), le temps nécessaire pour parcourir le registre ainsi que pour faire le choix du serveur est réduit.

La figure 3.17 représente l'évolution de la productivité pour les trois politiques en passant de 1 à 22 clients avec deux serveurs.

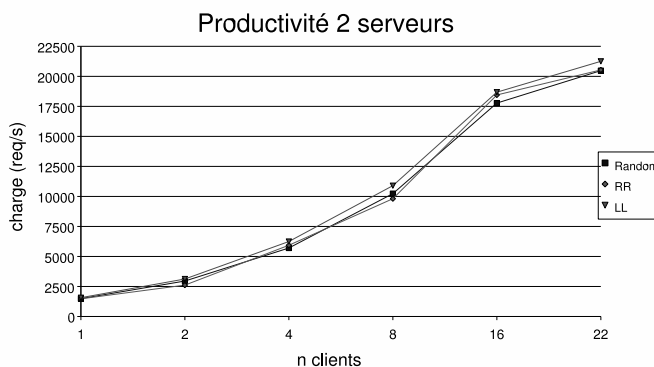


Figure 3.17 — Résultats de productivité avec 2 serveurs

Le comportement des politiques est équivalent, le dessin des courbes est quasi identique. Cependant, même si les politiques non adaptatives obtiennent une productivité semblable, la politique du « serveur moins chargé » obtient des résultats un peu plus performants lorsque le nombre de clients est élevé. En effet, dans les politiques non adaptatives, un changement de référence du serveur qui doit traiter la requête est fait à chaque requête (c'est un traitement par requête). Par contre dans la politique adaptative une re-direction des requêtes n'est effectuée que dans le cas où le serveur qui était le moins chargé exprime une variation de la charge significative. Ainsi, la politique du moins chargé provoque beaucoup moins de changement de références.

¹⁴La courbe commence à s'incliner pour ressembler à une droite.

b) Nombre élevé de serveurs :

Pour réaliser cette étude, nous avons choisi les résultats obtenus avec 8 serveurs. Avec un nombre élevé de serveurs, le système supporte sans problème l'évolution de la charge due à l'augmentation des clients. Cependant, avec ce nombre élevé de serveurs le temps de décision pour choisir le serveur possède une certaine influence sur le temps de réponse.

Dans la figure 3.18, on peut observer la productivité du système avec les trois politiques lorsque le nombre de clients augmente de 1 à 22 avec 8 serveurs. Même si les trois politiques réalisent un bon équilibrage de charge et offrent une bonne capacité d'extension, nous pouvons noter des différences par rapport au cas précédent. La politique du moins chargé offre des résultats meilleurs lorsque le nombre de clients est inférieur au nombre de serveurs, mais offre des résultats moins bons dans le cas contraire. Ceci s'explique par le fait que lorsqu'on augmente le nombre de clients, le composant PS doit évaluer plus fréquemment le choix du serveur le moins chargé, et sur plus de serveurs.

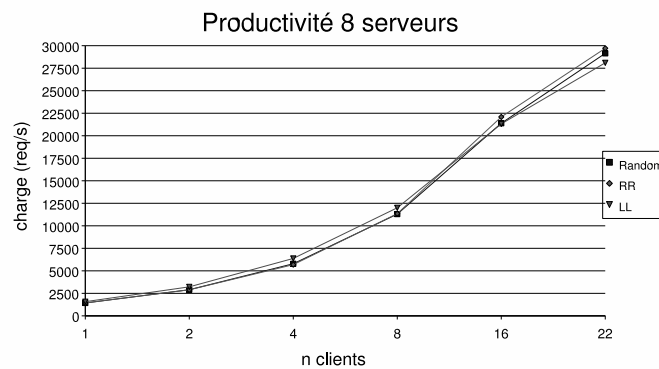


Figure 3.18 — Résultats de productivité avec 8 serveurs

En ce qui concerne les politiques non adaptatives, les résultats sont presque équivalents, avec un meilleur comportement de la politique Round Robin. On ne remarque cette différence que lorsque le nombre de clients est élevé, et on peut supposer qu'avec un nombre de clients plus important, on pourrait apercevoir une différence plus importante. Cela est dû au fait que le choix d'un numéro aléatoire prend un temps significatif lorsque le domaine est plus large. D'autre part, lorsque les requêtes arrivent avec une politique uniforme, la politique Round Robin effectue une répartition de la charge parfaitement équitable, avec un temps de calcul du choix du serveur négligeable.

La figure 3.19 montre l'évolution du temps de réponse quand le nombre de clients augmente de 1 à 22 pour 8 serveurs avec les trois politiques. Ces résultats complètent l'argumentation de l'étude de la productivité, les courbes sont également inversées.

À partir de ces expériences, nous pouvons conclure qu'on peut utiliser l'une des politiques ou l'autre suivant la fréquence des requêtes. Si elles sont uniformes, la politique non adaptative est plus efficace, sinon c'est la politique du moins chargé qui est plus efficace. Ainsi, on peut utiliser l'un ou l'autre sans affecter l'application.

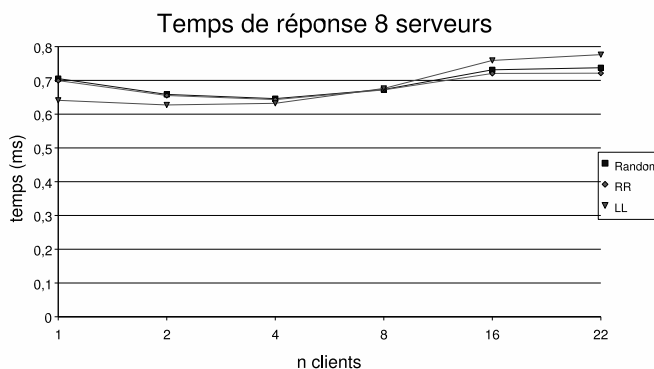


Figure 3.19 — Résultats de latence avec 8 serveurs

3.6 Synthèse

Nous avons vu dans cette section les principales caractéristiques de l'équilibrage de charge qui constitue la propriété du connecteur qui nous a servi de référence tout le long de cette thèse. Nous avons vu qu'il existe des stratégies qui effectuent l'équilibrage en fonction de la charge de travail des serveurs (adaptatives) ou indépendamment de cette charge (non adaptatives). Ces stratégies peuvent être implantées sur différents niveaux d'abstraction et nous avons choisi d'implémenter ce connecteur comme un générateur au niveau intergiciel. Ce dernier fournit un bon compromis entre la transparence de la propriété et la prise en compte de métriques de charge évoluées qui sont liées à l'application.

Nous avons implémenté le connecteur comme une famille de générateurs, au-dessus de la plate-forme Jonathan. Ces générateurs varient suivant la politique d'équilibrage de charge utilisée. La plate-forme Jonathan offre à la base des générateurs CORBA et RMI. En y intégrant les générateurs d'équilibrage de charge nous l'avons transformé d'un simple bus logiciel transportant des requêtes et des réponses en un générateur de composants de communication capables de prendre des décisions relatives aux communications entre les composants. Ce connecteur associé à un ensemble de générateurs offre plusieurs avantages :

- Il permet de réaliser la propriété d'équilibrage de charge avec transparence tout en ayant la possibilité d'utiliser des métriques de charge relatives à l'application ;
- Il offre une grande flexibilité pour le changement de sémantique de communication qui se fait par une simple substitution du connecteur et la re-génération du composant de liaison ;
- Donc, il est possible de remplacer le générateur du connecteur d'équilibrage de charge adaptatif avec le connecteur d'équilibrage de charge non adaptatif sans modifier ni réécrire les composants interagissant ;
- De même, il est toujours possible d'utiliser le générateur de Jonathan dans la même application sans réécrire ni re-concevoir les composants. Il est ainsi facile de reconfigurer les applications avec une grande souplesse.

Nous avons testé les performances des générateurs d'équilibrage de charge adaptatif et non adaptatif que nous avons réalisés et nous avons obtenu des résultats satisfaisants qui valident l'approche que nous défendons.

4 Classification

Nous avons vu dans la section 2.2 notre définition de la notion de connecteur. Dans ce chapitre, nous introduisons une classification des entités de communication entre les composants logiciels. Nous distinguons les connecteurs des composants de communication, ou « médiums » introduits dans la thèse d'E. Cariou [17]. Plus précisément, nous présentons deux moyens différents pour capitaliser le « savoir-faire » de la description et de la réalisation des abstractions de communication, dans le but de les préserver et de les réutiliser.

Il existe plusieurs travaux qui traitent la différence entre les connecteurs et les composants, et qui encouragent la différenciation et la séparation entre les propriétés fonctionnelles des composants de leur communication [5, 73] en mettant en avant les avantages de cette séparation. Le but de ce chapitre n'est pas de traiter cette différence mais de comparer les connecteurs à des composants particuliers : les composants de communication. Ces composants ont la même fonctionnalité que les connecteurs, c'est-à-dire réaliser la communication entre les composants standards, mais ces deux entités sont de nature différente. Les différenciations entre composants et connecteurs sont habituellement d'un point de vue purement fonctionnel. Nous proposons une distinction relative à la nature même de ces entités. Cette distinction apporte deux éléments : une différence dans le cycle de développement et une différence dans la nature des objets à mettre sur étagère.

Dans la suite de cette section, nous commençons par opposer les composants de communication aux connecteurs en précisant les principales caractéristiques de chacune des deux entités. Ensuite, nous donnons quelques éléments de différenciation, de choix de fabrication et d'utilisation pour les deux entités. Enfin, nous donnons une classification de certains moyens de communication usuels suivant qu'ils correspondent à des connecteurs ou à des composants de communication.

4.1 Deux types d'abstraction de communication

Nous distinguons deux moyens pour la capitalisation de la communication entre les composants logiciels : les connecteurs, tels que nous les définissons dans la section 2.2.3, et les composants de communication, qui ont fait l'objet de la thèse d'Eric Cariou [17]. Dans ce qui suit, nous commençons par définir les composants de communication, appelés aussi médiums, ensuite nous relèverons les principales caractéristiques de chacune des deux notions. Nous donnerons enfin des exemples pour éclaircir les concepts.

4.1.1 Les composants de communication : Définition

Un composant de communication, ou médium¹, est une réification d'un système d'interaction, de communication ou de coordination, d'un protocole ou d'un service dans un composant logiciel. Ces systèmes ou protocoles d'interaction implémentés ou intégrés dans un médium varient dans le type et la complexité. Des exemples sont : la diffusion d'événements ou la coordination à travers une mémoire partagée (tel que Linda [44]). Une application distribuée peut être construite en interconnectant des composants « conventionnels » avec les médiums qui gèrent leur communication et leur distribution.

Un médium est avant tout un composant, il satisfait les propriétés principales du paradigme de composant définies dans [87] :

- Il s'agit d'une entité logicielle autonome et déployable ;
- Il spécifie clairement ses services offerts et requis. Ceci permet l'utilisation du composant sans avoir besoin de connaître son fonctionnement. Ses services sont atteints à travers des points d'accès explicites et typés appelés ports ;
- Il peut être composé avec d'autres composants.

