RAPPORT DE STAGE

Assemblage de Composants et Programmation par Aspects

Frédéric Loiret

Responsables :
LIONEL SEINTURIER, MDC - LIP6
ERIC GRESSIER, MDC - CNAM

Laboratoire d'Informatique de l'Université Paris VI
Centre d'Études et de Recherche en Informatique du CNAM

Table des matières

1 Introduction 8

I Les notions de base 10

2 La notion de composant 12
2.1 Définitions d'un composant 12
2.2 Un modèle général de composant 12

3 La séparation des préoccupations 17
3.1 Définitions et difficultés 17
3.2 Ambiguïté 18

4 D'autres notions 19
4.1 Définition d'un framework (canevas) 19
4.2 Le Design Pattern Factory 20
4.3 Méta-objet, intercepteurs 21

II Présentation des plates-formes 26

5 Présentation de FRACTAL 28
5.1 Le framework FRACTAL 28
5.2 Le modèle de composant 29
5.2.1 Le modèle général 29
5.2.2 Le modèle concret 30
5.3 L'architecture du framework FRACTAL 35
5.3.1 Le noyau du framework 35
5.3.2 Les incrémentations du framework 37
5.4 Exemple d'implémentation 38
5.4.1 Formulation d'un exemple 38
5.4.2 Les étapes de la construction 39
5.4.3 La reconfiguration dynamique 41
5.4.4 Utilisation de l'ADL FRACTAL 42
5.5 Architecture de JULIA 43

6 Présentation de Kilim 44

6.1 Les motivations . . . 44

6.2 Les objectifs de Kilim . . . 44

6.3 Le modèle de composant et les concepts de Kilim . . . 45

6.3.1 Les `ports`, `providers` et les `transformers` . . . 45

6.3.2 `slots` et assemblage de composants . . . 45

6.3.3 La notion de `template` . . . 46

6.4 Exemple d'implémentation . . . 47

6.4.1 Les étapes de la construction . . . 48

6.4.2 La reconfiguration dynamique . . . 49

6.5 Architecture de Kilim . . . 50

7 Présentation de Jac 52

7.1 La programmation par aspect . . . 52

7.2 Mise en œuvre de la programmation par aspect . . . 54

7.3 Les objectifs de Jac . . . 54

7.4 Le modèle de composant . . . 54

7.5 Les étapes de la programmation . . . 56

7.5.1 Le niveau programmation . . . 57

7.5.2 Le niveau configuration . . . 58

7.6 Exemple d'implémentation . . . 58

7.6.1 Les étapes de la construction . . . 58

7.6.2 La reconfiguration dynamique . . . 61

7.7 Architecture de Jac . . . 61

III Implémentations 64

8 L'algorithme de Kai Li et Paul Hudak 66

8.1 Contexte et hypothèses . . . 66

8.2 Les alternatives possibles de l'algorithme . . . 66

9 Présentation de l'algorithme centralisé 68

9.1 Les structures de données . . . 68

9.2 Déroulement du protocole . . . 68

9.2.1 L'algorithmie des différentes fonctions . . . 69

9.2.2 Schématisation du déroulement du protocole . . . 70

9.2.3 Simplifications de l'algorithme . . . 70

10 Première implémentation 74

10.1 Contexte et hypothèses 74

10.1.1 Contexte général 74

10.1.2 Choix et simplifications au niveau de l'implémentation 75

10.1.3 Les échanges RMI 77

10.2 La séparation des préoccupations 78

10.3 Version implémentée en JAVA 80

10.3.1 Modélisation 81

10.3.2 Remarques sur l'implémentation 81

10.3.3 Exécution et déploiement 83

10.4 Version implémentée avec FRACTAL 83

10.4.1 Le modèle d'implémentation 83

10.4.2 Le modèle de composition 85

10.4.3 Remarques sur l'implémentation 86

10.4.4 Remarques sur l'assemblage 87

10.4.5 Exécution et déploiement 88

10.5 Version implémentée avec KILIM 88

10.5.1 Le modèle d'implémentation 88

10.5.2 Le modèle de composition 89

10.5.3 Remarques sur l'assemblage 90

10.5.4 Exécution et déploiement 90

10.6 Version implémentée avec JAC 91

10.6.1 Le modèle d'implémentation 91

10.6.2 Le modèle de composition 92

10.6.3 Remarques sur l'implémentation 93

10.6.4 Remarques sur l'assemblage 95

10.6.5 Exécution et déploiement 95

11 Présentation de l'algorithme décentralisé 96

11.1 Les structures de données 96

11.2 Déroulement du protocole 97

11.2.1 L'algorithmie des différentes fonctions 97

11.2.2 Schématisation du déroulement du protocole 99

12 Deuxième implémentation 102

12.1 Contexte et hypothèses . . . 102

12.2 La séparation des préoccupations . . . 102

12.3 Modélisation liée aux différentes fonctionnalités . . . 105

12.3.1 Schématisation de la modélisation . . . 105

12.3.2 Fil d'exécution . . . 105

12.4 Transposition de la modélisation . . . 108

12.4.1 Modèle d'assemblage de composants techniques . . . 108

12.4.2 Modèle de composition d'aspects . . . 109

12.4.3 Modèles d'implémentations liés aux différentes plates-formes . . . 110

12.5 Version implémentée avec FRACTAL . . . 110

12.5.1 Le modèle de composition . . . 110

12.5.2 Remarques liées à la plate-forme . . . 111

12.6 Version implémentée avec KILIM . . . 112

12.6.1 Le modèle de composition . . . 112

12.6.2 Remarques liées à la plate-forme . . . 113

12.7 Version implémentée avec JAC . . . 113

12.7.1 Le modèle de composition . . . 113

12.7.2 Remarques liées à la plate-forme . . . 115

IV Autres implémentations et comparaisons 122

13 D'autres implémentations ... 124

13.1 La composition répartie . . . 124

13.1.1 Avec FRACTAL . . . 124

13.1.2 Avec KILIM . . . 125

13.1.3 Avec JAC . . . 125

13.2 Evolutivité : cohérence causale . . . 125

14 Remarques, comparaisons, conclusions **128**

14.1 Comparaisons des points du modèle . . . 128

14.1.1 FRACTAL . . . 128

14.1.2 KILIM . . . 129

14.1.3 JAC . . . 131

14.2 Tableau récapitulatif . . . 133

14.3 Les étapes du développement . . . 134

14.3.1 Avec FRACTAL . . . 135

14.3.2 Avec Kilim . . . 137

14.3.3 Avec Jac . . . 138

14.4 Impact de la séparation des préoccupations . . . 139

14.5 Dispersion de code . . . 140

14.6 Remarque de dernière minute . . . 141

14.7 La vision de l'application . . . 141

14.7.1 Avec FRACTAL . . . 142

14.7.2 Avec KILIM . . . 142

14.7.3 Avec JAC . . . 142

14.7.4 Dans une équipe de développement . . . 143

14.8 Nombre de lignes de code . . . 144

14.9 Performances . . . 146

15 Conclusion **148**

Table des Figures **150**

Glossaire **152**

Index **154**

Bibliographie **156**

1 Introduction

La construction d'applications (réparties) est une tâche complexe : de nombreux paramètres entrent en ligne de compte et de nombreux services doivent être intégrés. Récemment, les approches fondées sur les composants (EJB, CCM, .NET) ont fait progresser l'écriture des applications. Parallèlement, plusieurs approches telles que FRACTAL, KILIM ou JAC s'intéressent à l'assemblage de composants logiciels et/ou à l'intégration de services systèmes au sein d'applications middlewares. Le domaine est émergent et très prometteur même si de maints travaux restent à réaliser.

Les objectifs de ce stage sont multiples. Nous avons cherché à comparer ces différentes plates-formes de construction d'applications : FRACTAL, basée sur un mécanisme d'assemblage de classes avec des notions d'interfaces clientes et serveurs, KILIM permettant de configurer des assemblages de classes et JAC, un framework (canevas) de programmation par aspects (AOP) réalisant une liaison dynamique entre des objets métiers et des objets d'aspect. FRACTAL et KILIM sont des approches basées sur des modèles de composants techniques, JAC est basé sur le paradigme de programmation par aspects. Chacune de ces approches propose son propre modèle de composition de briques logicielles que nous avons donc cherché à comparer.
Nous nous sommes également concentrés sur la notion de séparation des préoccupations (SoC) au sein de la conception d'une application. Il s'agit d'un principe élaboré très tôt dans l'histoire du génie logiciel, mais nous nous rendrons compte que sa mise en œuvre peut parfois se révéler complexe dans la pratique. Notre travail aura été de transposer notre vision de la séparation des fonctionnalités d'une application sur les trois plates-formes concernées.
D'un point de vue plus pragmatique, nous nous sommes focalisés sur les points essentiels des différents paradigmes de programmation à appréhender dans l'étape d'implémentation d'une application.
Un point très important et transverse à tous ces objectifs concerne l'appréhension et la gestion de la répartition.

La première partie (cf. I page 10) du rapport a pour but principal de présenter les points essentiels inhérents à tous modèles de composants ainsi que

la notion de séparation des préoccupations.

La deuxième partie (cf. II page 26) s'attache à présenter de manière indépendante les trois approches que nous avons été amené à étudier.

Le modèle d'application répartie que nous avons décidé d'implémenter est un algorithme de gestion d'une mémoire virtuelle répartie partagée proposé initialement par KAI LI et PAUL HUDAK. La troisième partie de ce rapport se focalise donc sur la présentation de deux alternatives de cet algorithme et sur leurs implémentations sur les plates-formes FRACTAL, KILIM et JAC (cf. III page 64).

Nous terminerons cette étude par une comparaison entre nos trois approches (cf. IV page 122).

Première partie

Les notions de base

Dans ce stage, notre objectif est de comparer différentes approches d'assemblage de briques logicielles dans l'optique de séparer les fonctionnalités dans la construction d'une application répartie. Dans cette première partie, nous nous attardons donc sur certaines explications ou définitions relatives aux notions de composants logiciels et de séparation des préoccupations au sein d'une application. Nous donnons également quelques explications sur certaines notions en marge de notre travail mais dont la compréhension est nécessaire pour la suite.

2 La notion de composant

2.1 Définitions d'un composant

Actuellement, il n'existe pas de définition très formelle du composant. Toutefois, plusieurs définitions ou caractérisations sont acceptées.

Ainsi, c'est de la façon suivante que Jed Harris a donné l'une des premières définitions d'un composant en 1995 [MP02] :

> « *Un composant est un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support. De plus, il est doté d'interfaces standards pour pouvoir interopérer.* »

Puis, en 1996, lors de la première édition du workshop on Component Oriented Programing, les participants ont convenu de la définition suivante [MP02] :

> « *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.* »

2.2 Un modèle général de composant

Comme nous pouvons le constater, le concept de composant logiciel est assez flou. De plus, il est sans doute peu probable que la communauté système s'accorde un jour à normaliser le concept de composant étant donné qu'il est difficile de prendre en compte tous les besoins des applications et des usagers au sein d'un modèle commun.
Il est également important de faire une distinction entre la définition de composant en tant qu'entité (dans le sens brique de logiciel complètement autonome et "autiste") et sa définition au sein du paradigme de programmation orientée composant. En effet, dans le premier cas, on peut définir le composant comme une extension du concept d'objet. Dans le deuxième cas, le composant est considéré comme faisant partie intégrante de son environnement, dans un contexte précis. On cherchera alors à définir un modèle plus ambitieux, où des notions et fonctionnalités moins intuitives de génie logiciel seront mises en avant, souvent remplies par une plate-forme d'exécution spécifique (comme par exemple la gestion de propriétés non-fonctionnelles d'un système déléguée à un conteneur* ou structure d'accueil - persistance, gestion des communications ...

etc). La définition d'un modèle dépend donc du référentiel dans lequel on se place.

Le but de cette section n'est donc pas de définir un nouveau modèle générique et exhaustif de composant, mais plutôt de tenter de caractériser de manière non ambigüe certains points inhérents à tous modèles. L'objectif est également de définir des critères d'évaluation dans le but de comparer nos trois plates-formes. Nous nous sommes sensiblement inspiré des modèles généraux donnés dans [MP02] et [BCS02] .

Les points sont donc les suivants :

1. **Encapsulation, abstraction et identité**

   L'encapsulation signifie qu'un composant doit interagir avec son environnement uniquement par l'intermédiaire de points d'accès et d'opérations bien définies, et qu'un éventuel changement de l'état d'un composant ne doit donc se faire que par ces points d'entrées (on parlera du concept d'interface, hérité de la POO*, définie de manière indépendante à toute implantation). L'abstraction signifie qu'un composant ne doit pas révéler plus de détails sur son implémentation qu'il n'en faut aux entités avec lesquelles il doit interagir. L'identité, point le plus important, signifie qu'un composant doit être suffisamment bien défini pour qu'aucune ambiguité ne soit levée quant à le distinguer d'un autre. En ce sens, on cherche donc à donner une définition abstraite de l'entité logicielle. Dans [MP02], R.Marvie et MC.Pellegrini parlent de type de composant, caractérisé par trois éléments : (a) ses interfaces, (b) les modes de coopération avec les autres types de composants et (c) ses propriétés configurables. De plus, il est possible de fournir plusieurs implantations pour un même type de composant ou encore de substituer deux implantations d'un même type de composant.

   - Les interfaces fournies par le type de composant définissent les services fournis par le composant, en énumérant les signatures des méthodes fournies, ainsi que les différents paramètres entrants et/ou sortants du composant. Elles correspondent donc aux points d'entrée du composant spécifiés plus haut. Les interfaces requises par le type de composant définissent les services que requiert le composant pour pouvoir être opérationnel.Ce concept d'interfaces requises est un apport significatif par rapport au paradigme de programmation orientée objet. En effet, dans une approche composant, une description d'architecture logicielle

est utilisée comme technique de remplacement des références (vers les services requis) codées en dur dans une approche objet. Ces interfaces permettent donc de gérer les connexions entre types de composants, et de spécifier de manière explicite les dépendances entre les entités nécessaires à la phase de composition.
- Chaque interface doit définir un mode de communication avec les autres types de composants. Le plus communément utilisé est le mode synchrone (comme par exemple l'invocation de méthode), mais il peut être judicieux qu'un modèle de composant puisse tolérer différentes sémantiques de communication, comme le mode asynchrone (par exemple par invocation de type *oneway*, ou par diffusion d'événements via des sources et des puits que l'on trouve dans CCM ou COM+), le mode de diffusion en continu (comme les flots de données), ou d'autres modes plus exotiques, comme le multicast, la gestion de la QoS, et éventuellement d'autres support qu'IP (comme l'ATM).
- Un composant est conçu et développé indépendamment des applications qui l'utilisent, et doit donc être adaptable en fonction de l'application cible et configurable pour caratériser le comportement d'une instance de composant dans un contexte donné. Un ensemble de propriétés configurables doit donc faire partie intégrante du composant pour satisfaire ces contraintes d'ordre applicatif ou d'ordre système. Bien que la configurabilité soit un point important pour différencier les types de composants, nous préférons l'évoquer plus tard.

2. **Composabilité**

   Un composant est une unité d'assemblage et de composition ; il n'est considéré comme viable que s'il devient une unité de composition pour plusieurs applications. Un modèle de composants doit permettre de composer ou d'assembler dynamiquement plusieurs composants de manière à pouvoir créer des entités de plus haut niveau. Cette notion de composabilité est au cœur de la notion de composant et sera l'un des critères majeurs qui nous permettra de comparer nos trois plates-formes. Autant que possible, le modèle ne doit pas prescrire une manière unique d'assembler les composants, mais doit pouvoir supporter différentes sémantiques de composition.
   Une autre manière d'exprimer le besoin de composabilité est de considérer que les relations entre composants doivent être explicitement maintenues, imposées et modifiables au moment de l'exécution de l'application. On peut également attendre d'un modèle de composant que la sémantique d'assemblage des entités soit vérifiée de manière à s'assurer de la bonne cohérence de cet assemblage.