Ainsi, un médium possède toutes ces propriétés mais, de plus, sa fonctionnalité est dédiée à la communication. Par nature, c'est un composant aux interfaces multi localisées ce qui fait sa différence par rapport aux modèles à composants standards. En effet, les modèles tels que EJB, .NET ou CCM permettent la réalisation de composants répartis mais dont les interfaces sont mono-localisées. Un exemple d'un composant de communication est le protocole dans les architectures de communication réseau. Il offre des interfaces explicites qui sont utilisées par les niveaux supérieurs de la pile protocolaire. Il est destiné à être installé sur plusieurs sites distants, ses interfaces sont donc multi localisées.

La spécification d'un médium repose sur le langage UML. Cette spécification décrit toutes les informations nécessaires à l'utilisation d'un médium. Ces informations peuvent être regroupées dans un contrat [11, 59]. Ce contrat doit inclure la liste des services offerts et requis par un médium mais cela ne suffit pas. Il doit aussi spécifier la sémantique et le comportement dynamique de chacun des services et du médium. Cette méthodologie définit des contrats au niveau de la spécification indépendamment de tout choix d'implémentation. Les composants utilisent, en fonction de leur besoin, certains services du médium mais pas tous. Dans un médium de diffusion par exemple, un composant voulant émettre utilise un service d'émission. Les autres composants désirant seulement recevoir des informations n'ont besoin que du service de réception. Les composants sont donc classés dans l'interaction en fonction du *rôle* qu'ils jouent du point de vue des services du médium qu'ils utilisent. Un médium est représenté par une collaboration UML. Les rôles des composants utilisant un médium pour leurs communications correspondent aux rôles utilisés dans les collaborations UML.

Comme les composants, le médium offre des services qui sont utilisés par d'autres composants. Le médium peut aussi avoir besoin d'appeler des services sur ces composants. Les services requis et offerts sont regroupés dans des interfaces offertes et requises. À chaque rôle de composant peut être associé une interface de services offerts (par le médium aux composants jouant ce rôle) et une interface de services requis (par le médium sur les composants

¹Dans la suite nous utiliserons plus souvent le terme médium pour désigner le composant de communication afin de le différencier des composants fonctionnels conventionnels.

jouant le rôle).

Un médium est spécifié à l'aide de trois « vues » UML :

- Un diagramme de collaboration pour décrire l'aspect structurel du médium. Des messages peuvent être ajoutés afin de décrire les appels d'opérations réalisés dans le cadre de l'exécution d'un service. La numérotation des messages permet de spécifier l'ordonnancement de ces appels ;
- Des contraintes OCL [89, 68] qui permettent de spécifier les propriétés du médium ainsi que la sémantique statique des services offerts et requis (pré et postconditions sur les services) ;
- Des diagrammes d'état associés au médium ou à ses services afin de gérer les contraintes temporelles et de synchronisation et de spécifier la sémantique dynamique des services.

Une collaboration représentant le médium est définie au niveau spécification ce qui permet de spécifier le médium de manière « générale ». Une collaboration au niveau instance ne permet pas de faire cela, elle ne représente qu'un cas particulier d'une interaction et ne permet pas de généraliser le fonctionnement d'une interaction. Afin de prendre en compte les caractéristiques des composants en général et des médiums en particulier, la structure de la collaboration suit une forme particulière. À l'intérieur de la collaboration, une classe représente le médium dans son ensemble. Les interfaces de services offerts et requis doivent être présentes ainsi que tous les rôles des composants pouvant se connecter au médium. Dans un diagramme de collaboration représentant le médium, un rôle correspond à un composant, jouant un rôle donné, *connecté au médium*. La figure 4.1 montre la structure générique entre un rôle de composant générique et un médium.

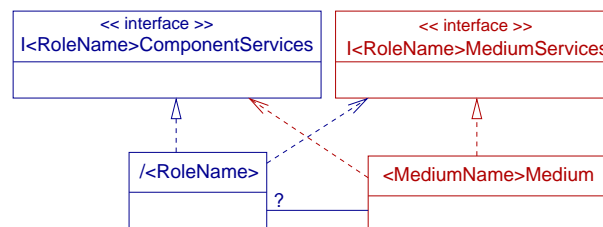


Figure 4.1 — Relation entre un rôle et un médium

Un médium nommé « <MediumName> » est représenté dans le diagramme de collaboration par une classe nommée <MediumName>Medium². Pour chaque rôle « <RoleName> » de composant, il existe deux interfaces :

- I<RoleName>MediumServices qui est l'interface regroupant les services *offerts* par le médium aux composants jouant le rôle « <RoleName> ». C'est-à-dire les services utilisés par ces composants. Ce rôle a donc une dépendance sur cette interface et le médium implémente cette interface ;
- I<RoleName>ComponentServices qui est l'interface regroupant les services qui doivent être réalisés par le composant jouant le rôle « <RoleName> » et qui sont appelés par le médium. Le médium a donc un lien de dépendance sur cette interface.

²Ce sont des conventions de nommage.

Dans la section qui suit, nous allons comparer les médiums aux connecteurs en faisant ressortir leurs principales caractéristiques.

4.1.2 Les connecteurs vs les composants de communication

Nous proposons de différencier les deux moyens pour la capitalisation de la sémantique de communication au niveau architecture : les composants de communication (médiums) et les connecteurs. À partir de la définition d'un composant de communication et des brèves explications sur sa spécification de la section précédente, nous décrivons dans cette section les principaux éléments ou les caractéristiques qui font qu'un connecteur est différent d'un composant, qu'il soit un composant de communication (médium) ou un composant conventionnel.

Comme nous l'avons déjà montré précédemment (section 2.1), la représentation des connexions complexes par une combinaison de communications point-à-point (les traits dans le modèle *boxes-and-lines*) peut être traduite de plusieurs manières différentes et conduit ainsi à des interprétations ambiguës de l'architecture d'une application. Nous avons proposé de donner une représentation différente aux connecteurs complexes en remplaçant la combinaison de traits par une ellipse. Désormais, nous représentons toutes les entités qui capitalisent des communications complexes, c'est-à-dire les médiums et les connecteurs, par des ellipses. De cette façon nous préservons la différence de fonctionnalité : les composants représentés par des boîtes capitalisent les propriétés fonctionnelles de l'application et les ellipses capitalisent les propriétés de communication entre ces composants. Cependant, nous introduisons une différence entre les éléments d'interaction : une différence de nature. Cette différence concerne la nature des interfaces des deux abstractions de communication. Le médium définit des interfaces explicites, elles sont de même nature que celles des composants. Le connecteur définit des interfaces implicites, donc différentes des interfaces des médiums et des composants.

Dans les figures 4.2 (a) et (b), les composants fonctionnels sont représentés au niveau architecture par les boîtes. La description des interactions entre les composants est isolée dans deux entités différentes ayant la même forme, l'ellipse. Cependant, les interfaces de ces ellipses sont représentées de deux manières différentes car elles sont de natures différentes :

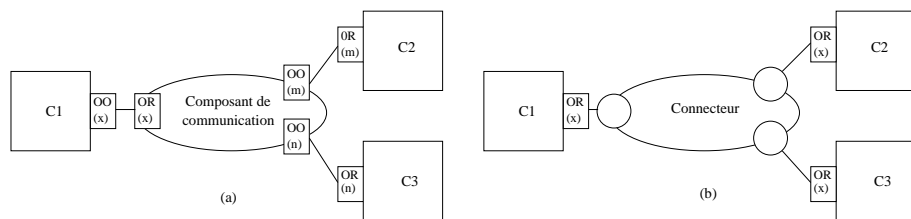


Figure 4.2 — Deux types d'abstraction de communication. (a) médium, (b) connecteur

1. Dans la figure 4.2 (a), l'ellipse représente le médium. Ce dernier possède des points d'attache explicites appelés ports qui sont représentés par des *carrés* autour de l'ellipse. Les ports implémentent les interfaces du médium qui définit des services propres à lui. Les signatures des méthodes requises et offertes sont ainsi bien définies et spécifiées. Dans cette figure, le médium requiert l'opération offerte $OO(x)$ du composant standard $C1$, et offre deux opérations $OO(m)$ et $OO(n)$. Ces dernières sont requises par les opérations $OR(m)$ et $OR(n)$ des composants standards $C2$ et $C3$ respectivement. Les opérations x ,

m et n de $C1$, $C2$ et $C3$ sont complètement différentes, elles ne se connaissent pas. Elles interagissent avec les opérations x , m et n du médium en utilisant un connecteur simple (un appel de procédure par exemple).

2. Dans la figure 4.2 (b), l'ellipse représente le connecteur comme une abstraction de première classe. Ses points d'attache implicites, les prises, sont représentés par des *ronds* autour de l'ellipse. Les prises reflètent des interfaces implicites et génériques, elles ne spécifient pas directement des opérations offertes ou requises. Elles donnent seulement des contraintes sur des interfaces explicites futures. Ces interfaces sont adaptables aux interfaces des composants standards qui s'y connecteront à l'assemblage. Dans cette figure, l'opération offerte $OO(x)$ du composant $C1$ est requise par les opérations $OR(x)$ des composants $C2$ et $C3$ mais les appels passent à travers le connecteur pour y suivre la sémantique de la communication. L'opération x , offerte par $C1$, est la même qui est requise par $C2$ et $C3$. Ces composants ne voient pas le connecteur, ils ne demandent pas directement ses services. La communication entre les composants standards se fait à travers le connecteur mais celui-ci n'est pas visible par les composants.

La construction d'applications en utilisant les deux formes d'abstraction de communication présentées ci-dessus offre une meilleure clarté et compréhension des architectures logicielles. En effet, isoler et expliciter la sémantique de communication dans une entité dédiée à cet effet lève les ambiguïtés d'interprétation des interactions multiples qui peuvent exister entre les composants. Cette distinction entre composants fonctionnels et entités se chargeant de la communication permet la réutilisation des composants ainsi que la réutilisation de la communication entre ces composants. Elle offre aussi d'autres avantages comme faciliter la maintenance et l'évolution de toutes les entités logicielles. En effet, ces entités étant différentes une intervention sur l'une d'elles, quelle qu'elle soit, permet de ne pas impliquer toutes les autres entités qu'il y a autour, ce qui fait partie des objectifs primordiaux de l'architecture logicielle.

Il nous paraît important d'ajouter au niveau architecture une distinction entre les éléments de communication. Les deux entités de communication doivent exister isolées au niveau architecture et chacune d'entre elles doit être décrite indépendamment de tout contexte et de tout environnement. Cette classification vient dans le souci d'uniformiser et de guider le passage de l'abstraction au déploiement. Bien que le but et la finalité des médiums et des connecteurs soient les mêmes, les deux entités sont différentes. Elles existent pour relier un ensemble de composants au niveau architecture en amont et obtenir un ensemble de composants déployés à l'exécution en aval. Cependant, les techniques de mise en relation et les chemins pour arriver au déploiement sont très différents. Elles offrent toutes les deux le moyen de préserver la sémantique de communication jusqu'au déploiement en réifiant les communications complexes et en utilisant les langages et outils existants mais chacune à sa manière, elles n'ont pas le même processus de fabrication et d'utilisation.