3. **Caractéristiques non-fonctionnelles des composants**

(a) Contrôle de comportement, (re)configurabilité

On entend par contrôle de comportement les capacités à créer différentes forme de composants que l'on nommera contrôleur, i.e. des composants susceptibles de fournir différentes capacités d'introspection*. L'introspection est une des caractéristiques communes aux modèles de composants. Il s'agit d'un mécanisme pouvant aller de la capacité à interroger un composant pour découvrir dynamiquement les méthodes proposées par ses interfaces jusqu'à la capacité que possède le composant à raisonner sur son propre état pour modifier son comportement dynamiquement. L'objectif du contrôle de comportement est donc de surveiller un ensemble de composants et d'exercer un contrôle sur leur execution. On retrouve ici la notion de méta-traitement [Teb02].
Nous avons déjà évoqué l'importance de la configuration d'un composant dans la construction d'application pour caratériser un type de composant. Cependant, dans un modèle de composants, il peut être judicieux de pouvoir différencier la configurabilité des aspects fonctionnels du composant de la configurabilité des aspects non fonctionnels. Même si l'objectif est de permettre une grande adaptabilité d'un composant, ces deux niveaux de configurabilité n'agissent pas du tout sur le même plan. La configurabilité des aspects fonctionnels se pensera au moment de l'étape de conception de l'application et les mécanismes pour l'assurer seront axés sur le langage d'implantation (on cherchera à fixer les valeurs initiales des propriétés du composant, par exemple par positionnement de certains attributs ou par invocation de traitements pour fixer un état initial). Par contre, les capacités de configurabilité des aspects non fonctionnels vont être plutôt dépendants de la manière dont a été conçue la plate-forme d'accueil. Il faut noter que la phase de configuration d'un composant est indépendante de la phase d'instanciation, dans le sens où différentes instances d'un même composant peuvent être configurées de manières complètement différentes. De plus, on peut être amené à reconfigurer une instance de composant arbitrairement et dynamiquement.

(b) Notions dépendantes de la plate-forme d'accueil

Nous pouvons ici considérer qu'elles sont de trois types : (b1) la notion de cycle de vie, (b2) la manipulation des activités et (b3) la flexibilité de déploiement (éventuellement en environnement réparti).
La notion de cycle de vie est liée à la notion d'instance de compo-

sant (ie. au même titre qu'une instance d'objet, une entité existante s'exécutant sur un système). Un modèle de composant se doit donc de gérer de manière transparente différentes formes et différentes envergures de cycle de vie, incluant différentes formes d'instanciation, d'initialisation, d'activation, de suspension ou encore de terminaison ... D'après [MP02], le cycle de vie d'une instance de composant est idéalement descriptible et configurable en fonction de son contexte d'utilisation. Par manipulation des activités, nous entendons le fait de pouvoir rendre explicite et de supporter les manipulations des activités qui prennent place dans le système (comme par exemple les processus, les threads, les transactions ...) qui peuvent de plus être rattachées à plusieurs instances de composants en même temps. La flexibilité de déploiement est une notion importante dans la construction d'application, cependant, nous ne nous attarderons pas spécialement sur ce critère, étant donné qu'elle demeure assurée essentiellement par des outils d'administration.
Ces différents critères ne concernent donc pas directement la notion de composant mais font plutôt référence à des mécanismes de gestion de plus haut niveau relatifs à la plate-forme d'accueil. De plus, une autre vision des choses pourrait consister à considérer ces trois critères comme faisant partie des propriétés non fonctionnelles susceptibles d'être gérées par n'importe quelle plate-forme de composant.

Une notion également importante dans le concept de composant logiciel est de considérer le composant comme une unité de réutilisation. L'idée de la réutilisation n'est pas nouvelle puisque le paradigme de programmation orientée objet était déjà sensé apporter des solutions à ce problème par l'intermédiaire de techniques telles que le polymorphisme ou l'héritage. Cependant, on peut considérer de façon critique le fait que la programmation par composants puisse être la panacée en terme de réutilisation de code. En effet, pour qu'un composant soit réutilisable, il doit être portable, et son fonctionnement doit être bien documenté. La notion de portabilité est en effet importante puisque de nombreux critères non-fonctionnels (pour la plupart) dépendent d'une plate-forme d'accueil spécifique. Sans même parler des obstacles liés aux langages de programmation ou du contexte d'éxécution d'un code binaire (compilé ou interprété), la portabilité des composants nécessite une standardisation et une normalisation mûre et bien établie (peut-être dans le sens de CCM*). De plus, on ne peut considérer de réutilisation (même dans le cadre d'une plate-forme spécifique) de code sans documentation. Ces facteurs s'éloignent donc de la notion de composant, c'est pourquoi nous en ferons sans doute allusion dans la suite du rapport mais ils ne feront pas partie de nos critères d'évaluation. Cependant, la possibilité d'évaluer la réutilisation de code dépendant de la

plate-forme sera évoquée dans le rapport de stage, après l'implémentation de notre prototype d'application répartie.

3 La séparation des préoccupations

3.1 Définitions et difficultés

En théorie, la notion de séparation des préoccupations (SoC*) est très attractive et d'une simplicité déconcertante :

> Il s'agit du principe qui consiste à séparer les définitions d'entités distinctes d'une même application. Celle-ci se modélise alors et se conçoit comme un ensemble de modules applicatifs dont les fonctionnalités sont clairement délimitées.

Le principe de la séparation des préoccupations (« Separation Of Concerns ») propose donc de séparer les définitions des différentes préoccupations (ou fonctionnalités) dans le but de diminuer les temps de développement et d'améliorer la modularité, la réutilisabilité et la maintenabilité des différentes composantes de l'application.

En pratique, l'application de cette notion de séparation est extrêmement complexe. En effet, de nombreux facteurs "brouillent" notre vision de ces séparations.

1. Tout d'abord, la notion de préoccupation est complètement subjective. Aucun axiome ne nous permet d'affirmer qu'une fonctionnalité demeure exclusivement dans le domaine du fonctionnel ou du non-fonctionnel. Au sein d'une entreprise par exemple, chacun peut avoir une vision différente des fonctionnalités d'une application, et la séparation de celles-ci sera complètement subjective d'un corps de métier à l'autre.

2. Comment choisir la granularité d'une préoccupation ou d'une fonctionnalité ? Par exemple, si l'on décide d'implémenter un certain algorithme effectuant différents types de calcul, le considère-t-on comme une entité factorisant un type de fonctionnalité ou cherche-t-on à segmenter les différents types de calcul sous forme de fonctionnalités distinctes ?

3. Il est souvent possible de concevoir une application sous forme de couches applicatives, comme par exemple un algorithme utilisant les services proposés par une couche middleware. Mais de cette abstraction, il est parfois possible de dégager une séparation transverse à ces deux couches. Comment modulariser cette préoccupation dans l'implémentation ?

4. Dans le cadre d'une application répartie, une fonctionnalité peut être transverse à différents sites. Dans ce cas, la préoccupation est modélisée en tant que service unique en faisant abstraction de la distribution et du déploiement des entités fonctionnelles qui assurent ce service. Modélisons-nous donc notre application en faisant abstraction de cette répartition, ou au contraire tentons-nous d'assigner les fonctionnalités par rapport aux entités physiques (c'est-à-dire les sites) ?

5. Dans le cycle de vie d'une application, nous pouvons également considérer que le temps demeure un facteur non négligeable. En effet, dans le cadre d'applications d'entreprises, les spécifications évoluent quasiment en permanence, et il n'est pas rare de devoir ajouter en cours d'analyse voire de développement, des fonctionnalités non prévues au départ par le client. Des points initialement considérés comme non fonctionnels car se situant à la marge du projet, peuvent ainsi devenir au cours du temps des fonctionnalités à part entière.

Les choix relatifs à la définition des séparations au sein d'une application vont avoir un impact direct sur toutes les étapes du cycle de vie du logiciel (conception, modélisation, implémentation, choix des plates-formes de développement, mises à jour, structuration de l'équipe de développement en fonction des spécialités de chacun ...etc).

3.2 Ambiguïté

Avant de continuer, nous devons clarifier un point important en ce qui concerne la notion de séparation des préoccupations.
En effet, dans le modèle de composant générique que nous venons d'énoncer, nous parlons de caractéristiques non-fonctionnelles d'une plate-forme d'accueil. En ce sens, nous voulons exprimer le fait que les mécanismes mis en œuvre au sein de la plate-forme pour gérer les composants (par exemple leurs instanciations, leurs cycles de vie ... etc) doivent être au maximum séparés du reste de l'application, c'est-à-dire doivent être (en quelque sorte) transparents pour le développeur.

Cela n'a donc pas de rapport avec la séparation des préoccupations que cherche le développeur au sein de son application. Dans ce cas, la notion de séparation est une notion de plus haut niveau, indépendante d'une plate-forme d'accueil, d'un langage de programmation. Elle intervient d'ailleurs au moment de la conception dans le cycle de vie d'une application, au niveau de son modèle.

Dans ce rapport, nous parlons de ces deux notions, il est donc important de les différencier de manière à ne pas faire surgir d'ambiguïtés.

4 D'autres notions

Dans cette partie, nous nous attarderons volontairement sur certaines notions qu'il est important de bien comprendre et de clarifier pour la suite du rapport. Elles sont parfois en marge du sujet mais nécessaires.

4.1 Définition d'un framework (canevas)

Il existe une multitude de manières de définir un framework. De manière informelle, on pourrait définir un framework comme le squelette, la charpente d'une application. On trouve aussi les termes "bibliothèques de classes spécialisées" ou encore "ensemble de classes en collaboration".
Cela peut donc être vu comme une simple biblothèque de fonctions ou comme une architecture (ou plate-forme) complète de développement d'applications.

Pour Jean-Pierre Briot, quatre points viennent étayer ces premières définitions [Bri02] :

1. Un framework est une généralisation d'un ensemble d'applications
2. Un framework est le résultat d'itérations
3. Un framework est une unité de réutilisation
4. Un framework représente la logique de collaboration d'un ensemble de composants : variables et internes/fixés

Le premier point met en avant le fait que beaucoup de code dans une application peut être considéré comme étant générique, et donc redondant à beaucoup d'applications de logiques métiers hétérogènes. Il fournit un ensemble intégré (tout en restant ouvert et extensible) de fonctionnalités spécifiques à un domaine. Il permet donc dans ce sens une factorisation de ce code générique

(par exemple, une liste chaînée peut-être vue comme une entité générique à beaucoup d'applications - mais on pourrait citer des composants de granularité bien plus élevée).
Le deuxième point exprime le fait que le code fourni par le framework a été de nombreuses fois testé et réutilisé. Il s'agit donc de code complètement "mûr" et opérationnel et dont les implémentations ont été optimisées par des spécialistes (*« If it has not been tested, it does not work »*).
Le troisième point est lié au premier puisque du code redondant à plusieurs types d'applications sera susceptible d'être réutilisé au sein de contextes hétérogènes.
Pour finir, le framework fournit un modèle d'interaction entre les différents objets instances des classes définies (ou seulement spécifiées pour les classes abstraites) dans le framework.

Beaucoup d'autres critères pourraient enrichir ces définitions d'un framework comme par exemple fournir un environnement cohérent de programmation, sur une plate-forme d'exécution locale ou distante (et ce de manière transparente), fournir des outils de déploiement automatique d'applications, fournir un ensemble de services non-fonctionnels ... etc.

Pour reprendre les mots de Jean-Marc Jézéquel : «Un framework présente en général une inversion du contrôle à l'exécution ; alors qu'une application utilisant une bibliothèque s'appuie sur celle-ci, dans le cas d'une application utilisant un framework, c'est le framework qui effectue l'essentiel du travail et appelle "de temps en temps" un composant spécifique réalisé par l'implanteur de l'application. Un framework peut donc être vu comme une application semi-complète. Des applications complètes sont développées en héritant et en instanciant des composants paramétrés de framework. Il suffit donc en quelque sorte d'enficher dans un framework les composants spécifiques de son application pour obtenir une application complète.»

Parmi les exemples de frameworks, on pourrait citer les conteneurs de composants EJB, les frameworks de développement d'applications web (Apache Struts, Java Server Faces), .NET de Microsoft ... etc.

Les plates-formes FRACAL, KILIM et JAC peuvent toutes trois être considérées comme des frameworks.

4.2 Le Design Pattern Factory

L'idée des Design Patterns (ou patrons de conception) est d'identifier des solutions récurrentes à des problèmes de conception [Bri02] (*« Patterns in solutions come from patterns in problems »* [Ralph Johnson]). En quelque sorte, on

cherche à résoudre un problème en une solution générique et réutilisable dans différents contextes (par exemple en analogie avec les principes d'architecture - contruction de bâtiments, de cathédrales...).

Le patron de conception dont nous allons parler dans la suite du rapport est une fabrique de création (ou factory) qui est en fait une classe qui n'a pour rôle que de construire des objets. Ce modèle de création d'objets permet donc de manipuler des instances d'objets sans avoir à se soucier de chercher la classe de gestion de ces objets et la ligne de code qui effectue l'instanciation. Cette classe de création d'objets utilise des interfaces ou des classes abstraites pour masquer l'origine des objets. Une fabrique peut également effectuer des traitements de plus haut niveau comme c'est le cas dans Jonathan [TUT02] : Une fabrique (pour une classe `c`) est une classe qui est utilisée pour créer des instances de `c` mais aussi pour exécuter différentes tâches administratives, comme par exemple garder les références des instances créées, supprimer les instances (et donc gérer leurs cycles de vie en sein de l'application) ... etc.

On distingue parfois deux types de fabriques :

- Les fabriques abstraites reposant sur l'exploitation de classes génériques (interfaces ou classes abstraites).
- Les fabriques concrètes masquant toutes les méthodes nécéssaires à la création et à l'initialisation de l'objet.

La notion de fabrique est très utilisée dans les plates-formes que nous allons présenter. C'est une alternative appropriée à la configuration des objets lors de leur instanciation. En effet, plutôt que de paramétrer en dur une configuration des objets, il est bien plus pratique d'utiliser des fichiers externes à l'implémentation de l'application qui seront alors analysés par le processus de fabrication, qui pourra alors paramétrer de façon dynamique les nouvelles instances d'objets.

4.3 Méta-objet, intercepteurs

Dans ce paragraphe, nous nous contenterons de définir quelques notions de base utiles pour la suite (voir [Teb02] pour une présentation plus exhaustive des systèmes réflexifs).

Pour N. Bouraqadi et T. Ledoux, un méta-objet est un objet qui joue le rôle d'interprète pour un ou plusieurs autres objets. Ainsi, un méta-objet, attaché à un objet par un lien méta, permet de réifier les éléments de base d'un système, dans le but d'ajouter une couche de contrôle non-prévue dans le

système initial par l'intermédiaire d'un MOP* (*Meta Object Protocol*).

Il existe de nombreuses solutions et implémentations pour rendre un système réflexif [Teb02]. Parmi elles, nous citerons l'utilisation d'intercepteurs* (utilisés par les plates-formes que nous allons présenter) qui permet de capturer un événement précis afin d'organiser le passage du niveau de base au niveau méta.

L'introduction d'intercepteurs dans les systèmes réflexifs peut se faire (entre autres) par encapsulation d'objets ou par encapsulation de méthodes.
Dans le premier cas, elle consiste à intercaler un objet particulier appelé *wrapper* entre l'objet de base à contrôler et son environnement (figure 1). Dans ce cas, toutes les interactions entre l'objet de base et son environnement son interceptées par le wrapper puis redirigées vers le méta-objet.

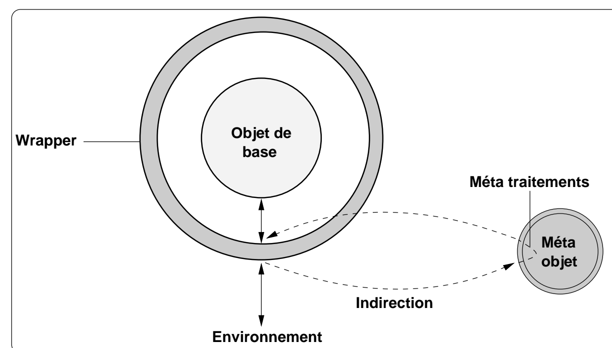

FIG. 1 – Encapsulation d'un objet

Dans le deuxième cas, on cherche à encapsuler les méthodes. L'objectif est alors de masquer l'appel à la méthode originale en la redirigeant vers une méthode de l'encapsuleur qui se chargera alors d'effectuer des appels au méta-objet correspondant et qui pourra à son tour appeler la méthode originale comme spécifié sur la figure 2. Une implémentation possible de cette indirection vers un niveau méta consiste à renommer la méthode à contrôler et à attribuer l'ancien nom à une nouvelle méthode de l'encapsulateur. Ainsi, l'indirection se fait de manière transparente, sans modifier le code source de l'application.
Dans le cadre de cet exemple, l'encapsulateur joue le rôle de wrapper de méthode en effectuant un traitement méta avant l'appel de la méthode initiale. Cependant, ce traitement peut s'effectuer également après l'appel à la méthode initiale. On pourrait imaginer des systèmes réflexifs permettant également de wrapper d'autre entités du langage, comme par exemple des boucles, des struc-

tures conditionnelles, des levées d'exceptions ... etc.

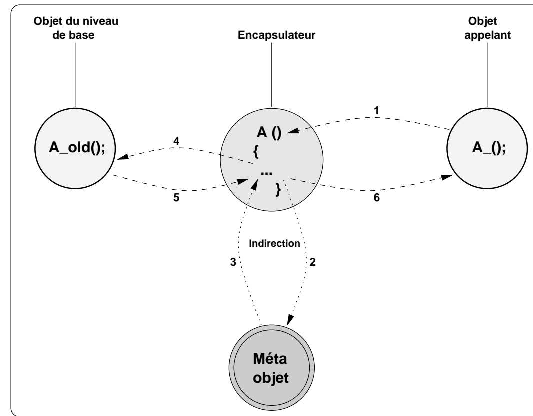

FIG. 2 – Encapsulation d'une méthode

Dans cette première partie, nous nous sommes intéressés à la notion de composant logiciel et nous avons cherché à définir les points essentiels inhérents à tous modèles de composants. Nous avons également défini la notion de séparation des préoccupations au sein d'une application et mis en avant les difficultés que ce principe est susceptible de faire apparaître, ainsi que certaines notions en marge de notre travail mais dont la compréhension est nécessaire pour la suite.

La prochaine partie de ce rapport est dédiée à la présentation des plates-formes que nous avons étudiées.

Deuxième partie

Présentation des plates-formes

Dans cette deuxième partie, nous présentons les trois plates-formes sur lesquelles nous avons travaillé dans le cadre de ce stage. Pour les plates-formes orientées composants (FRACTAL et KILIM) nous présentons leur modèle respectif de composant et d'assemblage. Pour la plate-forme orientée AOP (JAC), nous expliquons le paradigme de programmation par aspects ainsi que le modèle de composition de ceux-ci. Pour chacune d'elles, et de manière à bénéficier d'une approche plus pragmatique, nous donnons un exemple minimal d'implémentation.

5 Présentation de FRACTAL

FRACTAL est un projet du consortium OBJECTWEB.
Il peut se définir comme un framework de composition logicielle se basant sur la programmation orientée composant, incluant la définition des composants (leur type), la configuration, la composition et l'administration.
Cependant, FRACTAL ne doit pas être vu comme un produit en soi, mais doit plutôt être compris en tant qu'un framework (abstrait) d'architecture (défini par ses spécifications) et alors utilisé comme une base d'une famille de framework (concrets). Différentes implémentations peuvent alors reposer sur ces spécifications de FRACTAL, supportant différents langages de programmation. Cependant, à ce jour, la seule implémentation disponible est JULIA.

L'objectif premier de FRACTAL est de supporter la configuration dynamique au sein d'un système. Le modèle de composant FRACTAL est donc basé sur ces deux simples critères [BCS02] :

- Les composants sont considérés comme des structures d'exécution, elles se manifestent durant l'exécution du système
- Les composants sont construits en tant qu'unités de configuration dynamique pour le système

La présentation de la plate-forme FRACTAL repose sur les documents [BCS02, FT002, FAT03, JUT02].

5.1 Le framework FRACTAL

Le framework de FRACTAL peut-être décomposé en deux entités :

- Le noyau du framework : Il définit les concepts minimaux et les API nécessaires pour programmer avec les composants FRACTAL, et pour fournir les implémentations d'un framework de composition d'entités logicielles.
- Les incrémentations (ajouts) du framework : elles définissent les concepts additionnels et les API pour faciliter la programmation orientée composant, la configuration, la composition, l'administration, etc, qui sont considérées comme suffisamment générales pour être standardisées dans les spécifications.

L'objectif des spécifications FRACTAL est de proposer un framework très général sur la programmation orientée composant. En ce sens, (a) FRACTAL

peut-être utilisé comme base pour différents langages et outils, (b) peut supporter différentes sémantiques de composition telles la structurelle, la comportementale, l'opérationnelle, etc, (c) en tant que framework d'administration de composants, FRACTAL peut être étendu à un modèle de gestion de composants et peut être utilisé comme base à des outils d'administration, de supervision, de diagnostic, etc.

FRACTAL peut être vu comme un projet ambitieux visant à standardiser de la façon la plus générique possible le paradigme de programmation orienté composant, et ce également en environnement réparti.

5.2 Le modèle de composant

Les spécifications de FRACTAL sont basées sur deux modèles de composants : Un "modèle général" pour une description de haut niveau des composants et un "modèle concret", plus restrictif, mais plus facilement implémentable. Pour ce deuxième modèle, des choix ont été faits pour rendre possible l'implémentation en utilisant le langage Java.

5.2.1 Le modèle général

Le modèle général se base essentiellement sur quatre concepts généraux : Les noms, les interfaces, les signaux et les cellules (*kells*).

Les cellules sont une analogie aux cellules biologiques : Elles sont composées d'un plasme entouré par une membrane. Une cellule modélise un composant de haut niveau, une structure d'exécution. Le plasme contient d'autres cellules qui sont sous le contrôle de la membrane qui encapsule ces cellules. Le modèle est récursif pour les "sous-cellules". La membrane (ou le contrôleur) d'une cellule incarne le contrôle de comportement propre à la cellule. Une cellule peut appartenir à deux plasmes différents, alors soumise au contrôle de deux membranes distinctes.

Une cellule interagit avec son environnement par l'intermédiaire d'échanges de signaux sur des points d'accès bien identifiés, appelés interfaces. Les arguments de ces signaux peuvent être de trois formes : un nom (désignant une interface dans le système), une valeur ou une cellule (incluant alors l'état complet de la membrane et de son plasme permettant ainsi le déplacement des cellules). Les membranes disposent d'interfaces externes (points d'accès pour l'environnement de la cellule) et d'interfaces internes (invisibles de l'extérieur

mais visibles pour les cellules dans le plasme).

Le comportement d'une cellule peut être défini par un ensemble de transitions possibles. Chaque transition spécifie la cellule d'origine, un ensemble fini de signaux entrants et sortants, et un ensemble fini de cellules résultat, appelées résidu.

Un nom réfère l'interface d'une membrane. Chaque nom est relatif à un contexte, il n'y a pas d'ambiguïté possible entre deux noms d'un même contexte. La membrane d'une cellule constitue un contexte de nom primitif.

Les arguments, les signaux, les interfaces et les cellules sont typés.

5.2.2 Le modèle concret

Comme nous l'avons dit, le modèle concret est une spécialisation du modèle général. Il introduit de manière concrète la notion de composant et d'interface. La notion de nom dont nous avons déjà parlé est transposable à la notion de référence d'interface dans le modèle concret. Celui-ci va également définir la notion de type et de *template* qui permettent alors de décrire et d'instancier les composants.

Un composant est constitué de deux parties : le contrôleur (correspondant à la notion de membrane) et le contenu (correspondant à la notion de plasme).

Le contenu d'un composant est composé d'un ensemble fini d'autres composants (appelés sous-composants formant alors un ensemble non structuré) qui sont sous le contrôle du contrôleur du composant qui les encapsule. Ce modèle de composants est complètement récursif, la récursivité se terminant par les composants primitifs, qui encapsulent alors les objets Java usuels.

Comme nous pouvons le voir sur la figure 3, différents composants peuvent avoir des contenus qui se recouvrent, c'est-à-dire qu'un sous-composant peut être partagé par différents composants.

Chaque objet Java peut-être vu comme un composant (avec peu ou pas de capacité de contrôle) : un objet sans référence vers d'autres objets (figure 4(a), objets B et C) est considéré comme un composant primitif, c'est-à-dire un composant sans contenu. Cependant, l'objet A de la figure 4 peut-être vu de différentes façons :

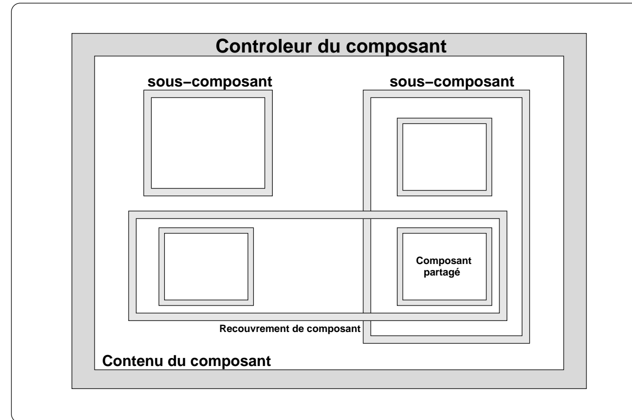

FIG. 3 – Modèle de composant - sous-composant - composants partagés dans FRACTAL

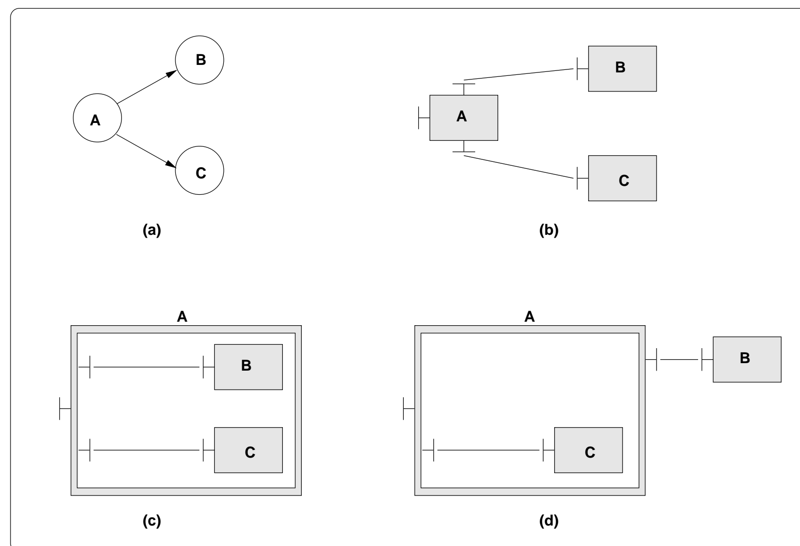

FIG. 4 – Composants et objets primitifs dans FRACTAL

- Comme un composant primitif avec des références vers d'autres objets à un même niveau d'emboîtement (figure 4(b)).
- Comme un composant composite composé de sous-composants mais sans référence vers d'autres composants (figure 4(c)).
- Comme un composant composite composé de sous-composants et avec des références vers d'autres composants (figure 4(d)).

Un composant peut interagir avec son environnement par l'intermédiaire d'invocations d'opérations sur une interface [1]. La visibilité d'une interface est déterminée par son contrôleur associé. C'est dans la configuration de ce contrôleur qu'est déterminé si une interface est visible pour l'environnement associé ou invisible. Les différents degrés de visibilité sont donnés dans la figure 5.

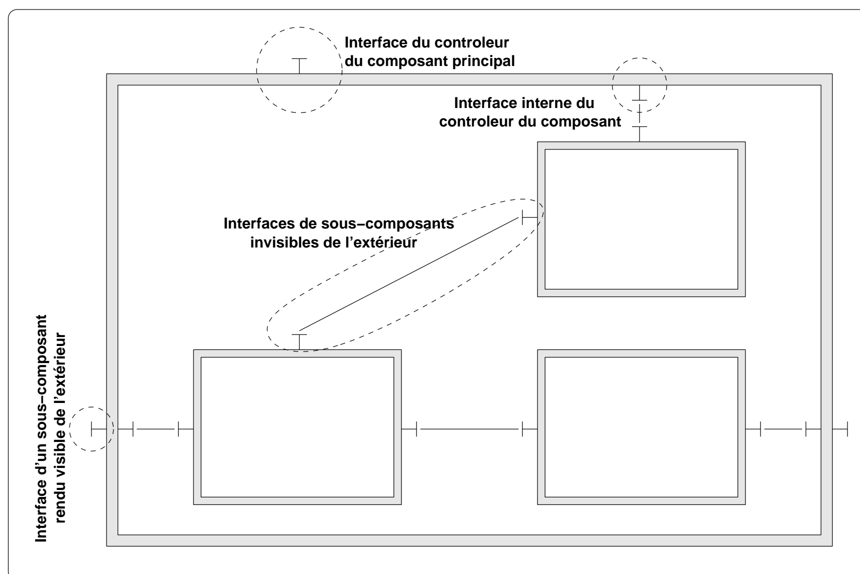

FIG. 5 – Visibilité des interfaces des composants dans FRACTAL

Dans le modèle concret, les opérations se définissent par des interactions basiques (elles sont implémentées par des appels de méthodes Java dans le framework JULIA). Elles sont de deux sortes : sens unique (*one way*) et aller-retour (*two ways*). Il s'agit d'une invocation d'opération sans retour dans le premier cas, avec retour de résultat dans le deuxième cas. Les arguments d'invocation et de retour peuvent-être des noms (c'est-à-dire des références d'interfaces), des valeurs, ou des formes passives de composants (non encore définis dans le modèle FRACTAL mais correspondent à des objets Java sérialisés et des objets

[1]Une interface doit être differenciée de son type. En effet, le type d'une interface (dans le sens de l'interface Java) peut-être utilisé par différents composants, alors qu'une interface proprement dite est associée à un et un seul composant.

à état persistant).

Les interfaces sont de deux types : les interfaces client (elles émettent des invocations d'opérations et récupèrent éventuellement les résultats dans une interaction de type aller-retour) et les interfaces serveur (elles sont susceptibles de recevoir des invocations d'opérations).