Cette distinction permet d'un côté de diversifier les concepts, et ainsi d'offrir aux architectes le choix d'utilisation d'un concept ou de l'autre pour orienter la suite du déroulement du processus de développement de l'application. D'un autre côté, elle offre plus de liberté et de souplesse pour utiliser l'entité qui convient le mieux au type de l'application à créer, et ainsi d'enrichir l'expressivité des architectures et d'améliorer la qualité du logiciel. Ces deux modèles de connexion constituent deux méthodes différentes et complémentaires pour guider le passage de la description de l'interaction du niveau architecture à la connexion réelle au niveau déploiement. C'est une façon d'anticiper et de préméditer la suite du déroulement du processus de développement. Nous traitons les différences et donnons des éléments de choix entre un médium et un connecteur dans la section 4.2. Mais avant cela, nous donnons dans la

section qui suit un exemple d'utilisation des deux entités pour l'équilibrage de charge comme sémantique de communication en mettant en avant leurs différences.

4.1.3 Exemples

Pour mieux comprendre la différence entre les composants de communication et les connecteurs, et pour pouvoir les comparer, nous allons illustrer l'usage de ces concepts à travers des exemples. Pour cela, nous allons reprendre l'exemple de la propriété d'équilibrage de charge et voir sa réalisation comme un médium, puis nous le comparerons avec le connecteur d'équilibrage de charge.

Utilisation d'un composant de communication dans l'application de l'équilibrage de charge

Nous allons voir dans cette section la réification de la propriété d'équilibrage de charge sous forme d'un composant de communication afin de la comparer avec sa réification sous forme d'un connecteur. Pour cela, nous allons commencer par décrire la spécification d'un médium d'équilibrage de charge avec une collaboration. Ensuite, nous allons montrer son utilisation dans une application. Enfin, nous verrons les inconvénients de cette réalisation.

Comme tout médium, le médium d'équilibrage de charge doit spécifier des interfaces bien définies destinées à être utilisées explicitement par des composants. Il existe deux ensembles de composants : ceux qui envoient des requêtes à répartir équitablement et ceux qui traitent ces requêtes. Le médium comporte alors deux rôles : un rôle client et un rôle serveur. Le diagramme de collaboration du médium d'équilibrage de charge « *LBMedium* » (pour *Load Balancing Medium*) est représenté dans la figure 4.3.

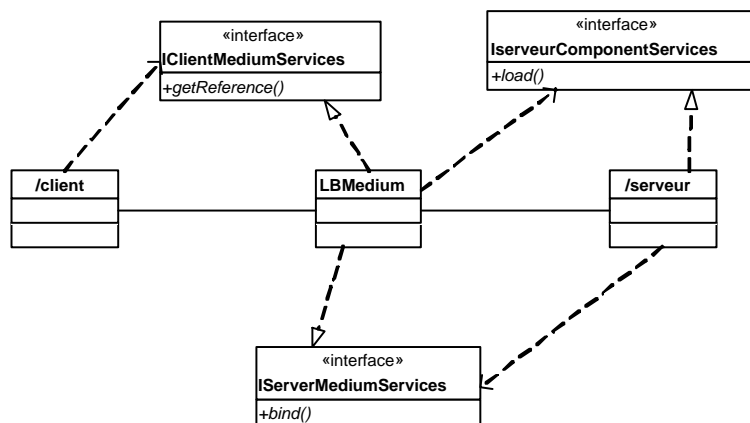


Figure 4.3 — Diagramme de collaboration du médium d'équilibrage de charge

La classe `LBMedium` représente le médium et possède deux rôles : `client` et `serveur`. On y retrouve les deux interfaces de services offertes qu'implémente le médium. La première interface, `IClientMediumServices`, définit l'opération `getReference()` qui renvoie la référence du serveur qui doit traiter la requête. Elle possède une dépendance du rôle `client`. La seconde interface, `IServerMediumServices`, définit l'opération `bind()` qui permet à un serveur de se déclarer. Elle possède une dépendance du rôle `Serveur`. On y retrouve également l'interface requise par le médium et offerte par un autre composant, `IServerComponentServices`,

qui définit l'opération `load()` permettant au médium de récupérer la charge du serveur connecté. Ainsi, les composants clients et serveurs qui doivent collaborer *savent* qu'un service d'équilibrage de charge doit être utilisé. Ces composants ne se préoccupent pas de la manière dont ce service est implémenté, et donc de comment est réalisée la politique d'équilibrage de charge.

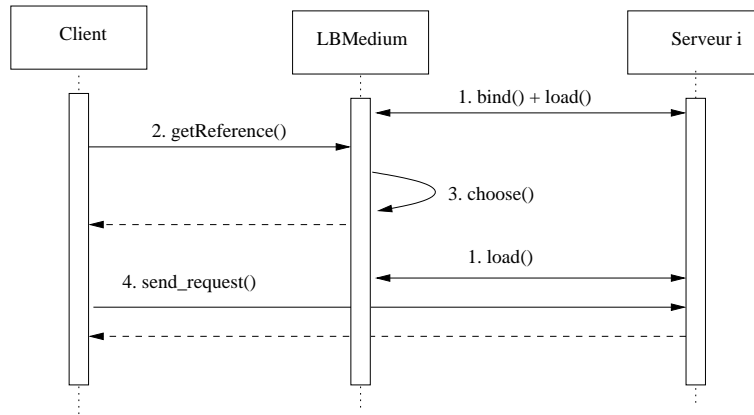


Figure 4.4 — Equilibrage de charge avec un composant de communication

Le diagramme de séquence, figure 4.4, montre une application client/serveur qui utilise le médium d'équilibrage de charge. Il illustre les différentes interactions entre les composants conventionnels — le client et les serveurs — et le LBMedium. Chaque interaction représentée dans cette figure est décrite ci-dessous :

1. Les serveurs communiquent explicitement avec le LBMedium pour s'enregistrer et lui communiquent leur charge ;
2. Le client appelle explicitement la méthode `getReference()` offerte par le composant de communication pour obtenir la référence du serveur qui traitera ses requêtes ;
3. Après application de la politique d'équilibrage de charge, le LBMedium choisit le serveur et renvoie sa référence au client ;
4. Le client lance alors l'appel effectif relatif à la requête en utilisant la référence obtenue. Cet appel peut être un appel de procédure classique ou bien à distance au travers d'un connecteur (RMI par exemple) qu'on ignore ici.

Cette approche préserve les avantages de l'équilibrage de charge du niveau application. Elle permet d'utiliser des métriques qui sont liées à l'application en assurant en plus la séparation des responsabilités. Elle permet de décharger le client du code encombrant lié à la réalisation de la politique d'équilibrage de charge et non lié à sa fonctionnalité. Ceci rend la réutilisation, la maintenance et l'évolution des programmes plus faciles. En effet, la modification de la politique d'équilibrage de charge n'affecte ni le client ni le serveur. Cependant, cette approche présente quelques limites :

- Les services du composant de communication étant explicites, le service d'équilibrage de charge n'est transparent ni du côté du client ni du côté du serveur. L'interaction entre tous les composants est totalement explicite. Ainsi, le remplacement du LB-Medium par un autre composant réalisant la même sémantique mais présentant des interfaces différentes affecte les autres composants. Il faudrait sinon standardiser les interfaces ;

- Lors de l'utilisation de l'interface offerte `getReference()`, les interfaces peuvent coïncider et bien aller ensemble, ou non. Dans ce dernier cas, il faudrait utiliser un adaptateur sur mesure ;
- Cette approche est efficace dans un mode de connexion par session, c'est-à-dire que le client envoie toutes ses requêtes au premier serveur dont il a reçu la référence par le composant de communication. Elle est cependant moins adaptée au mode de connexion par requête, où le client peut changer de serveur lorsqu'il y a changement de charge de travail. L'indirection due à l'utilisation explicite du médium ralentirait le système ;
- Ce service doit être prévu à l'avance et intégré dans l'application. Il doit être prévu à la conception des composants interagissant. Les serveurs calculent leur charge pour qu'elle soit récupérée par le médium. Ainsi, les serveurs doivent fournir, en plus des services auxquels ils ont été conçus, un service pour diffuser ou laisser chercher leur charge. Les clients doivent aussi être conçus en prenant en compte le fait qu'un service d'équilibrage de charge existe.

Avec cette approche on gagne en séparation des préoccupations mais on perd en efficacité car il est difficile de reconfigurer l'application sans toucher aux composants fonctionnels.

Comparaison avec l'utilisation d'un connecteur d'équilibrage de charge

Nous avons détaillé dans le chapitre précédent la réalisation de deux connecteurs d'équilibrage de charge sous forme de générateurs : le connecteur d'équilibrage de charge non adaptatif et le connecteur d'équilibrage de charge adaptatif. Nous reprenons dans cette section l'exemple de l'application client/serveur avec l'utilisation du connecteur d'équilibrage de charge adaptatif pour la comparer avec l'application de l'exemple de la section précédente qui utilise un médium pour réaliser le service d'équilibrage de charge. La figure 4.5 reprend le diagramme de séquence qui schématise l'enchaînement des événements dans l'application déployée avec un composant de liaison après la génération des proxies précédemment présenté (cf. figure 3.7 page 80).

Dans cette figure, l'appel du service offert, réalisé comme un appel local par le client, est traduit par un envoi de message (`sendRequest()`) intercepté par le proxy (`PxC`) du composant de liaison. Ainsi, dans cet exemple et en comparaison avec la séquence d'événement de l'exemple précédent, le client fait un appel explicite du service dont il a besoin et il reçoit directement la réponse. L'appel n'est pas adressé à un service du connecteur puisque celui-ci n'en offre pas. Par, contre dans l'exemple précédent le client fait d'abord un appel explicite vers le médium pour connaître l'identité du serveur qui va traiter ses requêtes, puis effectue l'appel effectif relatif à sa requête. Le service d'équilibrage de charge est visible lors de l'utilisation d'un médium alors que l'interaction entre le composant client et le connecteur est transparente. De plus, cette solution est plus adaptée au mode de connexion par requête car le changement de serveur est réalisé de façon transparente dans le proxy.

De même du côté du serveur, le composant serveur n'a pas à prévoir un service pour faire connaître sa charge. Celle-ci est récupérée avec transparence par le proxy serveur du composant de liaison généré. En effet, la surveillance de la charge de travail n'est plus de la responsabilité du serveur mais du proxy qui lui est attaché. Toutefois, pour assurer cette transparence en utilisant un médium, il est possible d'imaginer une solution où c'est le contrôleur du composant serveur qui collecte les informations sur la charge par un accès système. Par contre,

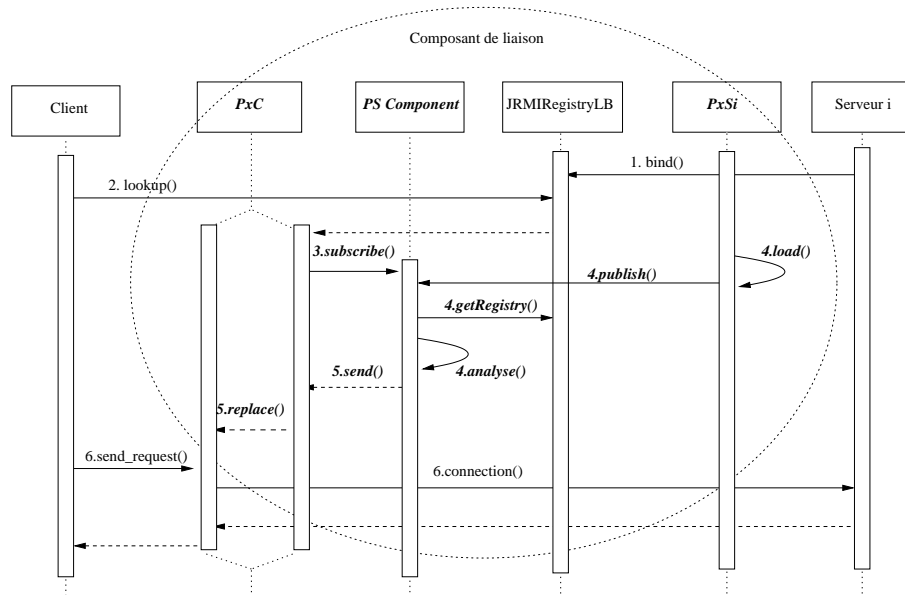


Figure 4.5 — Diagramme de séquence pour la politique d'équilibrage de charge du serveur le moins chargé

cette solution ne permet pas une grande liberté pour le choix de la métrique, le contrôleur du serveur n'a pas accès au nombre de requêtes reçues par exemple. Nous avons utilisé cette dernière métrique dans l'exemple de mise en œuvre du connecteur d'équilibrage de charge adaptatif qui a accès à ce type d'information applicative. Ainsi, l'utilisation d'un connecteur qui récupère la charge avec transparence semble la plus adaptée. Cependant, il peut exister des cas où la métrique de charge est tellement dépendante de l'application (par des calculs relatifs aux données métiers par exemple) qu'il n'est plus possible de garantir cette transparence. Comme il serait nécessaire d'effectuer une communication explicite entre le serveur et l'abstraction de communication, il serait alors plus approprié d'utiliser un médium. Une autre solution intermédiaire serait d'utiliser les connecteurs complexes que nous désignons par `ComplexConnector`, en référence à nos travaux dans le projet ACCORD [45]. Il s'agit d'une entité hybride entre le médium et le connecteur qui possède des interfaces implicites et explicites. L'utilisation d'une telle entité permettrait de préserver une part de transparence.

Dans les deux exemples que nous avons présentés, les deux abstractions de communication permettent au client d'envoyer ses requêtes à un serveur sans se préoccuper de la politique d'équilibrage de charge. La solution utilisant un connecteur possède plus d'avantages. Elle permet d'assurer la propriété avec plus de transparence. Les proxies générés ont la responsabilité de cacher les détails de distribution, de choisir le serveur qui traite les requêtes, et de surveiller et récupérer la charge des serveurs. Ils permettent aussi de connecter n'importe quels composants satisfaisant aux contraintes exprimées par les prises et d'offrir ainsi plus d'adaptabilité. Le connecteur permet de gagner en efficacité car il n'y a pas d'indirection et il n'y a pas besoin d'adaptateurs. Il permet en plus une meilleure flexibilité notamment par rapport à la substitution. Par exemple son remplacement n'affecte pas les composants, il suffit juste de faire une régénération des proxies avec le générateur d'un autre connecteur.

L'utilisation de l'une des deux entités reste néanmoins un choix de conception. La section suivante donne plus d'éléments de différence et de choix.

4.2 Différences et éléments de choix et d'utilisation

Nous avons présenté dans la section précédente deux entités différentes qui permettent de capitaliser les interactions entre les composants logiciels. Nous allons résumer dans cette section les principales différences qui existent entre les deux entités et donner quelques indications et éléments de choix pour la réification et l'utilisation d'une propriété de communication sous forme de composant de communication ou de connecteur.

4.2.1 Éléments de différenciation

Les composants de communication et les connecteurs possèdent un même objectif : contenir les interactions entre les composants ; et une même fonctionnalité : assurer les communications et la coordination entre les composants. Cependant, ils possèdent une différence de nature. Celle-ci se traduit principalement par une différence dans la nature des interfaces et dans le cycle de vie des deux entités. Cette différence engendre une différence de description, d'implémentation et d'utilisation.

Différence de description

Au niveau description et spécification, la différence entre un composant de communication et un connecteur se manifeste dans la nature des interfaces.

Comme décrit dans la section 4.1.1, le médium offre des interfaces explicites qui constituent ses points d'interaction avec l'extérieur et ses points d'accès depuis l'extérieur. Ces interfaces regroupent un ensemble de méthode avec des signatures (nom de méthodes) bien définies. Certaines d'entre elles sont destinées à être implémentées par le médium (cf. figure 4.1 en page 95). Les autres peuvent être nécessaires au médium pour son fonctionnement, elles sont offertes par d'autres composants. Le médium exprime ainsi une dépendance envers les interfaces d'autrui. Tout ceci est connu et bien spécifié durant la phase identification et description.

Le connecteur possède aussi des interfaces d'accès et d'interaction. Cependant, il ne définit pas de services explicites désignés par des noms de méthode car il n'offre pas un service de communication. Ainsi, il n'implémente pas d'interfaces. De plus, il n'exprime pas de dépendances vers d'autres composants car il n'a pas besoin de services offerts par d'autres composants. C'est une entité véritablement autonome et indépendante. Il offre un moyen de relier et de connecter un composant (port) vers le service adéquat (un autre port) en assurant une propriété de communication. Au lieu d'être relié d'une façon fixe (par un appel explicite de service), le connecteur propose des emplacements vides et génériques qui permettent de contenir les interfaces des autres composants pour leur appliquer la propriété de communication. Ces interfaces implicites posent des contraintes liées à la sémantique de communication sur les futures interfaces qui vont s'y connecter. Elles peuvent être spécifiées par la grammaire d'un langage (le langage capable de comprendre les noms de signature des interfaces des composants). Les interfaces du connecteur, les prises, se concrétisent par adoption.

Différence d'implémentation

Au niveau implémentation, le médium suit un cycle classique de développement d'un composant. Il est conçu, implémenté, compilé et posé sur étagère comme un composant

binaire (exécutable) qui cache son implémentation et qui est prêt à être utilisé. Il représente l'implémentation de la sémantique de communication décrite au niveau architecture qui sera utilisée telle quelle au déploiement. Il possède un processus de raffinement classique mais son déploiement n'est pas habituel et n'est pas nécessairement atomique.

Le connecteur de son côté représente un élément d'architecture. Il est conçu puis implémenté comme un générateur. C'est ce dernier qui est déposé sur étagère pour utilisation. Le générateur ne représente pas l'implémentation de l'interaction décrite au niveau architecture. Il a pour tâche de collecter les interfaces des composants qui veulent interagir pour combler ses prises génériques, de s'assurer de la connectabilité en vérifiant que les interfaces collectées satisfont bien aux contraintes exprimées par les prises, et de produire le composant de liaison avec des interfaces explicites. Ce dernier représente l'implémentation de l'abstraction de communication décrite au niveau architecture appliquée aux composants interagissant. Par conséquent, les binaires (exécutable) du connecteur ne sont obtenus qu'après le processus de génération, ils ne sont pas déposés sur étagère comme dans le cas du médium.

Le médium spécifie des interfaces et la phase implémentation fournit le code de ces interfaces. Il n'existe alors qu'une seule implémentation de ces interfaces qui est celle du médium. Tous les composants définis dans l'architecture possèdent une implémentation unique de leurs interfaces. Le connecteur, quant à lui, ne spécifie pas d'interfaces et adopte les interfaces des composants interagissant. En plus de l'implémentation de ces interfaces par les composants eux-mêmes, le générateur fournit une implémentation particulière de ces interfaces dans les proxies générés. Ainsi, il existe une implémentation traditionnelle de ces interfaces effectuée par les composants interagissant eux-mêmes, et une seconde implémentation effectuée par les proxies du composant de liaison.

Différence d'utilisation

D'un point de vue utilisation, c'est-à-dire une fois l'assemblage réalisé, un composant fonctionnel utilise directement les services offerts par le médium. Il est prévu à sa conception qu'il appelle dans son code un service de communication offert par le médium. Il demande explicitement au médium de lui fournir le service de communication en étant explicitement connecté aux interfaces du médium. Dans ce cas, et comme pour l'interconnexion des composants fonctionnels, les services peuvent coïncider et aller bien ensemble, ou bien ne pas s'accorder, et dans ce dernier cas il faut utiliser des adaptateurs afin de pouvoir réaliser l'assemblage.

Par contre, un composant n'utilise pas les services d'un connecteur. Il utilise les services offerts par un autre composant fonctionnel offrant le service par l'intermédiaire du connecteur. Ce dernier doit assurer une abstraction de communication qui est transparente aux composants interagissant. Dans le cas où un problème de concordance entre les interfaces des composants fonctionnels surviendrait, le connecteur peut servir également d'adaptateur.

Il existe également des différences d'un point de vue utilisation pour l'obtention d'une application déployée. Les composants de communication sont mis à disposition sur étagère, prêts à être déployés avec les autres composants. Leurs binaires (exécutables) sont prêts à être utilisés tels quel. Les connecteurs, par contre, sont disposés sur étagère comme des générateurs. Le générateur n'est pas déployable en tant que tel, il doit être utilisé pour générer le composant de liaison avec tous ses éléments, où cet ensemble constitue les binaires (exécutables) déployables. Le générateur est utilisé pour automatiser et intégrer avec transparence la sémantique de communication pour une application sur une plate-forme. Il

fabrique ainsi un composant de communication sur mesure pour des composants spécifiques. Par ailleurs, un point en commun existe entre les deux entités : les binaires obtenus ne sont pas dépolysés comme des entités atomiques. Il existe des fragments de chaque entité sur différents sites.

Synthèse

Toutes ces différences d'usage et de fabrication que nous avons étudiées dans cette section montrent l'intérêt de distinguer la représentation des interfaces des connecteurs et des médiums par des ronds et des carrés qui représentent les interfaces implicites et explicites respectivement. Elles incitent à ne pas confondre toutes les interactions entre les composants en les représentant uniquement par des ellipses au niveau architecture. Cette différence de nature que nous introduisons est motivée, et se traduit principalement, par la différence dans le processus de fabrication de l'entité à mettre sur étagère et de son utilisation. Les différencier au niveau architecture, c'est-à-dire tôt dans le cycle de vie, influe sur la façon d'implémenter l'abstraction de communication et permet d'anticiper la suite du déroulement du processus de développement. La communication entre les différents intervenants dans ce processus est améliorée puisque chacune des entités possède ses spécificités qui guident la suite du développement. Ceci limite alors la probabilité de rencontrer des problèmes inattendus pendant le cycle de vie.

Les deux entités sont utiles et complémentaires, elles possèdent chacune leurs avantages et leurs inconvénients. D'un côté, le composant de communication possède un processus de développement classique des composants. Ce processus est largement acquis et maîtrisé dans la communauté des développeurs. Cependant, utiliser des entités toutes prêtes qui offrent des services figés peut nécessiter l'utilisation d'adaptateurs et de réaliser quelques retouches. D'un autre côté, les connecteurs offrent une transparence à la communication et une flexibilité considérable quant à la reconfiguration des applications grâce à leurs interfaces qui s'adaptent aux interfaces des composants. Ainsi, la communication n'est pas reconstruite depuis le début en la mélangeant avec les composants mais elle est produite automatiquement et sur mesure pour les composants de l'application. Cependant, le « sur mesure » des connecteurs peut engendrer des coûts relatifs à la compilation. De plus la construction d'un générateur peut s'avérer un peu plus complexe que la construction d'un composant de communication. Mais cette construction ne se fait qu'une seule fois et permet de faciliter la tâche des développeurs par la suite.