La définition d'un *binding* (littéralement liaison) entre deux interfaces est la suivante [BCS02] : Il s'agit d'un lien entre une interface cliente et une interface serveur, signifiant qu'un appel de méthode émis par l'interface d'un client doit être acceptée par l'interface serveur à qui est destiné cet appel. Une liaison (*binding*) est considérée comme correcte entre une interface client et serveur seulement si le serveur est dans la mesure d'accepter au moins toutes les invocations d'opérations qu'est susceptible d'émettre le client, et si l'interface client peut accepter tous les retours possibles d'invocation d'opération. En d'autres termes, le type de l'interface serveur doit au minimum être un sous-type de l'interface client. Cette définition d'une liaison entre deux interfaces est satisfaisante, nous l'utiliserons telle quelle dans la suite du rapport.

Le contrôleur d'un composant incarne le contrôle de comportement associé à un composant particulier. Un contrôleur de composant peut :

- Intercepter des invocations d'opérations entrantes et sortantes et des retours d'opérations provenant ou destinés aux composants qu'il contrôle.
- Fournir une représentation explicite des composants connectés qu'il contrôle.
- Superposer un contrôle de comportement au comportement des composants qu'il contrôle, comme par exemple suspendre ou activer les activités de ses composants, et les rendre passifs.

Ainsi, un contrôleur peut être vu comme une implémentation spécifique de la sémantique associée à la composition des composants qu'il contrôle. Ces capacités de contrôle peuvent-être associées à celles des conteneurs des framework de composants industriels (comme par exemple les EJB), mais elles peuvent être nulles également. Dans ce cas, un contrôleur permet uniquement d'encapsuler d'autres sous-composants, fournissant simplement la représentation de la composition d'une entité logicielle.

L'identité d'un composant est spécifiée par une interface spéciale (qui doit être obligatoirement fournie par toutes les instances de composants) permettant d'identifier de manière non ambigüe un composant et de fournir les accès à ses interfaces externes.

Une référence d'interface est constituée d'une référence sur l'interface identité du composant qui contrôle cette interface, un nom utilisé au sein de ce composant pour référencer l'interface et d'une propriété spécifiant si cette interface est externe ou interne.

Le modèle concret de FRACTAL introduit les notions de types et certains mécanismes pour permettre à l'utilisateur du framework d'instancier les composants à partir de leurs types et ainsi de vérifier la bonne sémantique de l'assemblage de ces composants.

Un type d'interface est composé d'un identifiant, d'une signature d'interface et de trois booléens, indiquant respectivement le rôle, la contingence et le comportement de liaison des interfaces de ce type :

- L'identifiant d'une interface est un nom valide dans le contexte du composant possédant cette interface.
- La signature d'une interface est une collection de signatures de méthodes, elle est incarnée par les interfaces Java usuelles.
- Deux rôles sont possibles pour une interface : client ou serveur.
- Le contingence d'une interface peut prendre deux valeurs : obligatoire ou optionnelle. Pour une interface serveur, une interface obligatoire est une interface qui doit être fournie par un composant, une interface optionnelle peut ne pas être fournie par le composant. Pour une interface cliente, une interface mandataire est une interface qui doit obligatoirement être liée à l'exécution, une interface optionnelle peut ne pas être liée à l'exécution.
- Le comportement de liaison peut prendre deux valeurs : singleton ou collection. Ce concept de comportement de liaison est une façon de permettre à un composant de fournir plusieurs interfaces du même type (interfaces fournies ou requises). Ainsi, une opération de liaison (*binding operation*) sur une interface `C` retournera :

  - Une interface nommée `C`, `C` est alors appelée une interface singleton.
  - Ou une interface nommée `C`*suffixe* où *suffixe* est constituée d'une chaîne de caractères arbitraire concaténée à `C`. Dans ce cas, on parle d'une interface collection.

Un type de composant est alors une collection de types d'interfaces, décrivant les interfaces que le composant peut ou doit posséder à l'exécution.

Dans le modèle FRACTAL, les interfaces et les types de composants sont immuables, ils ne peuvent être modifiés à l'exécution.

Les composants FRACTAL sont instanciés par des templates, eux-mêmes créés par des usines de fabrication de templates. L'instanciation peut être de deux formes :

- La création d'un nouveau composant, à partir de son type de composant ;
- L'introduction d'un composant déjà existant dans le système, à partir d'une référence d'interface sur l'interface identité du composant.

5.3 L'architecture du framework FRACTAL

Dans cette section, nous présentons les notions les plus importantes des interfaces qui composent l'API de FRACTAL. Pour une présentation exhaustive, se reporter à [BCS02].

5.3.1 Le noyau du framework

Le noyau du framework (regroupant les concepts principaux et les plus mûrs au sein du projet) est organisé en quatre packages :

- `org.objectweb.fractal.api`

  Il correspond au noyau du framework. Il spécifie les concepts de référence de composant (c'est-a-dire l'identité d'un composant FRACTAL), de référence d'interface et des notions basique de type.

- `org.objectweb.fractal.api.type`

  Il spécifie le système de typage des interfaces et composants.

- `org.objectweb.fractal.api.control`

  Il spécifie les interfaces de composants fournissant différentes capacités de contrôle de ces composants.
  Dans le modèle FRACTAL, ces interfaces incarnent le contrôle de comportement exercé par les contrôleurs de composants (c'est-à-dire les membranes du modèle général). Il s'agit donc de spécifications minimales qui permettent une configuration dynamique du système, constituant l'objectif principal du projet FRACTAL. Nous allons donc nous étendre sur le contenu de ce package composé de quatre interfaces :

  Un contrôleur de liaison (*binding controller* spécifié par l'interface `BindingController`) permet de gérer les liaisons entre interfaces de composants, il est utilisé

pour effectuer un *lookup* d'interface cliente, pour créer une liaison entre une interface cliente et serveur, pour détruire une liaison d'une interface cliente. Le comportement lié à la gestion de ces liaisons peut avoir différentes sémantiques selon les emplacements relatifs des interfaces : liaison entre une interface cliente interne avec une interface serveur externe d'un sous-composant et inversement, liaison entre interfaces gérées par différents espaces d'adressage (différentes JVM).
Un contrôleur de contenu (*content controller* spécifié par l'interface `ContentController`) est utilisé pour contrôler le contenu d'un composant, représenté par un ensemble fini de sous-composants. Il permet de récupérer de façon explicite cet ensemble de sous-composants, d'ajouter ou de retirer des sous-composants de cet ensemble, de récupérer les interfaces internes de ses sous-composants.
Un contrôleur d'attributs (interface `AttributeController`) est utilisé pour configurer les attributs primitifs des composants (les attributs primitifs de Java en l'occurrence). Cette interface est actuellement vide, c'est à l'utilisateur de définir explicitement ses propres interfaces héritant de l'interface `AttributeController` spécifiant au moyen de méthodes *setter* et *getter* quels attributs primitifs doivent être contrôlés de l'extérieur. L'utilisateur doit respecter les conventions de nommage de ces méthodes telles qu'elles sont spécifiées dans le modèle JavaBeans de Sun.
Un contrôleur de cycle de vie (interface `LifeCycleController`) est utilisé pour contrôler le cycle de vie des composants au cours de l'exécution. Il incarne ce que le modèle général FRACTAL spécifiait comme une suite de transitions induite par un événement provoquant le changement de l'état des composants. Actuellement, seuls deux états sont "implémentés" : l'état *STARTED* pour lequel le composant est capable d'émettre ou d'accepter des appels de méthodes. L'état *STOPPED* pour lequel le composant ne peut émettre des appels de méthodes, mais peut cependant accepter des méthodes appelées par l'intermédiaire de ses interfaces de contrôle ou des interfaces non-fonctionnelles.

- `org.objectweb.fractal.api.factory`

  Il correspond aux spécifications liées à la gestion du démarrage de l'application (*bootstrap*), permettant d'instancier les composants initiaux du système.
  Il définit deux interfaces :

  L'interface `Template` fournit une méthode permettant d'instancier les composants et qui retourne une référence sur l'interface de l'identité du composant instancié. Deux méthodes permettent également d'accéder aux descriptions du contrôleur du composant et de son contenu.

L'interface `TemplateFactory` fournit deux méthodes permettant de créer les composants *templates* : l'une par *introduction*, c'est-à-dire à partir de la référence de l'identité d'un composant déjà créé, l'autre par *création*, à partir d'une référence d'un type de composant. Dans ce deuxième cas, trois paramètres supplémentaires sont à fournir : deux paramètres pour une description du contrôleur du *template* instancié (c'est-à-dire quelles interfaces de contrôle le supportent), et le troisième paramètre pour une description du contenu du composant instancié alors créé par l'usine de fabrication de *templates* (placé à `null` dans le cas d'un contenu intitial vide, ou une classe Java dans le cas d'un composant primitif).

5.3.2 Les incrémentations du framework

Elles correspondent à des APIs ou à des outils qui ne sont pas obligatoires dans l'implémentation du framework, ou à des notions qui ne sont pas encore suffisamment mûres pour être intégrées dans le noyau de FRACTAL.

- Support de programmation
  Le package `fractal.factory` permet une création plus flexible des *templates* et des composants, permettant d'instancier des composants de manière générique. Pour le moment, les mécanismes proposés sont spécifiques à l'implémentation de l'API. Comme il est spécifié dans le modèle FRACTAL, il faudrait que ces mécanismes soient raffinés de manière à modéliser les contrôleurs de composants et leurs contenus avant le processus d'instanciation.
  Des formes communes de composants ont donc été introduites, basées sur différentes combinaisons d'interfaces de contrôle. En quelque sorte, elles correspondent à des "alias" vers différents ensembles d'interfaces (non-fonctionnelles) permettant de spécifier des catégories de composants selon leurs besoins non-fonctionnels.

  Les "formes communes" de composants données par les incrémentations du framework sont les suivantes :

  - Les composants primitifs : typiquement, ils sont composés de contenu vide, dans le sens où ils ne peuvent être contrôlés par l'interface `ContentController`. Ils constituent les unités primaires de réutilisation en sein du système.

  - Les composants composites : c'est la forme la plus commune des composants FRACTAL, étant donné qu'ils permettent de matérialiser la notion de récursivité des contenus des composants telle spécifiée dans le modèle général. Ils constituent les unités primaires d'assem-

blage structurel d'un système complexe.

  - Les composants de domaine : il s'agit d'une forme spéciale de composants composites pour lesquels les liaisons ne peuvent être changées. Ils concernent des composants pour lesquels les interactions ne constituent pas un souci essentiel (comme les caches par exemple).

  - Les composants paramétrés : typiquement, ils supportent l'interface `AttributeController` permettant de les configurer.

  - Les composants statiques : Ils ne supportent pas l'interface `LifeCycle`, leurs cycles de vie ne peuvent donc pas être contrôlés.

  - Les composants *templates* : Il s'agit de composants spéciaux qui permettent d'instancier d'autres composants.

  - D'autres formes de composants sont à développer dans de futures implémentations de FRACTAL : mobiles, répliqués, asynchrones ...

- Support pour la configuration
  Il s'agit de spécifications relatives à certains outils d'aides à la configuration de l'application, par exemple par l'utilisation d'interfaces graphiques commes des langages de descriptions d'architectures (ADL), ou des outils pour gérer les types de composants, les *templates*.
- Support pour la composition
  Pour la gestion de différentes sémantiques d'assemblage. En cours d'étude.
- Support pour l'administration
  API's permettant la gestion de la journalisation, la supervision ...

5.4 Exemple d'implémentation

5.4.1 Formulation d'un exemple

Cet exemple est tiré du tutoriel de FRACTAL [FT002]. Il s'agit d'une application très simple composée de deux composants primitifs encapsulés dans un composant composite (figure 6).

Le composant primitif "serveur" fournit une interface qui permet d'afficher des messages sur la sortie standard. Il peut-être paramétré par deux attributs : Un attribut "entête" qui permet de configurer une entête qui sera affichée devant chaque message et un attribut "nombre" qui permet de configurer le nombre de fois que devra être affiché un message. Le composant primitif

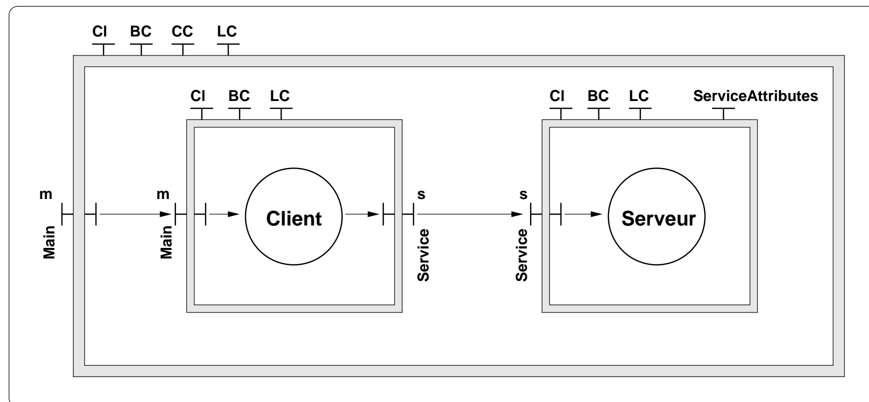

FIG. 6 – Deux composants primitifs dans un composant composite dans FRACTAL

"client" utilise le composant précédent pour afficher ses messages.

Le composant serveur fournit une interface nommée "s" de type `Service`, fournissant une méthode `afficher`. Il possède également une interface AttributeController de type `ServiceAttributes`, fournissant quatre méthodes pour récupérer (*get*) et paramétrer (*set*) les deux attributs du composant.

Le composant client fournit une interface nommée "m" de type `Main`, fournissant la méthode `main` alors appelée lorsque l'application est lancée, et une interface "s" de type `Service`.

5.4.2 Les étapes de la construction

Les étapes de construction de l'application sont alors les suivantes :

1. **Création des interfaces**
On déclare les deux interfaces fonctionnelles : l'interface `Service` déclare la signature de la méthode `afficher`, l'interface `Main` la méthode `main`. L'interface de contrôleur d'attribut `ServiceAttributes` déclare les signatures des méthodes `getEntete`, `setEntete`, `getNombre` et `setNombre`, elle hérite de l'interface `AttributeController` définie dans l'API de FRACTAL.

2. **Implémentation des classes des composants qui implémentent ces interfaces**
   On implémente la classe `ServeurImpl`, implémentant les interfaces `Service` et `ServiceAttributes`.
   La classe `ClientImpl` implémente l'interface `Main` et `UserBindingController`.

3. **Création des types de composants**
   Dans le cadre de notre application, trois types de composants doivent être créés que l'on nommera `rType` pour le type de composant racine (le type de composant composite), `cType` et `sType` pour les types de composants client et serveur. Comme nous l'avons mentionné dans le modèle concret, un type de composant se définit par l'ensemble des types d'interfaces qu'il fournit. C'est donc à cette étape que sont formulés les types de chaque interface, par l'intermédiaire de méthodes fournies par l'API de FRACTAL. Par exemple, le type de composant client sera composé de deux types d'interfaces (dont les paramètres à spécifier sont donnés dans le modèle concret des composants FRACTAL) : `("m", "Main", SERVER, MANDATORY, SINGLE)` et `("s", "Service", CLIENT, MANDATORY, SINGLE)`. Le comportement de liaison (*binding*) est de type unique (`SINGLE`) et elles sont toutes deux obligatoires (`MANDATORY`). La première est de type `SERVER` puisqu'elle fournit un service à l'interface interne du composant composite, la deuxième est de type `CLIENT` puisqu'elle demande un service au composant serveur. Les deux premiers paramètres sont respectivement le nom de l'interface et sa signature (l'interface Java). Une usine de fabrication de type de composant sera utilisée telle qu'elle est définie dans l'API.