Nous allons voir dans la section suivante quelques éléments qui permettraient de choisir entre l'utilisation et la fabrication de ces deux entités.

4.2.2 Éléments de choix d'utilisation et de fabrication

L'utilisation et la création d'un composant de communication ou d'un connecteur sont considérées comme des choix d'architecture. Toutefois, nous pouvons donner quelques éléments qui peuvent guider les choix de fabrication et d'utilisation de ces entités. Pour cela, nous allons donner quelques critères qui permettraient de décider de la nature de l'entité qui va capitaliser l'abstraction de communication. Ces critères concernent principalement la dépendance à l'application. Il existe des dépendances par rapport à la fonctionnalité de l'application ou à son évolution.

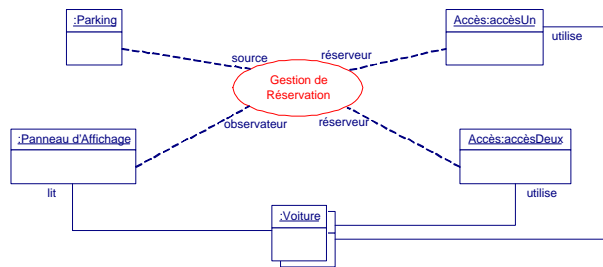


Figure 4.6 — Une application qui utilise un médium de réservation

Dépendance à la fonctionnalité

Il est possible de choisir de réifier une abstraction de communication suivant son lien avec l'application. Ainsi, dans le cas où il s'agit de réifier une interaction métier, il serait plus logique de réifier cette abstraction de communication comme un composant de communication, ou médium. En effet, les composants fonctionnels sur étagère peuvent être conçus pour appeler directement un service de communication offert par un médium qui traite des informations relatives à l'application. Ces informations à traiter sont également obtenues par le médium à partir des composants fonctionnels. Il est ainsi possible de créer des abstractions de communication très spécifiques à certains domaines. Cependant, ceci les rend moins réutilisables et limités à certains types d'application. Par contre, lorsque l'abstraction de communication est assez générique et peut se retrouver dans de nombreux types d'applications, il serait plus judicieux de la réifier comme un connecteur. Celui-ci pourra être ainsi largement réutilisé et intégré dans n'importe quelle application.

Afin d'illustrer cela, prenons exemple d'un médium de réservation tel qu'il est présenté dans [18]. Ce médium propose des places qui seront utilisées pour être réservées ou libérées, et qui seront également affichées pour montrer la disponibilité. Pour cela, ce médium possède trois rôles : **source** qui permet de récupérer le format des identificateurs des places, **reserveur** qui permet de réserver ou libérer les places, et **observateur** qui permet d'afficher les places. La figure 4.6 illustre cette collaboration dans une application de gestion de places de parking.

Dans cet exemple, la tâche de gestion des identificateurs des places est attribuée au médium. Celui-ci récupère leur type par le rôle **source**. Ce choix de conception permet la réutilisation du médium de réservation dans plusieurs applications et ne pas restreindre son utilisation à un type de places particulier. En cachant l'implémentation de la gestion de distribution, ce médium permet également de réaliser plusieurs variantes de la même implémentation : en centralisé pour gérer un petit nombre de places ou en distribué pour gérer un grand nombre d'objets réservés. Cela dit, l'utilisation de ce médium sera limitée aux applications qui ont besoin de cette fonctionnalité de réservation, comme la réservation de places de parking, de billets d'avions, de chambres d'hôtels, etc.

D'un autre côté, dans l'exemple de l'équilibrage de charge, comme l'abstraction de communication est indépendante de l'application, contenir cette propriété dans un connecteur semblait être la solution la plus adéquate. Par contre, si la métrique de l'équilibrage de charge se trouve fortement liée à l'application, il serait préférable de la réifier comme un composant de communication.

Dépendance à l'évolution

L'utilisation d'une abstraction de communication comme un composant de communication ou un connecteur pourrait être relatif à l'évolution de l'application par rapport à l'interaction. En effet, on pourrait choisir d'utiliser un composant de communication si l'application ne demande pas beaucoup d'évolution puisque les composants seront reliés d'une manière fixe. Ainsi, s'il n'y a pas lieu de modifier la sémantique de communication, il n'y a pas lieu de changer de composants de communication ni de modifier les composants fonctionnels qui utilisent ce composant de communication. Par contre, si l'application est susceptible d'évoluer, il serait préférable d'utiliser un connecteur. Avec ses prises génériques qui se spécialisent en fonction des interfaces des composants, le connecteur offre une meilleure flexibilité. Il facilite et favorise la reconfiguration des applications ainsi que la substitution de la sémantique de communication. En effet, le remplacement d'un connecteur par un autre connecteur réalisant une sémantique différente se fait avec transparence par rapport aux composants communicants. Il ne faut qu'une simple régénération et il n'est pas nécessaire d'intervenir au niveau des composants, ils ne sont pas modifiés.

Un autre critère de choix est d'utiliser un connecteur lorsque les propriétés générales de l'application évoluent et sont à décider par l'architecte. Les composants ne sont pas conçus pour communiquer avec le connecteur, donc il est possible d'intégrer ces propriétés avec transparence. Un composant a besoin d'un service et il doit l'obtenir quel que soit le moyen. C'est l'architecte qui fixe ce moyen en choisissant le connecteur qui répond à ces besoins et qui convient le mieux à l'application qu'il veut mettre en place. L'architecte doit chercher le connecteur qui permet de satisfaire ces propriétés. L'utilisation d'un connecteur implique que le moyen de communication est transparent. D'un autre côté, il est possible de choisir d'utiliser un médium lorsque la décision de communication est décidée par le composant fonctionnel, puisqu'il a été conçu pour utiliser explicitement les propriétés du médium.

4.3 Classification de quelques éléments de connexion

Après avoir dressé une liste non exhaustive sur les différences entre les composants de communication et les connecteurs et présenté quelques éléments de choix pour la réification des abstractions de communication et leur utilisation, nous allons présenter dans cette section quelques moyens de communication connus et habituels et les classer en composants de communication ou en connecteurs.

Fournisseur/Consommateur

Un modèle de communication appartenant à la famille des modèles de communication les plus connus et les plus utilisés pour relier des composants logiciels est le modèle de communication fournisseur/consommateur (*publish/subscribe*) [32]. C'est une technique efficace pour la construction d'applications distribuées où l'information doit être disséminée des fournisseurs (producteurs d'événement) vers les consommateurs (consommateurs d'événement).

Pour ce faire, les composants consommateurs doivent s'inscrire à un thème pour montrer leur intérêt et accepter de recevoir des messages relatifs à ce thème. Les fournisseurs doivent produire des événements et les envoyer à des consommateurs qu'ils ne connaissent pas. Ainsi, il existe un intermédiaire entre les composants fournisseurs et consommateurs qui s'occupe des inscriptions des consommateurs, de la collecte des messages et événement des fournisseurs,

et du tri de ces événements pour les renvoyer aux consommateurs intéressés.

Par abus de langage, cet intermédiaire est désigné comme un connecteur. Cependant, en l'examinant de plus près on voit bien que les composants ne se connaissent pas et ils s'adressent directement à cet intermédiaire :

- Les fournisseurs utilisent explicitement le service *publish()* offert par cet intermédiaire pour envoyer les messages ou les événements produits ;
- Les consommateurs utilisent le service *subscribe()* offert par cet intermédiaire pour s'inscrire et recevoir les informations qui les intéressent.

De ce fait, il s'agit bien d'un composant de communication et non pas d'un connecteur.

Canal/Filtre

Le modèle canal/filtre (*pipe & filter*) constitue un style d'architecture connu et très utilisés dans les environnements UNIX. Dans ce modèle, les filtres s'occupent des calculs et sont considérés comme étant des composants. Les canaux s'occupent de relier ces composants et sont souvent appelés connecteurs.

Dans ce style d'architecture, les composants (les filtres) sont autonomes et construits indépendamment les uns des autres. Ils sont conçus pour recevoir des flux de données, réaliser un traitement sur ces données, et produire un autre flux de données. Par exemple, un filtre peut recevoir en entrée une liste de fichiers, faire un calcul pour ne sélectionner que les fichiers qui dépassent une certaine taille, et produire une autre liste avec ces fichiers. Cependant, ce filtre n'a pas besoin de savoir d'où provient la liste des fichiers, et ne sait pas à qui sera adressée la liste de fichiers qu'il produit. C'est à l'architecte de construire l'application :

- Il doit choisir les composants qui fournissent les données d'entrée de ce filtre, et ceux qui reçoivent le résultat de sortie du filtre ;
- Ensuite, il doit les connecter aux extrémités des canaux pour que ces derniers puissent s'occuper du transfert de données. Ces canaux offrent un service explicite de transfert des données.

Donc, un composant filtre ne doit connaître que les interfaces explicitent des canaux (*pipes*) car ils correspondent à leurs interlocuteurs directs. Ainsi, un composant filtre ne doit connaître que les interfaces : *write()* qui est l'interface « puit » de données (*sink*) du canal pour écrire ses données, et *read()* qui est l'interface source du canal pour lire les données.

Suite à ces descriptions, le canal du modèle canal/filtre (*pipe & filter*) est plus assimilé à un composant de communication qui fait du transfert de données qu'à un connecteur.

CORBA

CORBA (*Common Object Request Broker Architecture*) [67] est un standard défini par l'OMG (*Object Management Group*) pour faire communiquer des objets dans des applications réparties. C'est une architecture unifiée qui offre un environnement logiciel pour supporter la

réutilisabilité et la portabilité des composants, ainsi que l'interopérabilité et l'hétérogénéité entre les différents langages et environnements informatiques (machine, OS). L'architecture CORBA comporte plusieurs éléments dont un courtier de requêtes (*Object Request Broker - ORB*), qui s'occupe de la localisation transparente des objets, du transfert des requêtes et le retour des résultats (c'est le canal de communication), et deux mécanismes d'invocation : statique et dynamique.

Le mécanisme d'invocation statique (*Static Invocation Interface - SII*) nécessite que les interfaces des objets requis soient connues à la compilation. Ces interfaces sont définies par un langage de description d'interface, l'IDL, et sont invoquées statiquement dans le code du client. Elles sont fournies aux générateurs de l'IDL pour une traduction (projection) vers plusieurs langages de programmation (IDL2JAVA, IDL2C++, ...) pour générer le *stub* du côté du client et le *skeleton* du côté du serveur. Ainsi, le mécanisme d'invocation statique n'offre pas des interfaces explicites mais utilise une interface cachée (avec l'opération *invoke()*) qui permet d'accéder à l'ORB et transférer les requêtes. Il correspond exactement à un connecteur.

Le mécanisme d'invocation dynamique (*Dynamic Invocation Interface - DII*) permet à un client d'accéder à de nouveaux objets qui ont été ajoutés et que le client ne connaissait pas au moment de la compilation. Ceci est réalisé par la création d'une requête dynamiquement. DII fournit une interface unique (standardisée) (APIs) qui permet au client de découvrir de nouveaux objets et leurs interfaces ainsi que de construire et distribuer des invocations. Ainsi, le client accède explicitement au service d'invocation dynamique pour lui demander de découvrir pour lui les interfaces à utiliser. Ce mécanisme offre un service de communication explicite, il peut donc être apparenté à un composant de communication.

CORBA fournit également un service d'événement appelé *Event Service* qui permet de découpler la communication entre les objets. Il définit deux rôles pour les objets : le rôle de producteurs (*supplier role*) et le rôle de consommateur (*consumer role*). Les producteurs produisent des données événementielles (*event data*) tandis que les consommateurs traitent ces données. Les données événementielles sont transmises entre les producteurs et les consommateurs en utilisant les mécanismes standards de CORBA. Pour ce faire, l'Event Service définit la notion de canal d'événement (*event channel*). Il s'agit d'un objet qui permet à plusieurs producteurs de communiquer avec plusieurs consommateurs de façon asynchrone. Un canal d'événement est à la fois un producteur et un consommateur d'événements. Il faut noter que les producteurs et les consommateurs n'ont pas besoin de se connaître, que les canaux d'événement sont des objets CORBA standards et les communications avec un canal d'événement est accomplie en utilisant les requêtes CORBA standards. Comme le modèle de communication « Fournisseur/Consommateur » que nous avons vu un peu plus haut, il s'agit ici également d'un composant de communication et les communications avec ce composant se font à l'aide d'un connecteur.

Pour conclure, on peut dire en général les services CORBA sont considérés comme des composants de communication dans notre classification.

Protocoles

Un protocole réseau peut être défini comme une méthode standard qui permet la communication entre des processus localisés sur des sites distants. Il s'agit d'un ensemble de règles et de procédures à respecter pour émettre et recevoir des données sur un réseau. Il en existe plusieurs selon ce qu'on attend de la communication. Certains protocoles sont

par exemple spécialisés dans l'échange de fichiers (ex. FTP - *File Transfer Protocol*), d'autre pourront servir à gérer simplement l'état de la transmission et des erreurs (ex. ICMP *Internet Control Message Protocol*). Sur Internet les protocoles utilisés font partie d'une pile de protocoles, c'est à dire un ensemble de protocoles reliés entre eux. Cette pile de protocoles constitue le modèle TCP/IP. Ce modèle est équivalent au modèle en couches OSI (*Open Systems Interconnection*) créé par l'organisation internationale de normalisation ISO (*International Standards Organisation*). La pile TCP/IP comporte 4 couches protocolaires :

- Couche Application : comporte l'ensemble des applications standards du réseau (Telnet, FTP, http, ...);
- Couche Transport : assure l'acheminement des données entre applications, ainsi que les mécanismes permettant de connaître l'état de la transmission (TCP, UDP);
- Couche Internet : chargée de fournir les paquets de données (IP et ses protocoles associés);
- Couche Accès réseau : elle spécifie la forme sous laquelle les données doivent être acheminées quel que soit le type de réseau utilisé.

Les protocoles sont souvent désignés comme étant des connecteurs. Cependant, dans une pile de protocole comme celle que nous venons de citer, chaque protocole d'une couche offre et utilise des interfaces explicites aux protocoles qui sont juste au-dessus ou juste en dessous dans la pile. Les protocoles offrent un service de communication et s'engagent à rendre ce service. Par exemple, le protocole TCP doit transformer un message en un segment et le protocole IP doit transformer ce segment en un paquet lors d'un envoi de message. Ils sont chargés des opérations inverses lors de la réception d'un message. Ainsi, le protocole TCP n'est pas sensé savoir que le protocole IP s'adresse à un autre composant en dessous pour pouvoir lui assurer le service. Un protocole est ainsi une boîte noire et on y accède à travers ses interfaces explicites. Dans notre classification, il doit être assimilé à un composant de communication non pas à un connecteur.

4.4 Synthèse

Nous avons distingué dans ce chapitre deux moyens différents pour réifier les communications entre les composants fonctionnels dans une application : les composants de communication, ou médiums, et les connecteurs. Les composants de communication sont avant tout des composants car ils offrent des interfaces explicites et ils ont un processus de développement équivalent. Cependant, ils possèdent quelques caractéristiques particulières qui les rendent différents des composants des autres modèles comme EJB et CCM : ils ont une fonctionnalité dédiée à la communication et leurs interfaces sont multilocalisées. Les connecteurs n'offrent pas de services explicites mais possèdent des interfaces implicites.

Cette différence de nature des interfaces entre les composants de communication et les connecteurs est nécessaire car elle conditionne le processus de développement et le reste du cycle de vie :

- D'un point de vue assemblage un composant de communication est utilisé par un accès direct et un appel explicite de ses interfaces. Les composants fonctionnels lui

Composant de communication	Connecteur
Interface fixe	Interface adaptable
Architecture a priori (Système fermé)	Architecture a posteriori (Système plus ouvert et dynamique)
Développement classique (cycle de vie)	Développé comme générateur (cycle de vie)
Plus visible	Plus transparent
Contrôlable par le composant (Configurable à la demande)	Moins contrôlable par le composant (configurable une fois)

Tableau 4.1 — Comparaison entre Composant de communication et Connecteur

demandent explicitement de leur rendre le service de communication. Par contre, dans le cas d'un assemblage avec un connecteur les composants ne demandent pas explicitement un service de communication mais un service offert par un autre composant. Le connecteur adopte les interfaces des autres composants afin de leur appliquer la propriété de communication. Ainsi, le connecteur rend la propriété de communication transparente aux composants interagissants alors qu'elle est plus visible dans le cas de l'utilisation d'un composant de communication.

- D'un point de vue réalisation, un composant de communication est réalisé puis déposé sur étagère comme un composant binaire (exécutable) pour être déployé avec les autres composants. Il possède alors un processus de développement classique. Par contre, un connecteur est réalisé comme un générateur et est déposé sur étagère en tant que tel. Les exécutables du connecteur ne sont obtenus qu'après l'activation du générateur et l'obtention du composant de liaison qui, lui, représente le composant binaire à déployer. La partie réutilisable d'un médium est le composant déposé sur étagère pour être déployé mais la partie réutilisable d'un connecteur est le générateur de composant déployable déposé sur étagère. Ainsi, bien qu'au final tout devienne composant au moment du déploiement, nous avons identifié des entités et des chemins différents qui mènent à ce déploiement. Le tableau 4.1 résume les principales différences entre un composant de communication et un connecteur.

Ces deux entités de communication sont complémentaires. Elles peuvent être combinées pour assurer ensemble la communication d'un côté, et peuvent être fusionnées pour former une entité hybride d'un autre côté.

Pour le premier cas, il est possible qu'un connecteur repose sur un composant de communication pour accomplir sa fonction de communication. En effet, le générateur associé au connecteur peut se servir des interfaces d'un composant de communication existant, qui est lié à une plate-forme par exemple, pour générer le composant de liaison. Nous pouvons citer comme exemple le protocole IIOP (*Internet Inter ORB Protocol*) qui est, en tant que protocole, un composant de communication utilisé par le générateur CORBA. Nous avons également utilisé cette technique pour la réalisation du générateur du connecteur d'équilibrage de charge adaptatif qui utilise un composant qui suit le modèle Fournisseur/Consommateur. D'un autre côté, dans le cas où la réification d'une abstraction de communication existe comme un composant de communication mais que ses interfaces ne coïncident pas avec celles des composants fonctionnels, le connecteur, avec son générateur, peut servir à générer la glu qui fait l'adaptation entre les interfaces. Le connecteur sert ainsi d'adaptateur pour le composant de communication. Comme il existe réellement plus de composants de communication que de connecteurs, cette méthode peut être utile pour économiser les coûts d'adaptation des composants de communication.

Pour le second cas, il est possible de combiner les connecteurs et les composants de communication pour avoir une entité hybride appelée `ComplexConnector` [45]. Il s'agit d'une entité qui contient à la fois des interfaces explicites (ports) et des interfaces implicites (prises). Un exemple concret est CORBA. Ce dernier regroupe les interfaces implicites du connecteur, avec le mécanisme d'invocation statique (SII), et les interfaces explicites, avec le mécanisme d'invocation dynamique (DII).

Conclusion

Contribution de la thèse

Nous avons présenté dans ce document notre proposition d'un processus de réification d'abstractions de communication sous forme d'un connecteur logiciel associé à des générateurs. Cette proposition constitue une contribution dans les domaines de l'architecture et du génie logiciel.

Dans le domaine de l'architecture logicielle, le principe de séparation entre les composants et leurs interconnexions est un concept bien accepté dans la communauté. Cependant, nous avons montré que la définition des interconnexions entre les composants est encore instable. De plus, bien que l'objectif du domaine de l'architecture logicielle est d'éviter de pousser le niveau de détail des descriptions, il existe des détails qui ne sont pas suffisamment traités, comme la nature des interfaces des interconnexions. Ceci rend la définition des concepts imprécise, crée des divergences dans leur considération, et laisse apparaître ainsi une multitude d'approches différentes. Nous supportons dans notre travail le principe de séparation entre les composants et leurs interconnexions mais nous conceptualisons les interconnexions au niveau architecture comme des *connecteurs*. Ces connecteurs sont différents des composants non seulement d'un point de vue fonctionnalité mais également d'un point de vue de leur nature. Nous donnons corps à ces connecteurs en définissant un processus qui les considère comme des entités distinctes dans quatre moments différents du cycle de vie.

La définition du cycle de vie constitue la contribution génie logiciel. Les grandes étapes du processus se traduisent par quatre formes de l'interaction qui sont :

- Un *connecteur* qui existe au niveau architecture comme une entité abstraite indépendante de toute plate-forme et de tout composant. Le connecteur doit assurer une propriété de communication et interagit avec les composants à travers des interfaces implicites appelées prises ;
- Une famille de *générateurs* qui intègrent la propriété du connecteur au-dessus de différentes plates-formes de mise en œuvre. Le mécanisme de génération constitue une bonne solution pour préserver le caractère abstrait des prises ;
- Une *connexion* lorsque le connecteur est assemblé avec les autres composants. À ce moment là, les prises se concrétisent en adoptant les services des composants interagissant et forment les interfaces de connexion ;
- Un *composant de liaison* lorsque le système est déployé. Ce composant de liaison représente une entité complètement concrétisée obtenue en joignant les interfaces de

connexion à l'un des générateurs.

La définition de ce cycle de vie répond à deux exigences du génie logiciel : la séparation des responsabilités et la séparation entre la description et l'implémentation. Le premier objectif est atteint grâce à la distinction entre un connecteur et une connexion. Le second objectif est atteint grâce à la distinction entre un connecteur et son générateur. Ce cycle de vie participe également à la précision du vocabulaire utilisé pour désigner les interactions entre les composants dans les différents moments du processus de développement : la description, l'implémentation, l'assemblage et le déploiement.