4. **Création des composants génériques**
   A ce stade, il faut créer les composants *template*, qui pourront être instanciés et donc qui pourront s'exécuter dans le système.
   On associe un type de composant à chaque composant, le comportement des instances ainsi qu'une desciption de leurs contenus initiaux. Ce dernier paramètre sera placé à `null` dans le cas des composants composites, ou l'on spécifiera la classe Java à instancier dans le cas de composants primitifs. Ces *templates* sont créés à partir d'une usine de fabrication définie par l'API.
   C'est également à ce stade que l'on configure les attributs du composant serveur.

5. **Assemblage structurel des composants génériques**
   Dans le cadre de notre exemple, deux méthodes d'assemblage sont possibles : dans un premier cas, il est possible d'instancier les *templates* un par un, de placer les composants primitifs résultant dans le composant composite et de les connecter. Dans le deuxième cas, il est possible

de placer les *templates* primitifs dans le *template* composite, connecter chaque *template* entre eux puis d'instancier l'application en instanciant seulement le composant *template* composite. Dans les deux cas, les assemblages sont effectués par des méthodes explicites spécifiées par l'API FRACTAL. Les interfaces sont ensuite liées (*bindées*) entre elles.

6. **Instanciation du composant racine**
   Si on choisit la deuxième alternative d'assemblage des composants, il suffit donc d'instancier le composant composite racine pour que tous les composants qu'il contient soient automatiquement instanciés à leur tour.

7. **Démarrage du composant**
   On appelle explicitement le composant racine obtenu à se rendre actif.

8. **Appel à la méthode principale**
   Il suffit d'appeler ensuite la méthode `main` disponible par l'interface externe du composant composite pour que l'application puisse s'exécuter.

5.4.3 La reconfiguration dynamique

Il s'agit là d'un des objectifs principaux du framework FRACTAL. Elaborons donc les étapes nécessaires au niveau applicatif pour permettre une reconfiguration d'une application en cours d'exécution (toujours dans le contexte de l'exemple donné plus haut, supposons que l'on veuille changer l'implémentation du composant serveur) :

1. **Récupération de l'identité du composant**
   On récupère l'identité du composant serveur en invoquant une méthode *lookup* sur ce composant.

2. **Arrêt du composant racine**
   On stoppe le service de persistance du composant composite racine, cedernier se trouvera donc dans l'état *STOPPED*.

3. **Suppressions des liaisons entre interfaces**
   On effectue un *unbind* de l'interface "s" entre le client et le serveur.

4. **Suppression du composant**
   On supprime alors le composant serveur qui jusqu'à présent se trouvait dans le contenu du composant racine.

5. **Création du nouveau composant**
   Il faut ensuite créer un nouveau composant *template* qui va permettre d'instancier la nouvelle implémentation du composant serveur. Pour cela, on procède comme nous l'avons expliqué lors de l'énumération des étapes de création de l'application, en utilisant le type de composant précédemment défini (dans le cas où celui-ci ne change pas bien sûr) et la nouvelle classe d'implémentation. Ce composant est alors instancié.

6. **Ajout du nouveau composant**
   On ajoute la nouvelle instance au composant racine de notre application.

7. **Création de la nouvelle liaison**
   Un nouveau *bind* est effectué entre l'ancien composant client et le nouveau composant serveur.

8. **Relancement du service de persistance**
   On "démarre" le composant racine.

5.4.4 Utilisation de l'ADL FRACTAL

Dans la description des étapes de la construction de l'application dans le cadre de notre exemple, nous avons utilisé les méthodes définies dans l'API de FRACTAL pour assembler les composants ; l'assemblage est donné par les étapes 3, 4 et 5, respectivement la création des types de composants, la création des composants génériques et finalement l'assemblage structurel de ces composants. La description de cette composition est donc codée "en dur" dans les sources de l'application. Pour modifier le contenu de notre composant racine, il faut donc modifier le code source puis le recompiler pour prendre en compte ces nouveaux changements.
Cependant, ces trois étapes peuvent être décrites par l'ADL de FRACTAL, dans un fichier XML. Cette description de l'architecture de l'application devient donc indépendante de son implémentation. Pour l'utiliser, il suffit d'invoquer un parser dans le code de l'application (qui est en fait un composant composite FRACTAL qui implémente le parseur de l'ADL de FRACTAL), qui chargera les définitions des types de composants et des *templates* et qui invoquera alors automatiquement les méthodes pour assembler ces composants et effectuer les liaisons entre les interfaces (également décrites par l'ADL). Il est également possible de paramétrer les attributs primitifs d'un composant (les paramètres `entete` et `nombre` de l'interface `ServerAttributes` du composant serveur).
Un élément XML `external-template` permet de récupérer l'identité d'un composant qui a déjà été instancié dans le système (il peut s'agir d'une instance locale ou distante).

Un mécanisme d'héritage a été défini dans les spécifications de l'ADL permettant de rendre plus modulaire la description de l'architecture de l'application et ainsi de faciliter sa réutilisation.
L'intégralité de la sémantique d'assemblage structurelle de FRACTAL est prise en charge par les spécifications de l'ADL.

5.5 Architecture de JULIA

JULIA [JUT02] est une implémentation de France Telecom des spécifications FRACTAL.

Un composant FRACTAL est implémenté sous forme d'un ensemble d'objets intercepteurs de contrôle qui sont instanciés en même temps que l'application. Les classes de ces objets de contrôle sont spécifiables à partir d'un fichier de configuration (`julia.cfg`) qui sera parsé à l'exécution de l'application.
Au sein de JULIA, de nombreuses classes ont été implémentées de manière à ce que l'utilisateur puisse disposer d'une large palette de choix au niveau des mécanismes de composition des composants c'est-à-dire allant d'un assemblage très statique (très performant) mais peu reconfigurable dynamiquement à un assemblage permettant une grande capacité de reconfiguration dynamique (mais moins performant).
Au moment de l'exécution, JULIA se chargera donc de parser les fichiers de description de l'assemblage des composants (l'ADL) ainsi que son fichier de configuration, et d'effectuer toutes les liaisons nécessaires entre les objets de contrôle et les instances implémentées par le développeur.

6 Présentation de KILIM

KILIM est un projet d'un éditeur français, Kelua, fondé en octobre 2000, et faisant partie du consortium OBJECTWEB. Il s'agit d'un framework de configuration open-source pour Java. Il permet de configurer des assemblages de classes.

6.1 Les motivations

L'équipe de Kelua part du principe que l'intégration constitue un défi pour les systèmes orientés objet [Hor03], et ce en terme de réutilisabilité, de flexibilité, d'adaptabilité (au niveau du matériel, du système d'exploitation ...) et de maintenabilité. Les bonnes abstractions doivent être dégagées, et des outils d'intégration et de composition sont alors essentiels. On peut considérer que les séparations entre interfaces et implémentations constituent des abstractions de niveau trop bas. Dans le souci de développer de nouvelles technologies et de dégager des abstractions architecturales de plus haut niveau, KILIM propose donc un modèle de composants et d'opérateurs de composition, ainsi qu'un mapping entre ces abstractions et des abstractions au niveau du langage.

6.2 Les objectifs de KILIM

Actuellement, KILIM en est à sa deuxième version.
L'objectif de cette dernière version est de pouvoir obtenir et manipuler un framework* générique et dynamique dans le but de construire des systèmes opérationnels à partir de sous-systèmes (ses composants), d'intégrer des composants systèmes opérationnels dans des systèmes existants, de vérifier la consistance architecturale des systèmes ainsi composés.
Une séparation claire est faite entre la vue composant et la vue objet, avec des règles explicites de mapping entre elles.

Dans KILIM2, trois points importants caractérisent la vue composant :

- Le modèle de composant (en terme d'entité) est similaire à celle de Fractal.
- Deux niveaux de description : Le premier, statique et textuel (basé sur une syntaxe XML), correspondant à une description de l'assemblage des composants appelé *template*, qui permettra de construire une méta-représentation de l'assemblage de l'application, le deuxième concerne la description du système au niveau exécutif (qui peut être dynamiquement modifié), permettant de gérer l'application en fonction de sa méta-représentation.

6.3 Le modèle de composant et les concepts de KILIM

Dans cette section, nous expliquons les concepts fondamentaux de KILIM, d'après les documents [Hor03, Del03, HD03].

6.3.1 Les `ports`, `providers` et les `transformers`

Les `ports`, les `providers` et les `transformers` sont les trois concepts de base de Kilim. Un `port` (caractérisé par un nom et un type) est une variable utilisée pour récupérer la référence de l'interface d'un objet. Un `provider` est une abstraction d'un mécanisme du niveau du langage qui permet d'obtenir des références d'objets. Leur forme la plus commune peut être des constructeurs, mais aussi des méthodes de type *getter*, des méthodes (type *lookup* permettant de récupérer une référence distante) ...

Un `transformer` est également une abstraction d'un mécanisme du niveau du langage qui permet de transformer l'état d'un objet. La forme la plus commune de `transformer` sont les méthodes de type *setter* qui permettent de changer les valeurs des attributs d'un objet. L'exécution d'un `transformer` se fait par la génération d'un événement (*trigger*).
Les deux types d'événements disponibles actuellement dans le modèle de KILIM sont les suivants :

- Le *bind* qui est généré lorsqu'un `port` récupère une (nouvelle) valeur.
- L'*unbind* qui est généré lorsqu'un `port` reçoit une valeur nulle

A ce niveau d'abstraction, il est alors possible de dresser un graphe entre les différentes entités énoncées. Dans ce cas, les noeuds du graphe seront rattachés à des attributs d'objets, des `ports`, des `providers`, des `transformers`. Les arcs permettront de modéliser les relations entre les noeuds (dépendances entre entités, événements ou support). A un niveau plus pragmatique, ce graphe représentera la manière dont est gérée l'instanciation des objets, les dépendances entre références (on doit par exemple posséder la référence d'un objet avant d'invoquer une méthode de type *setter* sur celui-ci) ... etc.

6.3.2 `slots` et assemblage de composants

Un composant KILIM (aussi appelé instance de `template` comme nous le verrons plus tard) est une extension des abstractions usuelles du concept d'objet. Il est attaché à un ensemble de `ports`, `providers` et `transformers`. L'assemblage des composants se fera alors par des opérations de liaison entre ses `ports`.

Les `ports` (ou plus simplement les interfaces) constituent les points d'accès du composant. Au sein du modèle, elles peuvent être de deux types : Une interface peut être unaire (*unary*), dans ce cas, une seule et unique liaison (ou *binding*) peut-être effectuée sur cette interface, ou n-aire (*nary*), pour laquelle un nombre indéterminé de liaisons peut-être effectué.

Un concept important lié à l'assemblage est la notion de `slot`, qui correspond à un ensemble de `ports` permettant de matérialiser l'atomicité du processus de liaison (*binding*) entre deux composants. Par exemple l'assemblage entre deux composants - à ce niveau d'abstraction - par l'intermédiaire d'un `slot` composé de trois `ports` se fera en une opération (`plug`) qui implicitement, correspondra à trois liaisons.
Un `slot` est une sorte de "prise multi-fonctions" qui permet donc de grouper un ensemble d'interfaces. En quelque sorte, un `slot` permet de spécifier l'aspect d'un composant, dans le sens où l'on pourra séparer les slots techniques (non-fonctionnels) et les slots spécifiques à l'application (correspondant alors à la logique métier).

6.3.3 La notion de `template`

Un `template` peut être considéré comme une externalisation statique de la représentation d'un composant KILIM2, et fournit toutes les abstractions que nous avons déjà décrites. En quelque sorte, il s'agit d'un ADL* fournissant les méta-informations nécessaires à la description, la création, la composition et à la configuration des composants, et d'une unité opérationnelle, dans le sens où elle contient toutes les informations nécessaires à son instanciation au sein de la plate-forme KILIM. Les `templates` sont décrits en XML.

L'ADL de KILIM2 propose un mécanisme d'héritage, permettant la spécialisation de composants et la factorisation des descriptions.

Comme dans le modèle de composant de Fractal, le contenu d'un composant KILIM peut-être un ensemble de composants, permettant ainsi des assemblages récursifs. De plus, la description de l'assemblage d'un système est susceptible d'être modifiée dynamiquement au cours de l'exécution du système. Il en est de même pour la configurabilité des propriétés.

De manière plus pragmatique, à un composant du système sera associé un `template`, dans lequel on spécifiera les `ports` (avec leurs rôles - offert ou fourni, leurs arités - 1 ou non défini) associé à ce `template`, les `slots` et les propriétés configurables. Puis on spécifiera les règles de mapping (les procédés

d'instanciation et de liaison) entre le niveau `template` et le niveau langage (correspondant aux notions de `transformers` et de `providers`). C'est par exemple à cette étape que l'on spécifiera les arguments à fournir au constructeur de l'objet décrit par le `template`, ou encore les méthodes à appeler lorsque la liaison avec un `slot` du `template` est effectuée.

Une troisième étape consiste à définir la composition du composant final. A ce niveau d'abstraction, on "manipule" des noms d'instance de classe dont les assemblages sont explicitement décrits, par l'intermédiaire des `slots`.
L'instanciation du composant pourra ensuite se faire en utilisant les méthodes de l'API de KILIM, ainsi que la reconfiguration dynamique de celui-ci (comme la spécification d'un nouvel assemblage, une nouvelle paramétrisation d'attributs).

6.4 Exemple d'implémentation

Nous allons reprendre l'exemple extrait du tutoriel de FRACTAL (cf. 5.4.1 page 38) et le transposer à KILIM.

Notre application sera composée de deux composants KILIM, un composant `Client` et un composant `Serveur`. Le `Client` sera associé à la classe `ClientImpl` qui implémente la méthode `afficher` (prenant une chaîne de caractères en argument) dont le corps invoque en fait la méthode `printServeur` de la classe `ServeurImpl` associée au composant `Serveur`. La classe `ServeurImpl` implémente les méthodes *setter/getter* pour chacun des attributs `entete` et `nombre` (soient quatre méthodes).
Les deux composants seront donc assemblés de manière à construire un composant de plus haut niveau (nommé `Appli`), comme le schématise la figure 7.

Le composant `Serveur` est composé d'un port nommé `portAppli` et d'un slot `slotServeur` (composé lui-même d'un port `portServeur`). Le provider de ce composant permet de récupérer une référence de l'instance de la classe `ServeurImpl` (dans notre cas, nous utilisons un constructeur). Daux transformers sont associés également au `portAppli` : il s'agit du mécanisme permettant de configurer les deux attributs `entete` et `nombre`. Un troisième transformer lié au `portServeur` déclenchera un événement qui permettera au provider du composant `Client` d'instancier la classe `ClientImpl` (toujours en utilisant un constructeur). Le transformer du composant `Client` permet à l'instance de `ClientImpl` de récupérer la référence de `ServeurImpl` dont elle a besoin dans la méthode `afficher`.
Le port `portClient` du composant `Appli` permet d'externaliser la référence du composant `Client` et donc d'invoquer par la suite les méthodes sur l'instance

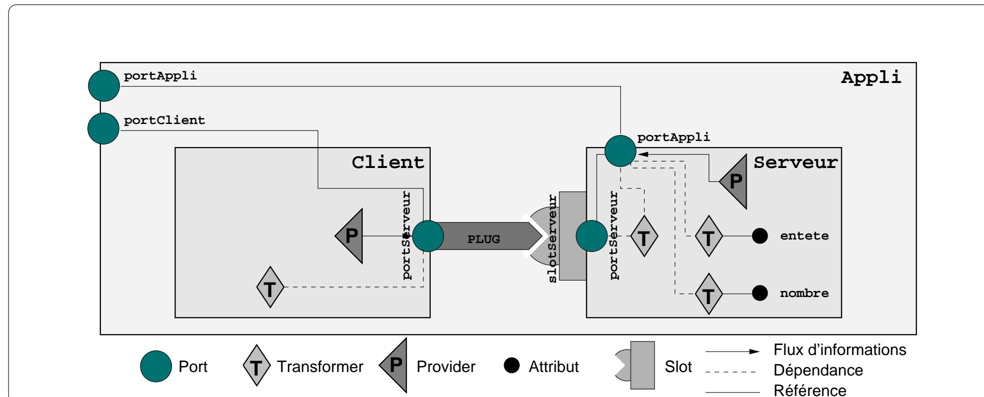

FIG. 7 – Modélisation de la composition de notre application avec KILIM

de `ClientImpl`.

6.4.1 Les étapes de la construction

Les étapes de la construction de l'application sont alors les suivantes :

1. **Implémentation des classes**
   On implémente les classes `ClientImpl` et `ServeurImpl`. La classe `ClientImpl` implémente donc la méthode **`afficher`** qui invoque la méthode **`printServeur`** sur le serveur. Elle implémente également la méthode **`setServeur`** prenant en argument une instance de `ServeurImpl`. La classe `ServeurImpl` implémente la méthode **`printServeur`** qui affiche l'**`entete`** et la chaîne de caratères passée en argument **`nombre`** fois sur la sortie standard ainsi que les méthodes *setter/getter* pour gérer les deux attributs.

2. **Description du composant `Client`**
   Le *template* associé au composant `Client` définit donc le `portServeur` et le provider associé au *binding* de ce port : la création d'une nouvelle instance de `ClientImpl`.

3. **Description du composant `Serveur`**
   Le *template* associé à ce composant définit deux property (qui correspondent à des propriétés au niveau du modèle KILIM), nommées **`entete`** et **`nombre`**, dont nous spécifions de manière explicite leur contenu (remarquons qu'à ce niveau de modélisation, ces deux property ne sont pas mappées avec les deux attributs correspondant du niveau langage, elles

le seront par l'intermédiaire des transformers). On définit le provider lié à ce composant, son slot qui permettera de *plugger* les deux composants ainsi que les trois transformers dont nous avons parlé plus haut.

4. **Description du composant `Appli`**
   Son *template* est composé de deux instances (instances abstraites du modèle KILIM associées aux *templates* du composant `Client` et du composant `Serveur`), des deux ports `portAppli` et `portServeur` référençant les ports des sous-composants comme schématisé sur la figure ci-dessus (cf. 7 page 48). C'est dans ce *template* que l'on définit l'assemblage des composants, c'est-à-dire l'assemblage des deux instances.

5. **Implémentation de la méthode `main`**
   Dans cette méthode, on utilise l'API de KILIM (`newComponent` pour créer un nouveau composant `Appli`, il s'agit alors d'une référence vers l'objet représentant l'instance du *template*, et donc d'une méta représentation de l'ensemble de l'application (à ce stade, aucune instanciation n'est effectuée au niveau du langage). Sur cette représentation du composant `Appli`, on invoque la méthode `getInterface` avec en argument `portAppli` qui nous permet de récupérer une référence de l'"interface" défini par `portAppli`. Enfin, on invoque un `getValue` sur cette "interface". Cet appel génére une liaison (*binding*) sur le port concerné, permettant une exécution en chaîne des providers et `transformers` afin d'instancier au niveau du langage notre application. Ce dernier appel nous renvoie une référence sur l'objet du langage, dans ce cas, une référence sur l'instance de `ServeurImpl`.
   On récupère ensuite la référence de l'instance de `ClientImpl` qui nous permet alors d'invoquer la méthode `afficher` sur cet objet.

6.4.2 La reconfiguration dynamique

1. **Modification de la valeur d'une propriété**
   A partir d'un composant, en utilisant l'API de KILIM, il est possible de récupérer la référence d'une property à partir de son nom. Il suffit alors d'invoquer une méthode `setValue` pour modifier le contenu de cette propriété.

2. **Modification de l'assemblage entre composants**
   Il suffit de récupérer la référence du slot concerné, puis de *plugger* le nouveau composant sur ce slot.

3. **Prise en compte au niveau du langage**
   Les modifications sur les propriétés et l'assemblage cités ci-dessus s'effec-

tuent au nivau méta, c'est-à-dire au nivau de la méta représentation de l'application, et non au niveau du langage (des objets JAVA). Cependant l'API implémente une méthode `update` qui permet alors de réexécuter tous les *triggers* associés à une méta interface, et ainsi de mettre à jour les propriétés et les instances de composants.

4. **Reconfiguration avec JMX**
La technologie JMX (*Java Management Extensions*) permet de surveiller, rendre visibles, configurer les composants d'une application. KILIM permet d'interfacer ses composants avec JMX, il suffit alors d'utiliser un *template* prévu à cet effet et de spécifier les entités de notre modèle KILIM qui pourront être gérables par JMX.

6.5 Architecture de KILIM

L'architecture de KILIM peut être schématisée de la façon suivante [Hor03] :

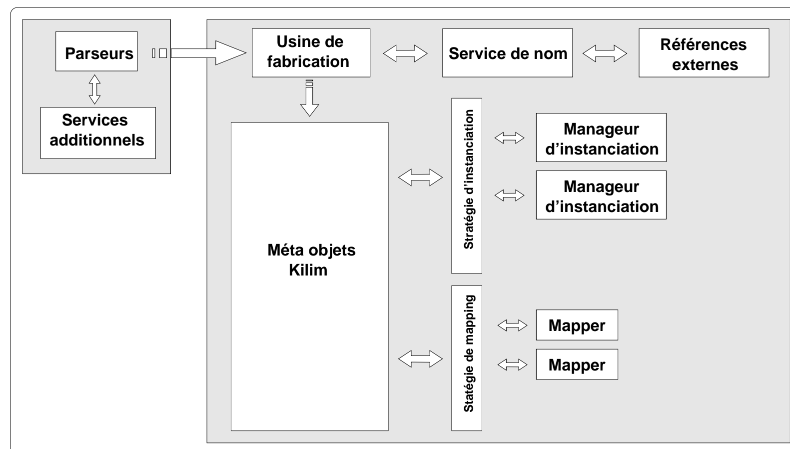

FIG. 8 – Schématisation de l'architecture de KILIM

Le parseur se charge de parser les fichiers de configuration de KILIM (les *templates*) et un mécanisme d'usine de fabrication se charge d'instancier les méta objets décrits par ces fichiers. A l'exécution est donc instanciée une méta représentation de l'assemblage des composants ainsi que toutes les abstractions qui sont définies dans le modèle de KILIM. Le service de noms permet de gérer le nommage des abstractions définies dans les *templates* (le noms des slots, des ports ... etc). Les références externes correspondent à des entités KILIM déjà instanciées dans le système (et dont le développeur a besoin pour éventuellement effectuer des liaisons avec l'application qu'il cherche à instancier). La

méta représentation de l'application sera associée à des mécanismes d'instanciation des objet de l'application ainsi qu'à des règles de mapping explicite entre abstractions du modèle et entités du langage.
Après exécution, si l'application ne maintient plus de référence vers ses méta objets, ces-derniers seront "garbage collectés" par la machine virtuelle.

7 Présentation de JAC

JAC (Java Aspect Components) est un framework pour la construction d'applications (réparties) orienté programmation par aspect* pour Java. JAC est le résultat d'un projet de recherche commun entre les laboratoires CEDRIC, LIP6, LIFL et la société AOPSYS, et depuis peu est intégré au consortium OBJECTWEB.

7.1 La programmation par aspect

La programmation orientée aspect (POA ou AOP* pour *Aspect Oriented Programming*) est un paradigme de programmation qui permet de séparer les préoccupations lors du développement d'une application.
Les préoccupations d'une application peuvent être vues comme un ensemble d'entités, qui composées, permettent d'effectuer les traitements voulus. Cependant, une préoccupation (ou aspect) correspond dans la plupart des cas à une fonctionnalité transversale à l'application, qui peut donc s'avérer difficile à isoler de façon monolithique (par exemple représentée sous forme d'objet pour des fonctionnalités de type sécurité ou persistance). Une application est en fait le résultat d'entrelacements et de chevauchements de sous-problèmes d'ordre technique ou fonctionnel, et conduit à une dispersion du code non-fonctionnel. C'est, partant de cette limite de la programmation orientée objet, que l'équipe de Gregor Kiczales a proposé en 1997 le concept de programmation par aspect, visant à séparer les préoccupations au sein de langages de programmation classique.

L'objectif de ce paradigme est alors de concevoir des composants logiciels indépendants (composants d'aspects fonctionnels et composants d'aspects non-fonctionnels) et de fournir un moyen de les assembler, de manière à augmenter la modularité et la réutilisation d'entités logicielles.

Il faut cependant faire une remarque sur la différence entre les techniques de développement par aspects et les techniques d'assemblage de composants. En effet, les techniques de développement par composants se focalisent sur les règles de composition d'un composant particulier par rapport aux autres, alors que les techniques de développement par aspects se focalisent sur la composition d'un ensemble de composants particuliers et correspondant à une préoccupation (comme la sécurité) par rapport à un autre ensemble de composants (le reste de l'application). En d'autres termes, l'AOP se focalise sur l'intégration de composants.

Dans le paradigme de l'AOP, nous serons amenés à manipuler deux types d'aspects : d'abord les aspects de base, correspondant aux préoccupations métier de l'application, et les aspects non-fonctionnels, spécifiques aux contraintes non-fonctionnelles. Comme schématisé sur la figure 9, ces aspects devront être explicitement identifiés et décomposés selon les exigences de l'application. Chacun de ces aspects devront être implémentés indépendamment les uns des autres de manière à respecter clairement le principe de séparation des préoccupations. La construction de l'application se fera ensuite par l'intermédiaire d'une étape de composition (tisseur ou *weaver*) qui se chargera d'assembler les aspects de base et les aspects non-fonctionnels et qui fournira en sortie un programme monolithique.

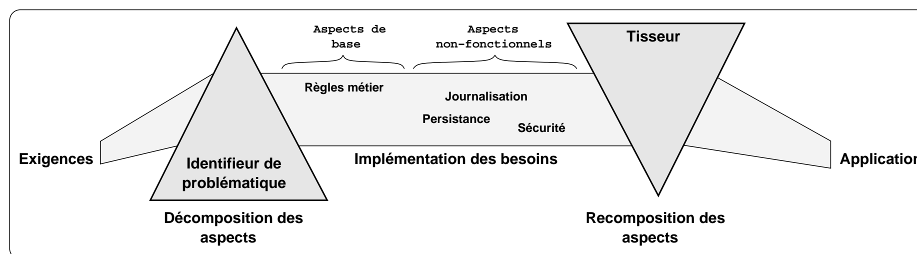

FIG. 9 – Etapes de la programmation par aspects

De manière plus pragmatique, on définit les points de jointure, qui correspondent à des éléments du programme de base comme par exemple des classes, méthodes, exceptions, boucles, affectations (...) et des coupes transversales, qui correspondent à un ensemble de points de jonction ayant un lien logique entre eux. Ainsi, un aspect sera caractérisé par une coupe transversale ainsi qu'une méthode (au sens large du terme, comme par exemple un appel de méthode du programme, une levée d'exception ...) associée à chacun des points de jonction. Un même point de jonction peut intervenir pour introduire les traitements de plusieurs aspects et un aspect peut intervenir sur différents points de jonction. Après cette phase de tissage, le flot d'exécution de l'application est alors identique à celui obtenu si l'application avait été écrite de façon monolithique. La programmation orientée aspect inverse la dépendance entre services et programmes [PDF+02c]. Dans les langages de programmation, les applications effectuent des appels à des primitives de bibliothèques, alors que dans le cas de l'AOP, ce sont les services, implémentés sous forme d'aspects qui modifient la sémantique de l'application.

7.2 Mise en œuvre de la programmation par aspect

Afin d'implémenter la programmation par aspect, il est nécessaire de trouver d'une part une représentation des aspects et d'autre part de les composer. Il existe trois familles de solutions pour la séparation des préoccupations [PDF+02c] : Les solutions fondées sur l'héritage (et le polyporphisme), les solutions transformatives (les aspects sont implantés sous forme de méta-programme ou d'interpréteur qui transforment le programme de base dans le but de produire un nouveau programme avec sa nouvelle sémantique) et les solutions événementielles (le code fonctionnel est lié à l'aspect par l'intermédiaire de points d'entrées dans l'aspect et d'événements bien définis dans le code de base telles que les invocations de méthodes).
Les deux dernières solutions sont implémentées par des mécanismes réflexifs. En effet, les aspects de base (niveau de base) sont composés avec les traitements des aspects non-fonctionnels (niveau méta), et selon la méthode employée, les intercepteurs (cf. 4.3 page 21) peuvent-être un support adéquat pour effectuer la liaison entre les deux niveaux.

7.3 Les objectifs de JAC

Les objectifs de JAC sont de disposer d'un environnement de programmation pour la construction d'applications orientée aspect (et plus particulièrement d'applications distribuées), c'est-à-dire disposer d'une plate-forme :

- D'applications middleware, pour lesquelles la répartition est gérée de manière transparente
- Dynamique, reconfigurable.
- Sans extension syntaxique. Tout est programmé en Java
- Permettant d'adapter du bytecode. Les sources de l'application ne seront jamais modifiées, seules les classes seront manipulées et par l'intermédiaire d'une JVM "normale".
- Proposant des outils pour gérer la composition d'aspects.

La plate-forme JAC dispose de nombreux aspects non-fonctionnels déjà implémentés tels des services de persistance, d'authentification, de cohérence [Paw].

7.4 Le modèle de composant

Le modèle de composants de JAC repose essentiellement autour de la notion de composants d'aspect [PDF+02b]. Un composant d'aspect est l'unité

technique de réutilisation de la plate-forme. Par exemple, les services de persistance et d'authentification dans JAC sont implémentés dans des composants d'aspect qui sont complètement indépendants du reste de l'application.

Ces composants sont hébergés au sein de conteneurs JAC. Contrairement aux conteneurs EJB (qui n'hébergent que des composants métier), les conteneurs JAC hébergent les composants métier (implémentant ce que nous avons déjà nommé les aspects de base de l'application) et les composants d'aspect.

Les composants d'aspect sont des unités d'implémentation qui permettent de définir de manière externe à l'application les caractéristiques transversales à l'ensemble des objets de base qui composent cette application. Ces caractéristiques devront ensuite être tissées par le tisseur (*weaver*) JAC. Ce tisseur fait partie intégrante du conteneur qui héberge les composants, et il sera responsable du déploiement des composants d'aspect sur les aspects de base appropriés. Le code du tisseur définit de manière implicite les points de jonction et effectue le lien entre ces points et les aspects. Cette étape est donc assimilable à une étape de composition dans le sens de la programmation orientée composant, comme nous l'avons déjà vu dans la présentation des plates-formes précédentes.