Nous avons validé le concept de connecteur et sa mise en œuvre sur un exemple d'équilibrage de charge. Nous avons transformé la plate-forme Jonathan d'un simple bus de transport faisant du transfert de données à distance entre deux composants en un connecteur capable de prendre des décisions de répartition de charge entre plusieurs composants. Cette plate-forme offre désormais une nouvelle abstraction de communication qui permet de connecter des composants. Avec cet exemple, nous avons montré l'intérêt de notre approche de générateur notamment par rapport à la flexibilité et l'adaptation. En effet, il est possible de changer de sémantique de communication, et ainsi de reconfigurer l'application, sans affecter les composants. Les connecteurs et les composants sont indépendants jusqu'au déploiement. Il est également possible d'interconnecter tout type de composants puisque les prises du connecteur s'adaptent aux interfaces de ces composants.

Le but du concept de connecteur est de remonter des abstractions de communication, habituellement considérées comme des détails de développement, à un niveau architecture en les réifiant. Ceci permet d'anticiper l'introduction des propriétés pour prévenir l'évolution et la maintenance du logiciel. Le savoir-faire est ainsi capitalisé pour être réutilisé dès que nécessaire. Ceci peut améliorer considérablement la qualité du logiciel en réduisant les coûts de re-développement. Nous encourageons donc l'adoption de cette approche de générateur pour la généraliser à la réalisation d'autres propriétés de connecteurs. La tendance actuelle fait que les propriétés complexes sont automatiquement réalisées comme des composants. Avec notre approche, par connecteur, nous diversifions les concepts et apportons le moyen de choisir s'il n'est pas plus avantageux de représenter ces propriétés comme un connecteur et d'offrir la transparence à la communication.

Perspectives

L'étude que nous avons menée dans cette thèse ouvre quelques voies d'investigation qui couvrent le processus de définition et d'implémentation des connecteurs.

Concernant le connecteur, nous le décrivons pour l'instant en langage naturel. Une perspective à cet effet serait de formaliser ces descriptions de connecteurs. En effet, la description des connecteurs à l'aide de langages formels apporterait plus de rigueur à notre approche. Cette spécification décrirait la propriété du connecteur pour approfondir son expressivité. Elle décrirait également les prises en formalisant leurs interfaces implicites par une grammaire qui permettra de faire collaborer plusieurs langages de définition d'interface. Cette grammaire pourrait couvrir les niveaux de contrat [11] syntaxique, sémantique, synchronisation et qualité de service en s'appuyant sur différents formalismes.

En approfondissant l'aspect formel, il serait possible également d'étudier l'intégration de plusieurs propriétés dans un connecteur. En effet, la propriété que doit assurer le connecteur

pourrait en englober plusieurs. Nous avons donné des exemples de connecteurs qui réifient les propriétés d'invocation (connecteur RPC), de choix (connecteur d'équilibrage de charge) ou de fusion (connecteur de consensus). Ces connecteurs assurent chacun une certaine qualité de service et peuvent être combinés comme nous l'avons fait pour le connecteur d'équilibrage de charge qui répartit des requêtes émises en RPC. Il serait ainsi envisageable de créer de nouveaux connecteurs qui combinent d'autres propriétés. Par exemple, on pourrait réaliser un connecteur qui combine l'équilibrage de charge et le consensus afin d'équilibrer la charge uniquement entre les serveurs fournissant des valeurs fiables. Ceci ouvre le champ à deux études possibles. D'un côté, identifier un moyen de classier les connecteurs pour les mettre à disposition. Il s'agirait d'étudier la meilleure disposition des connecteurs : à « plat », où chaque connecteur est considéré indépendamment, ou bien en « hiérarchie », où les connecteurs sont regroupés en famille. Il s'agit alors de trouver une frontière ou des limites entre les propriétés. D'un autre côté, il faudrait étudier la cohabitation des propriétés pour éviter les éventuels conflits qui peuvent survenir.

Un autre aspect qui pourrait étendre notre travail est la définition de connecteurs avec plusieurs prises. En effet, nous avons traité dans nos exemples des communications entre deux ensembles de composants de même service (connecteurs avec deux prises). Tester des connecteurs avec un nombre de prises supérieur à deux serait un bon complément. Ces connecteurs seraient utilisés pour exprimer une propriété de coordination entre plusieurs ensembles de connecteurs avec les mêmes services. Ils nécessitent ainsi d'exprimer une coordination complexe qui peut être décrite avec des langages de coordination. Il s'agira alors de réaliser des générateurs pour ces langages de coordination.

Hormis les aspects architecturaux, certains aspects relatifs à la connexion et à l'implémentation des générateurs peuvent être étendus. D'une part, concernant la connexion, une perspective est de réaliser des outils d'assemblage pour la réalisation de la relation d'adoption entre les prises et les interfaces des composants pour décrire les interfaces de connexion obtenues à l'assemblage. La validité de la connexion pourrait être assurée par l'utilisation des contrats exprimés par les prises et de ceux exprimés par les interfaces des ports. D'autre part, concernant l'implémentation des générateurs, ces derniers sont à présent développés manuellement sur une plate-forme donnée par un développeur. Ces générateurs automatisent l'intégration d'une propriété de communication sur une plate-forme pour des composants spécifiques. Ils permettent ainsi la prise en charge de détails complexes et répétitifs. Nous envisageons une étude plus approfondie des caractères communs des générateurs afin de permettre l'automatisation de la génération des générateurs en se basant sur des outils de transformation de modèles par exemple.

Bibliographie

- [1] G. D. ABOWD, R. ALLEN et D. GARLAN, *Using style to understand descriptions of software architectures*, ACM Software Engineering Notes, vol. 18, no. 5, pp. 9–20, December.
- [2] J. ALDRICH, C. CHAMBERS et D. NOTKIN, *Archjava : Connecting software architecture to implementation*, dans *Proceedings of the 24th International Conference in Software Engineering, ICSE'02*, Orlando, Florida, United States, May 2002.
- [3] J. ALDRICH, V. SAZAWAL, C. CHAMBERS et D. NOTKIN, *Language support for connector abstractions*, dans *European Conference on Object-Oriented Programming, ECOOP'03*, Darmstadt, Germany, July 2003.
- [4] R. ALLEN, *A Formal Approach to Software Architecture*, Thèse de doctorat, Carnegie Mellon Univ., May 1997.
- [5] R. ALLEN et D. GARLAN, *A formal basis for architectural connection*, ACM Trans. Software Eng. and Methodology, vol. 6, no. 3, pp. 213–249, July 1997.
- [6] S.-E. AMMOUR, A. BEUGNARD, S. MATOUGUI et B. TRAVERSON, *Architecture logicielle et abstractions de communication : une application dans le domaine du calcul scientifique*, dans *Les nouvelles technologies de la répartition (NOTERE'04)*, Saïdia, Maroc, 2004.
- [7] K. ARNOLD, J. GOSLING et D. HOLMES, *The Java Programming Language (3rd Edition)*, Addison-Wesley, June 2000.
- [8] J. BALASUBRAMANIAN, D. C. SCHMIDT, L. DOWDY et O. OTHMAN, *Evaluating the performance of middleware load balancing strategies*, dans *Proceedings of the 8th International IEEE Enterprise Distributed Object Computing Conference, EDOC'04*, Monterey, California, United States, September 2004.
- [9] L. BERGER, *Mise en œuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*, Thèse de doctorat, Université de Nice, October 2001.
- [10] G. D. BERGLAND, *A guided tour of program design methodologies*, IEEE Computer, vol. 14, no. 16, pp. 13–37, October 1981.
- [11] A. BEUGNARD, J.-M. JÉZÉQUEL, N. PLOUZEAU et D. WATKINS, *Making Components Contract Aware*, Computer, vol. 32, no. 7, pp. 38–45, July 1999.
- [12] A. BEUGNARD et R. OGOR, *Encapsulation of protocols and services in medium components to build distributed applications*, dans *Engineering Distributed Objects (EDO'99), an ICSE'99 Workshop*, Los Angeles, California, United States, May 1999.
- [13] G. BLAIR et J.-B. STEFANI, *Open Distributed Processing and Multimedia*, Addison Wesley, 1998.

- [14] T. BRISCO, *DNS Support for Load Balancing*, April 1995.
IETF RFC 1794, <http://www.ietf.org/rfc/rfc1794.txt>.
- [15] E. BRUNETON, *Julia Tutorial, version 2.0*, 2003.
<http://fractal.objectweb.org/tutorials/julia/>.
- [16] E. BRUNETON, *Developing with Fractal*, March 2004.
<http://fractal.objectweb.org/tutorial/>.
- [17] E. CARIOU, *Contribution à un Processus de Réification d'Abstractions de Communication*, Thèse de doctorat, Université de Rennes 1, école doctorale Matisse, June.
- [18] E. CARIOU, A. BEUGNARD et J.-M. JÉZÉQUEL, *An Architecture and a Process for Implementing Distributed Collaborations*, dans *The 6th IEEE International Enterprise Distributed Object Computing Conference, EDOC'02*, Lausanne, Switzerland.
- [19] D. CHAPPELL, *Au coeur de Activex et Ole*, Microsoft Press, September 1996.
- [20] D. CHAPPELL (CHAPPELL AND ASSOCIATES), *DCE and Objects*, March 1996.
http://www.opengroup.org/dce/info/dce_objects.htm.
- [21] P. CHATONNAY, B. HERRMANN et L. PHILIPPE, *Etude comparative d'algorithmes d'équilibrage de charge sur multicalculateurs*, dans *Proceedings of the SIPAR'95 Workshop*, Biel-Bienne, Switzerland, October 1995.
- [22] CISCO SYSTEMS INC., *High availability web services*, 2000.
http://www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/mnibm_wp.htm.
- [23] P. COAD et E. YOURDON, *Object-Oriented Analysis, Second Edition*, Yourdon Press, 1991.
- [24] G. COULOURIS, J. DOLLIMORE et T. KINDBERG, *Distributed Systems : Concepts and Design (3rd Edition)*, Pearson Education Limited, 2001.
- [25] T. COUPAY, E. BRUNETON et J. B. STEFANI, *The Fractal Composition Framework, Proposed Final Draft of Interface Specification version 0.7*, January 2002.
<http://arcad.essi.fr/020129/spec-framework.pdf>.
- [26] E. M. DASHOFY, N. MEDVIDOVIC et R. N. TAYLOR, *Using off-the-shelf middleware to implement connectors in distributed software architectures*, dans *Proceedings of the 21st international conference on Software engineering, ICSE'99*, (pp. 3–12), IEEE Computer Society Press, Los Angeles, California, United States, 1999.
- [27] F. DEREMER et H. KRON, *Programming in the large vs programming in the small*, IEEE Transactions on Software Engineering, vol. 2, no. 2, pp. 80–86, June 1976.
- [28] E. W. DIJKSTRA, *The structure of the t.h.e. multiprogramming system*, Communications of the ACM, vol. 11, no. 5, pp. 341–346, 1968.
- [29] F. DOUGLIS et J. OUSTERHOUT, *Process migration in the sprite operating system*, dans *Proceedings of the 7th International Conference on Distributed Computing Systems, ICDCS'87*, (pp. 18–25), Berlin, WestGermany, September 1987.
- [30] S. DUCASSE et T. RICHNER, *Executable connectors : Towards reusable design elements*, dans *In 6th European Software Engineering Conference / 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, ESEC/FSE'97* (édité par S. VERLAG), vol. 1301 de *Lectures Notes in Computer Science - LNCS*, (pp. 483 – 500), Zurich, Switzerland, September 1997.
- [31] W. EMMERICH, *Engineering Distributed Objects*, Wiley.