Un composant d'aspect implémente donc une préoccupation, qui consiste à raffiner ou modifier le comportement de base de l'application. Dans la plateforme JAC, trois points d'entrée sont définis (que nous appelerons dorénavant les trois types de méthodes d'aspect) :

- Les *Wrapping methods* (ou encapsulateurs de méthodes - cf. 4.3 page 21) : Une méthode wrappante peut encapsuler n'importe quelle méthode des objets (ou aspects) de base et est susceptible d'exécuter du code avant et après la méthode initiale. Une méthode de base peut de cette façon être encapsulée par un nombre illimité d'autres méthodes. Ces méthodes wrappantes peuvent être ajoutées et retirées dynamiquement à l'exécution.
- Les *Role methods* : elles peuvent être attachées à plusieurs aspects de base (à l'exécution) et permettent d'étendre les interfaces de ces objets de base.
- Les *Exception handlers* : il s'agit d'une méthode qui est appelée lorsqu'une exception est levée à partir d'une méthode d'un objet de base (ou d'une méthode à laquelle est rattaché cet objet).

Il est très important dans le paradigme de programmation orienté aspects d'appréhender le problème de la composition inter-aspects [PSDF01]. En effet,

la composition de plusieurs aspects au sein de mêmes objets de base pose de nombreux problèmes liés à la compatibilité des aspects (comme par exemple le déploiement d'un aspect lié à un service de redondance, un autre lié à un service de tolérance aux fautes), de dépendances des aspects (un aspect de *binding* devra être couplé avec un aspect gérant un service de nommage), de redondance entre aspects, d'ordonnancement entre aspects ... etc. Dans [PSDF01], ces problèmes de composition d'aspects sont classés en deux catégories : ceux intervenant au moment du tissage proprement dit (qui peut se faire à la compilation ou à l'exécution selon le type de système), et ceux intervenant au moment de l'exécution, qui se manifestent alors après le tissage lors des traitements des aspects sur un point de jonction. Cette deuxième catégorie dépend du contexte d'exécution de l'application, et représente alors une classe de problèmes plus complexes à résoudre. Naturellement, ces problèmes de composition d'aspects ne dépendent pas de la plate-forme JAC mais sont inhérents au concepts de programmation par aspects. Cependant, des mécanismes peuvent être mis en œuvre pour ne pas avoir à reporter toutes les responsabilités sur le programmeur de l'application.

Les objets de base de l'application ne sont composés que des méthodes métier, c'est-a-dire implémentant la logique métier de l'application. Les objets d'aspect (ou composants d'aspect), comme nous l'avons déjà mentionné sont les unités d'implémentation qui permettent de définir les préoccupations (et donc de les séparer). Ils fournissent une interface de configuration qui permet au programmeur d'adapter et d'intégrer de nouveaux aspects au sein d'une application. Nous n'allons pas entrer dans les détails de la composition entre les objets de base et les composants d'aspect, car ils sont fortement liés à l'implémentation de la plate-forme.

7.5 Les étapes de la programmation

La manipulation des concepts de l'AOP se fait par extension des classes du framework et en utilisant l'API de JAC. Deux niveaux de détails sont alors possibles [PDF+02b, Paw02] :

- Le niveau programmation : on programme alors des aspects totalement nouveaux. A ce niveau, le programmeur créé de nouveaux aspects, de nouveaux points de jointure et de nouveaux wrappers pour implémenter les préoccupations transversales de son application.
- Le niveau configuration : on peut configurer les aspects existants au sein du framework (ou des nouveaux aspects programmés). Ce niveau est supporté via un langage de configuration avec une syntaxe générique permettant au programmeur d'invoquer des méthodes de configuration

sur les aspects.

Dans les deux cas, le programmeur devra en premier lieu implémenter les classes de base de l'application.

7.5.1 Le niveau programmation

Les nouveaux composants d'aspects dont implémentés au sein d'une sous-classe de la classe `AspectComponent`. Celle-ci fournit des primitives qui permettent de modifier la sémantique de base de l'application correspondant aux concepts de l'AOP expliqués plus haut. Trois méthodes sont définies pour aspectiser le programme de base :

- A travers la définition de méta-informations, il est possible d'étendre la sémantique des classes de base. Dans JAC, un méta-modèle (RTTI, *Run-Time Type Information*) est défini au moment de l'exécution d'une application. Dans ce cas, des morceaux de code du programme de base (classes, méthodes, champs ...) peuvent être définis comme éléments du RTTI et ainsi étiquetés par des composants d'aspects.
- En implémentant une interface MOP interne, permettant aux composants d'aspects de réagir à certains événements qui se produisent au sein du système.
- En contruisant des points de jointure, c'est-à-dire en ajoutant de nouveaux traitements (avant et/ou après) autour d'exécutions de méthodes du programme de base.

Les points de jointure sont alors définis dans le code en utilisant une sémantique précise, permettant de localiser les méthodes qui devront être aspectisées (en spécifiant le nom de l'objet, la classe associée, la signature de la méthode concernée, le conteneur JAC ...).

Pour utiliser les trois types de méthodes d'aspect, le programmeur devra implémenter des wrappers dynamiques [PDFS01] (classes héritant de la classe `Wrapper`). Un wrapper dynamique est en fait un objet qui définit des comportements de manière à étendre les comportements des objets de base. Au sein de la plate-forme, les wrappers peuvent être instanciés ou détruits à la demande, permettant un tissage dynamique au cours de l'exécution de l'application.

7.5.2 Le niveau configuration

Lorsque l'on programme de nouveaux composants d'aspect, il est possible de les implémenter avec des méthodes qui facilitent une utilisation des aspects par configuration plutôt que par points de jointure codés en dur. Dans ce cas, les objets/classes/méthodes (...) qui devront être aspectisés seront décrits dans un fichier de configuration externe à l'implémentation de l'application (et qui sera alors parsé à l'exécution). Les aspects déjà implémentés au sein du framework JAC sont d'ailleurs accessibles à ce niveau.

Quel que soit le niveau utilisé, JAC propose un aspect de composition qui permet de gérer les problèmes de composition d'aspects évoqués dans le modèle de composant JAC qui peut être utilisé pour activer certaines règles de composition, spécifiant par exemple l'ordonnancement dans lesquels doivent être appliqués les wrappers, et les dépendances et/ou incompatibilités entre différents wrappers.

7.6 Exemple d'implémentation

Nous allons reprendre l'exemple extrait du tutoriel de FRACTAL (cf. 5.4.1 page 38) et tenter de le transposer à JAC.

Nous allons donc considérer que notre application sera composée d'un objet de base (représentant la logique métier, assimilable à l'objet client de l'exemple de FRACTAL) un d'un composant d'aspect (assimilable à l'objet serveur). Pour tenter de transposer l'exemple, le corps de la méthode d'affichage de l'objet client sera vide (mais prendra tout de même en argument une chaine de caractère), puisque dans notre cas, le composant d'aspect sera responsable de l'affichage proprement dit sur la sortie standard. Comme dans le cas précédant, le composant d'aspect dispose d'un attribut `entete` et `nombre`.

Le composant d'aspect va instancier un *wrapper* définissant une méthode *wrappante* qui sera tissée de telle sorte à s'exécuter avec la méthode métier d'affichage de l'objet client. La composition de l'aspect peut donc se schématiser comme nous le montrons sur la figure 10 (nous utilisons la notation UML pour la conception orientée aspect définie dans [PDF+02a]).

7.6.1 Les étapes de la construction

Les étapes de la construction de l'application sont alors les suivantes (dans ce cas, nous utilisons le niveau programmation en construisant notre méthode

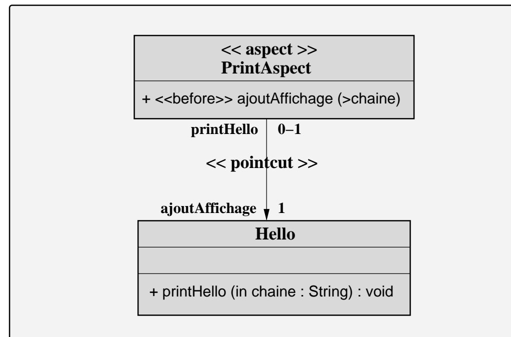

FIG. 10 – Modélisation de la composition de l'objet de base et du composant d'aspect dans le cadre de notre exemple avec JAC

wrappante et notre composant d'aspect mais également le niveau configuration afin de paramétrer notre application - c'est-à-dire le point de jointure et les paramètres des attributs - par l'intermédiaire d'un fichier qui sera alors parsé au début de l'exécution du programme) :

1. **Implémentation du composant d'aspect**
   On implémente notre composant d'aspect que l'on nomme `PrintAspectAC`, il hérite de la classe `AspectComponent` définie dans le package `jac.core`. Dans ce composant, on programme la méthode `ajoutAffichage` qui prend cinq paramètres en arguments, trois chaînes de caractères respectivement pour caractériser sous forme d'expressions régulières l'instance de l'objet métier pour lequel sera tissé l'aspect, le type de classe et le(s) nom(s) de la (des) méthode(s) qui sera wrappée. Le quatrième argument sera une chaîne pour paramétrer l'attribut `entete`, le dernier un entier pour paramétrer l'attribut `nombre` correspondant au nombre de fois que devra être affiché le message sur la sortie standard. Le corps de cette méthode permet juste d'instancier un objet wrappant (avec les paramètres `entete` et `nombre`) et de définir notre point de jointure grâce à la méthode `pointcut`. Pour "construire" ce point de coupure, nous nous servons des expressions régulières données en paramètres, et nous indiquons le nom de la méthode wrappante que nous utilisons : `afficher`.

2. **Implémentation des méthodes wrappantes**
   On implémente notre wrapper, que l'on nomme `PrintAspectWrapper`, cette classe hérite de `Wrapper`. Dans cette classe existe deux attributs : nos attributs `entete` et `nombre`. Le constructeur de cette classe prend en paramètre ces deux attributs (on appelle ce constructeur à partir du composant d'aspect comme expliqué ci-dessus). On implémente alors notre méthode wrappante `afficher`. Elle prend en argument un objet

`Interaction` qui est en quelque sorte une structure de donnée représentant la chaîne wrappante qui entoure la méthode métier initiale. De cette `Interaction` on peut récupérer le paramètre qui a été donné à la méthode initiale (c'est-à-dire dans notre cas la chaîne de caractères à afficher). Le corps de cette méthode consiste donc à afficher `nombre` fois l'`entete` et la chaîne initiale sur la sortie standard avant d'appeler la méthode métier initiale (`before`).

3. **Implémentation des objets de base**
   On implémente la classe métier `Hello` dans laquelle on trouve la méthode `printHello` prenant la chaîne de caractères à afficher en argument. Dans notre cas, le corps de cette méthode est vide, puisque nous considérons l'affichage sur la sortie standard de la chaîne comme étant non-fonctionnel, et donc traité par le composant d'aspect. Dans notre exemple, cette méthode constitue donc une simple interface métier à d'éventuelles autres applications susceptibles de l'utiliser.

4. **Prise en charge de notre nouvel aspect dans la plate-forme**
   On édite le fichier `jac.prop` dans lequel sont définies les propriétés des aspects de JAC. On y ajoute notre aspect que l'on nomme `printaspect` qui s'instancie avec la classe `PrintAspectAC`.

5. **Configuration de l'aspect**
   On édite le fichier `printaspect.acc` qui correspond au fichier de configuration de l'aspect `printaspect`. Dans celui-ci, on paramètre l'aspect en indiquant la méthode qui permettra de le tisser autour de l'objet métier (c'est-à-dire la méthode `ajoutAffichage`) ainsi que les cinq paramètres de cette méthode : `hello0`(1) pour spécifier la première instance (et la seule dans notre cas) de la classe `Hello`, `Hello`(2) pour spécifier le type de classe concernée (dans notre cas, la seule concernée est la classe métier), `printHello`(3) pour spécifier la méthode à wrapper (dans notre exemple, la seule méthode à wrapper est l'unique méthode de la classe métier), puis `Mon_entete >`(4) pour spécifier l'entête à paramétrer et `4`(5) pour (par exemple) demander à la méthode wrappante d'afficher 4 fois la chaîne passée en argument.

6. **Enumération des aspects à tisser à notre application**
   On édite le fichier `hello.jac` qui correspond au paramétrage des aspects pour l'ensemble de l'application. Ici, nous nous contentons d'indiquer que nous voulons que l'aspect `printaspect` soit tissé. A ce stade, la vue d'ensemble de notre application peut être schématisée de la manière suivante :

7. **Exécution de l'application**
   On implémente une classe `Run` avec la méthode `main` qui instancie une

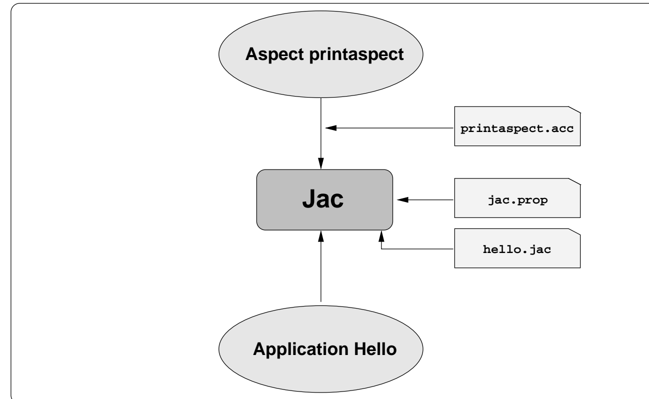

FIG. 11 – Schématisation de la vue d'ensemble de notre application JAC

nouvelle classe `Hello` puis on exécute notre application.

7.6.2 La reconfiguration dynamique

L'API de JAC permet de tisser ou détisser des aspects dynamiquement après l'exécution de l'application mais nous n'avons pas exploré cette possibilité.

7.7 Architecture de JAC

L'architecture de JAC est schématisée sur la figure 12 [PSDF01].

- Le programme de base représente l'ensemble des objets de base modélisant la logique fonctionnelle de l'application.
- Les composants d'aspects représentent la logique non-fonctionnelle de l'application, ils sont implémentés par un ensemble d'objets d'aspect (les trois types de méthodes d'aspect que nous avons définis).
- Le tisseur est responsable du déploiement des aspects sur les objets de base, il est configuré par le fichier `jac.prop` qui définit où et quand les composants d'aspects doivent être déployés.
- L'aspect de composition permet de définir les règles de composition des aspects.

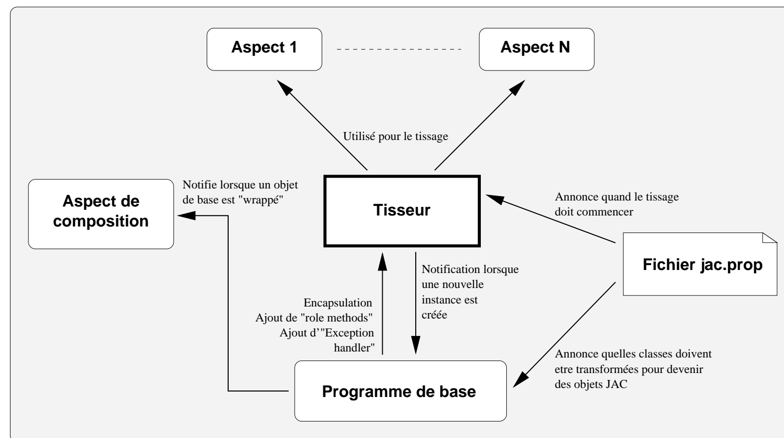

FIG. 12 – Schématisation de l'architecture de JAC

Dans cette deuxième partie, nous avons présenté les trois plates-formes FRACTAL, KILIM et JAC, leurs modèles de composants et d'assemblage. L'exemple minimal d'implémentation nous a permis de bénéficier d'une vision pragmatique du paradigme de programmation orientée composants d'une part et du paradigme de programmation orientée aspects d'autre part.