- [32] P. T. EUGSTER, P. A. FELBER, R. GUERRAOUÏ et A.-M. KERMARREC, *The many faces of publish/subscribe*, ACM Computing Surveys, vol. 35, no. 2, pp. 114–131, 2003.
- [33] D. GARLAN, *Software architecture : a roadmap*, dans *The Future of Software Engineering, 22nd International Conference on Software Engineering, ICSE'00*, (pp. 93–101), June 2000.
- [34] M.-. C. GAUDEL, *Formal specification techniques*, dans *Proceedings of the 16th International Conference on Software Engineering, ICSE'94*, (pp. 223–232), Sorrento, Italy, May 1994.
- [35] M.-. C. GAUDEL, B. MARRE, F. SCHLIENGER et G. BERNOT, *Précis de Génie Logiciel*, Masson, 1996.
- [36] A. GEORGIN, F. LEGOND-AUBRY, S. MATOUGUI, N. MOTEAU, A. MULLER, A. TAUVÉRON, J.-P. THIBAUT et B. TRAVERSON, *Description des assemblage et des contrats pour la conception par composants (le projet accord)*, dans *Journées Composants*, Lille, France, March 2004.
- [37] M. GUDGIN, M. HADLEY, N. MENDELSON, J.-J. MOREAU et H. F. NIELSEN (ED.), *SOAP Version 1.2 Part 1 : Messaging Framework*, June 2003.
W3C Recommendation, <http://www.w3.org/TR/soap12-part1/>.
- [38] J. GUTTAG et B. LISKOV, *Abstraction and Specification In Program Development*, The MIT Press/Mc Graw-Hill, 1986.
- [39] C. A. R. HOARE, *Communicating Sequential Processes*, Prentice-Hall, 1995.
- [40] C.-C. HUI et S. T. CHANSON, *Improved strategies for dynamic load balancing*, IEEE Concurrency, vol. 7, no. 3, pp. 58–57, July 1999.
- [41] INPRISE CORPORATION INC., *VisiBroker for Java 4.0 : Programmer's Guide : Using the POA*, 1999.
<http://www.inprise.com/techpubs/books/vbj/vbj40/programmers-guide/poa.html>.
- [42] IONA TECHNOLOGIES, *Orbix 2000*, 2000.
<http://www.iona.com/products/orbix/welcome.htm>.
- [43] E. JOHNSON et A. COMMUNICATIONS, *A Comparative Analysis of Web Switching Architectures*, 1998.
http://www.arrowpoint.com/solutions/white_papers/ws_archv6.html.
- [44] T. KIELMANN, *Designing a Coordination Model for Open Systems*, dans *Coordination Languages and Models* (édité par S. VERLAG), Lecture Notes in Computer Science 1061, 1996.
- [45] LE PROJET ACCORD, *Assemblage de Composants par Contrats en environnement Ouvert et Réparti*, 2001-2003.
<http://www.infres.enst.fr/projets/accord/>.
- [46] M. LINDERMEIER, *Load management for distributed object-oriented environments*, dans *Proceedings of the 2nd International Symposium on Distributed Objects and Applications, DOA'00*, Antwerp, Belgium, September 2000.
- [47] D. LUCKHAM, J. KENNEY, L. AUGUSTIN, J. VERA, D. BRYAN et W. MANN, *Specification and analysis of system architecture using rapide*, IEEE Trans. Software Eng., vol. 21, no. 4, pp. 336–355, April 1995.
- [48] D. LUCKHAM et J. VERA, *An event-based architecture definition language*, IEEE Trans. Software Eng., vol. 21, no. 9, pp. 717–734, September 1995.

- [49] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER, *Specifying distributed software architectures*, dans *Fifth European Software Eng. Conf., ESEC '95* (édité par S. VERLAG), Lecture Notes in Computer Science 989, Siges, Spain, September 1995.
- [50] J. MAGEE, N. DULAY et J. KRAMER, *Regis : A constructive development environment for distributed programs*, IEEE Distributed Systems Engineering Journal, vol. 1, no. 5, pp. 304–312, September 1994.
- [51] J. MAGEE et J. KRAMER, *Dynamic structure in software architectures*, dans *Proceedings of ACM SIGSOFT Fourth Symp. Foundations of Software Eng., FSE'96*, (pp. 3–14), San Francisco, California, United States, October.
- [52] S. MATOUGUI et A. BEUGNARD, *How to implement software connectors ? a reusable, abstract and adaptable connector*, dans *Distributed Applications and Interoperable Systems (DAIS'05)*, Athens, Greece, 2005.
- [53] S. MATOUGUI et A. BEUGNARD, *Two ways of implementing software connection among distributed components*, dans *Distributed Objects and Applications (DOA'05)*, Agia Napa, Cyprus, 2005.
- [54] N. MEDVIDOVIC, P. OREIZY, J. ROBBINS et R. TAYLOR, *Using object-oriented typing to support architectural design in the c2 style*, dans *Proceedings of ACM SIGSOFT Fourth Symp. Foundations Software of Eng., FSE'96*, (pp. 24–32), San Francisco, California, United States, October 1996.
- [55] N. MEDVIDOVIC, D. ROSENBLUM et R. TAYLOR, *A language and environment for architecture-based software development and evolution*, dans *21st Int'l Conf. Software Eng., ICSE'99*, (pp. 44–53), Los Angeles, California, United States, May 1999.
- [56] N. MEDVIDOVIC et R. N. TAYLOR, *A classification and comparison framework for software architecture description languages*, dans *IEEE Transaction on Software Engineering*, vol. 26, (pp. 70–93), January 2000.
- [57] N. MEDVIDOVIC, R. N. TAYLOR et E. J. WHITEHEAD, *Formal modeling of software architectures at multiple levels of abstraction*, dans *Proceedings of the 1996 California Software Symposium*, Los Angeles, California, United States, April 1996.
- [58] N. R. MEHTA, N. MEDVIDOVIC et S. PHADKE, *Towards a taxonomy of software connectors*, dans *the 22nd International Conference on Software Engineering, ICSE'00*, (pp. 178–187), Limerick, Ireland, June 2000.
- [59] B. MEYER, *Applying "design by contract"*, IEEE Computer (Special Issue on Inheritance & Classification), vol. 25, no. 10, pp. 40–52, October 1992.
- [60] B. MEYER, *Object-Oriented Software Construction, Second Edition*, Prentice-Hall, 1997.
- [61] P. NAUR et B. RANDALL (ED.), *Software Engineering*, NATO Scientific Affairs Division, January 1969.
- [62] J. R. NESTOR et D. L. STONE, *Idl : Background and status, special issue on the interface description language idl*, ACM SIGPLAN Notices, vol. 22, no. 11, pp. 5–9, November 1987.
- [63] OBJECTWEB, *the ObjectWeb Consortium*.
<http://www.objectweb.org/>.
- [64] OBJECTWEB OPEN SOURCE MIDDLEWARE, *Jonathan : an Open Distributed Objects Platform*.
<http://jonathan.objectweb.org/>.
- [65] OMG, *ORB FAQs - ORB Basics : How does CORBA support load-balancing ?*.
http://www.omg.org/gettingstarted/orb_basics.htm/ .

- [66] OMG, *OMG Unified Modeling Language Specification, version 1.3*, 1999.
<http://www.omg.org/cgi-bin/doc?ad/99-06-08>.
- [67] OMG, *CORBA specifications*, 2003.
http://www.omg.org/technology/documents/corba_spec_catalog.htm.
- [68] OMG, *UML 2.0 OCL 2nd revised submission, version 1.6, January 6, 2003*.
<http://www.omg.org/cgi-bin/doc?ad/2003-01-07>.
- [69] OMG, *CORBA Component Model RFP*, November.
<http://www.omg.org/docs/orbos/97-05-22.pdf>.
- [70] O. OTHMAN, C. O'RYAN et D. SCHMIDT, *The design of an adaptive corba load balancing service*, IEEE Distributed Systems Online, vol. 2, no. 4, April 2001.
- [71] M. OUSSALAH, *Ingénierie des composants logiciels : principes et fondements*, Vuibert, June 2005.
- [72] M. OUSSALAH et AL, *Ingénierie Objet : Concepts et techniques*, InterEditions, 1997.
- [73] M. C. OUSSALAH, A. SMEDA et T. KHAMMACI, *Software connectors reuse in component-based systems*, dans *IEEE International Conference on Information Reuse and Integration*, (pp. 543–550), 2003.
- [74] D. E. PERRY et A. L. WOLF, *Foundations for study of software architecture*, ACM Software Engineering Notes, vol. 17, no. 4, pp. 40–52, October 1992.
- [75] R. PRIETO-DIAZ et J. NEIGHBORS, *Module interconnection languages*, Journal of Systems and Softwares, vol. 6, no. 4, pp. 307–334, November 1986.
- [76] J. R. PUTMAN, *Architecting with RM-ODP*, Prentice Hall, October 2000.
- [77] W. E. RIDDLE et J. C. WILEDEN, *Tutorial on Software System Design : Description and Analysis*, Computer Society Press, 1980.
- [78] D. T. ROSS et J. K. E. SCHOMAN, *Structured analysis for requirements definition*, IEEE Transactions on Software Engineering. Special collection on Requirement Analysis, vol. 3, no. 1, pp. 6–15, January 1977.
- [79] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY et W. LORENSEN, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [80] J. RUMBAUGH, I. JACOBSON et G. BOOCH, *UML 2.0 Guide de référence*, Campus Press, 2004.
- [81] M. SHAW, R. DELINE, D. KLEIN, T. ROSS, D. YOUNG et G. ZELESNIK, *Abstractions for software architecture and tools to support them*, IEEE Trans. Software Eng., vol. 21, no. 4, pp. 314–335, April 1995.
- [82] M. SHAW, R. DELINE et G. ZELESNIK, *Abstractions and implementations for architectural connections*, dans *Third Int'l Conf. Configurable Distributed Systems*, May 1996.
- [83] M. SHAW et D. GARLAN, *Software Architecture : Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
- [84] J. SIEGEL, *Corba 3 Fundamentals and Programming (2nd Edition)*, John Wiley & Sons, 2000.
- [85] F. SPIES, *Conception et évaluation de stratégies de répartition de charge Dynamique dans les systèmes distribués*, Thèse de doctorat, Université de Franche-Comté, December 1994.
- [86] SUN MICROSYSTEMS INC., *Enterprise java beans technology*.
<http://java.sun.com/products/ejb/>.

- [87] C. SZYPERSKI, *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley, 1999.
- [88] A. VAN LAMSWEERDE, *Formal specification : a roadmap*, dans *The Future of Software Engineering, the 22nd International Conference on Software Engineering, ICSE'00*, (pp. 247–160), IEEE Computer Society Press / ACM Press, Limerick, Ireland, June 2000.
- [89] J. WARMER et A. KLEPPE, *The Object Constraint Language : Precise Modeling with UML*, Addison-Wesley, 1998.