La prochaine partie de ce rapport est destinée à la description de l'implémentation d'un prototype d'application répartie.

Troisième partie

Implémentations

L'objectif de notre stage est de comparer différentes approches de développement pour la construction d'applications réparties à travers trois plates-formes fondées sur la séparation des préoccupations.

Dans la deuxième partie de ce rapport, nous avons présenté les modèles de composants et d'assemblage de FRACTAL, KILIM et JAC. Dans la suite logique de notre étude, il était donc nécessaire d'étudier l'implémentation d'un prototype d'application afin de bénéficier d'une comparaison plus pragmatique et exhaustive.
L'objectif de cette implémentation est donc de tenter d'appréhender le développement d'applications réparties en fonction de la plate-forme d'accueil, de manière à comparer les phases de conceptions et d'implémentations de notre application.

Dans la suite de ce rapport, nous allons présenter les deux algorithmes que nous avons choisi d'implémenter, nous expliquerons les simplifications que nous leurs avons apportées, les choix d'implémentation qui nous ont paru judicieux. Nous détaillerons les étapes de notre réflexion sur les manières dont nous avons décidé de séparer les préoccupations de notre application, étape essentielle de notre travail.
Pour chacun des algorithmes, nous donnerons un modèle de conception de l'application indépendamment de la plate-forme d'accueil (ainsi qu'une implémentation en JAVA "pur" correspondant à ce modèle de manière à bénéficier d'un critère de comparaison supplémentaire). Puis nous donnerons les modèles de conception et d'assemblages des différents composants pour chacune des plates-formes FRACTAL, KILIM et JAC. Nous montrerons ainsi que toutes les étapes du cycle de vie d'un logiciel sont fortement dépendantes de la vision des différentes fonctionnalités que nous avons séparées, de la plate-forme d'accueil, et que la manière d'appréhender une application n'a pas beaucoup de points en commun entre une approche orientée composants techniques et une approche orientée programmation par aspects ...

8 L'algorithme de KAI LI et PAUL HUDAK

Nous avons choisi comme modèle d'implémentation l'algorithme de KAI LI et PAUL HUDAK [LH86]. Il s'agit d'un algorithme de gestion de mémoire virtuelle répartie partagée.
La mise en oeuvre de l'algorithme est complètement logicielle (couche applicative), elle est donc indépendante du noyau du système d'exploitation utilisé.
Le choix de cet algorithme est intéressant dans le cadre de notre comparaison puisqu'il est représentatif de nombreuses applications réparties : il aborde les problèmes de cohérences de données, de nommage, de localisation, de communications distantes. On peut également considérer que cet algorithme est proche des problématiques de nombreux domaines du réparti telles la tolérance aux pannes, l'équilibrage de charge ou encore la gestion de caches.

8.1 Contexte et hypothèses

On fait les hypothèses que les données peuvent être répliquées et qu'il n'existe qu'une seule copie de référence dans le système. On a donc exclusivement un écrivain ou N lecteurs. En univers réparti, les communications sont considérées comme fiables.
L'algorithme est fondé sur la notion de propriétaire de la copie de référence, utilisant un mécanisme d'invalidation sur écriture.
Il existe une fonction de localisation du site détenteur de la copie de référence.

8.2 Les alternatives possibles de l'algorithme

Différentes déclinaisons de l'algorithme de KAI LI et PAUL HUDAK ont été proposées :

- **L'algorithme centralisé :** Le service de localisation de la copie de référence est centralisé, un site (serveur) fournit alors un service de type annuaire aux sites clients pour trouver la copie de référence. Le *copyset* (liste de site ayant une copie de la donnée) est également centralisé, maintenu par le site propriétaire de la copie de référence.

- **L'algorithme décentralisé avec notion de propriétaire probable de la copie de référence :** Dans ce cas, on utilise une heuristique de localisation pour trouver la copie de référence dans le système. Cependant, le *copyset* reste centralisé sur le propriétaire.

- **L'algorithme décentralisé avec notion de propriétaire probable et *copyset* distribué :** Le *copyset* est alors réparti sur l'ensemble des sites qui disposent d'un exemplaire de la copie de référence.

Cette dernière alternative de l'algorithme peut se transposer à un système Peer To Peer (P2P). En effet, dans un tel système, les sites communiquent deux à deux pour s'échanger des informations, chaque site peut être considéré comme client et serveur, ce qui est tout à fait le cas dans le contexte de l'algorithme décentralisé.

De plus, deux alternatives sont à considérer dans le cadre de la mise en oeuvre du mécanisme d'invalidation :

- Mécanisme d'invalidation **synchrone** : chaque invalidation est alors acquittée, on se rapproche alors de la mise en oeuvre d'une cohérence atomique (la plus stricte possible).
- Mécanisme d'invalidation **asynchrone** : les invalidations ne sont pas acquittées (mais considérées comme fiables), la cohérence est alors de type séquentielle.

Dans le cadre de notre stage, nous avons décidé d'implémenter deux versions de cet algorithme :

- La version centralisée
- La version décentralisée avec *copyset* distribué

Dans les deux implémentations, nous avons utilisé un mécanisme d'invalidation synchrone. Mais il faut remarquer que ce dernier choix (invalidation synchrone ou asynchrone) ne change rien au comportement de notre application simulée comme nous l'expliquerons dans la suite du rapport.

9 Présentation de l'algorithme centralisé

Donnons une description détaillée du fonctionnement de l'algorithme de KAI LI et PAUL HUDAK en version centralisée (et mécanisme d'invalidation synchrone). Dans ce cas, rappelons que le service de localisation de la copie de référence est centralisé sur un site annuaire. Il en est de même pour le *copyset*, géré de manière centralisée par le site propriétaire de la copie.

9.1 Les structures de données

Chaque site dispose d'une structure de données que l'on nomme `TableDesPages` composée d'un ensemble de tuples permettant de stocker les informations relatives à toutes les pages mémoire actives dans le système. Les champs de ces tuples sont les suivants :

- **Numéro :** il permet d'identifier de manière unique une page active dans le système.
- **Copyset :** il correspond à une liste de sites ayant une copie de la page concernée. Cette liste n'a de sens que si le site est propriétaire de la page.
- **Droits :** il s'agit des droits d'accès du site pour la page concernée. Ils peuvent être de trois types : `l` (si la page est accessible en lecture), `e` (si la page est accessible en écriture, le site est alors propriétaire de la page) ou invalide (le contenu de la page est alors incohérent ou indéfini).
- **Propriétaire :** indique si le site sur lequel est la page est propriétaire de celle-ci (le propriétaire est le dernier site qui a écrit sur une page, on dit qu'il détient la copie de référence).

Le serveur (l'annuaire qui pour chaque page connaît le propriétaire courant) dispose d'une structure de données que l'on nomme `TableInfo`, composée de tuples dont les champs sont les suivants :

- **Numéro :** Le numéro de la page.
- **Propriétaire :** Le numéro du site qui est propriétaire de la page concernée.

9.2 Déroulement du protocole

Dans cette partie, nous donnons une description détaillée de la mise en oeuvre du protocole.

L'algorithme proposé par Kai Li et Paul Hudak spécifie trois types de fonctionnalités en lecture et en écriture :

- Le site effectuant un défaut (en lecture ou en écriture).
- L'annuaire (le serveur) qui maintient une table de correspondance entre les pages du système et leurs propriétaires.
- Le site répondant à une demande générée par un demandeur (effectuant un défaut). Ce site est alors propriétaire de la page.

9.2.1 L'algorithmie des différentes fonctions

Les différentes fonctions du protocole sont décrites par les étapes suivantes :

Défaut en lecture sur un site i :

1. Envoyer sur l'annuaire : [i, lecture, page]
2. Recevoir du proprietaire : [page, contenu]
3. Mise a jour : page.contenu ← contenu
4. Mise a jour : page.droits ← lecture

Réception d'une demande en lecture sur l'annuaire :

5. Recevoir d'un demandeur : [demandeur, lecture, page]
6. Envoyer au proprietaire de la page : [demandeur, lecture, page]

Réception d'une demande en lecture sur le propriétaire :

7. Recevoir de l'annuaire : [demandeur, lecture, page]
8. Mise a jour : page.droits ← lecture
9. Mise a jour : page.copyset ← page.copyset + demandeur
10. Envoyer au demandeur : [page, page.contenu]

Défaut en écriture sur le site i :

11. Envoyer a l'annuaire : [i, ecriture, page]
12. Recevoir de l'ancien proprietaire : [page, contenu, liste_lecteurs]
13. Mise a jour : page.contenu ← contenu
14. Mise a jour : page.droits ← ecriture
15. Mise a jour : page.proprietaire ← vrai
16. Procedure Invalider pour (page, page.copyset)

Procédure Invalider :

17. Pour tout site j du copyset : Envoyer a j [Invalider, page], Recevoir de j [ok, page]

Réception d'une demande en écriture sur l'annuaire :

18. Recevoir d'un demandeur : [demandeur, ecriture, page]
19. Envoyer au proprietaire de la page : [demandeur, ecriture, page]
20. Mise a jour : page.proprietaire ← demandeur

Réception d'une demande en écriture sur le propriétaire :

21. Recevoir de l'annuaire : [demandeur ecriture, page]
22. Mise a jour : page.droits ← invalide
23. Envoyer au demandeur : [page, page.contenu, liste_lecteurs]
24. Mise a jour : page.contenu ← vide
25. Mise a jour : page.proprietaire ← faux

Traitement effectué lors d'une demande d'invalidation :

26. Recevoir d'un demandeur : [Invalider, page]
27. Mise a jour : page.contenu ← vide
28. Mise a jour : page.proprietaire ← faux

9.2.2 Schématisation du déroulement du protocole

Les différentes étapes des défauts en lecture et écriture peuvent être schématisées de la manière suivante :

Le site noté `L` effectue un défaut en lecture (figure 13 page 71).

Le site noté `E` effectue un défaut en écriture (figure 14 page 71).

9.2.3 Simplifications de l'algorithme

Nous avons donné une version simplifiée de l'algorithme initialement proposé par Kai Li et Paul Hudak. En effet, ce-dernier prend en charge le fait que les requêtes en écriture et/ou en lecture s'effectuent de manière concurrentes dans le système. Ainsi, de manière à préserver la cohérence des accès aux pages mémoire, une mécanisme de verrouillage doit être mis en place de manière à ce que certains morceaux de l'algorithme s'exécutent en exclusion

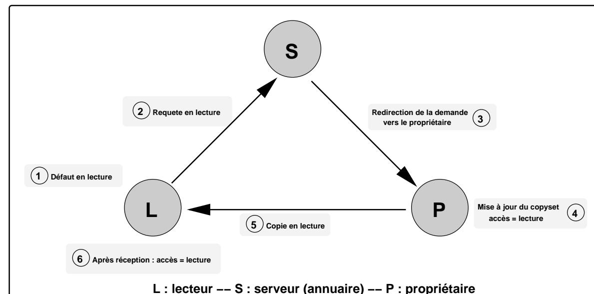

FIG. 13 – Défaut en lecture effectué sur le site `L` dans le cadre de l'algorithme centralisé

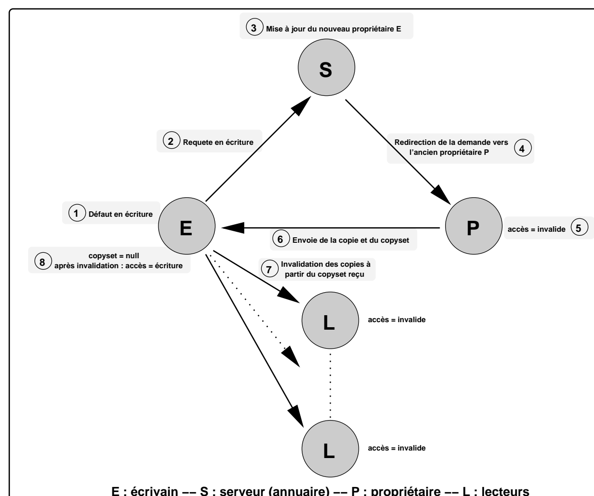

FIG. 14 – Défaut en écriture effectué par le site `E` dans le cadre de l'algorithme centralisé

mutuelle.
Ainsi, sur les sites est assigné un verrou pour chacune des pages (ce verrou fait alors partie de la structure `TableDesPages`). Il permet d'exécuter en exclusion mutuelle les défauts/réceptions en lecture/écriture, c'est-à-dire autour des lignes (description détaillée de l'algorithme (cf. 9.2.1 page 69)) 1-4, 7-10, 13-16 et 22-28.
Il en est de même du côté du service d'annuaire, où l'utilisation d'un sémaphore permet de rendre exclusif les accès aux lignes 6 et 19-20.
La file d'attente des verrous associés à chaque page et du sémaphore sont alors gérés en première requête arrivée, première requête servie.

Nous avons choisi volontairement d'omettre ces aspects de gestions d'ex-

clusions mutuelles [2] car (comme nous le verrons plus loin) dans le prototype d'application que nous avons réalisé, tous les accès se font de manière séquentielle. En effet, dans le cadre de ce prototype, nous restons dans le domaine de la simulation, sans nous éloigner de l'objectif principal de notre stage qu'est la comparaison de différentes approches, et non la réalisation d'une application complètement fonctionnelle utilisable en production.
A l'exécution, la simulation est réalisée par l'intermédiaire d'appels de méthodes sur les différents sites, via une fonction **main** qui séquentialise donc tous les accès, il n'y a donc pas d'incohérence possible entre ceux-ci. De plus, une implémentation multi-threadée aurait grandement compliqué la transposition du code de l'application sur les différentes plates-formes.
Cette remarque est également valable par rapport au choix du mécanisme d'invalidation. En effet, qu'il soit synchrone ou asynchrone, cette séquentialisation des accès nous permettera de toujours observer une cohérence atomique au sein du système.

[2] Dans le cadre de l'implémentation de l'algorithme décentralisé (cf. 11 page 96), nous avons cependant pris en charge la gestion de ces exclusions mutuelles pour ajouter un aspect non-fonctionnel à notre modélisation même si cela ne change rien à la cohérence des données comme nous l'expliquons.

10 Première implémentation

Cette partie traite de l'implémentation de l'algorithme de KAI LI et PAUL HUDAK en version centralisée (c'est-à-dire annuaire centralisé, cf. 8.2 page 66) et *copyset* centralisé.

Le prototype de notre application répartie a été réalisé en JAVA "pur", puis à l'aide des trois plates-formes FRACTAL, KILIM et JAC.

10.1 Contexte et hypothèses

Expliquons de manière claire le contexte dans lequel nous nous plaçons pour implémenter ce premier prototype d'application répartie.

10.1.1 Contexte général

Tout d'abord, dans le cadre de l'algorithme proprement dit, on suppose qu'il n'y a jamais de panne, que chaque site communique avec les autres au travers d'un canal fiable (premier message émis, premier message reçu, ordre FIFO). Il n'y a par conséquent pas de perte de message ni d'erreur de transmission.

En ce qui concerne notre prototype, nous considérons que notre système est constitué d'un nombre statique de machines, et donc que nous n'avons pas à gérer l'ajout ou le retrait dynamique de celles-ci. De plus, lorsque le système est fonctionnel, toutes les liaisons distantes entre machines sont établies, chaque entitié du système est alors connue de toutes les autres.

Notre application sera composée de machines clientes (qui effectuent des lectures et des écritures sur les objets sur lesquels porte notre service de gestion de la cohérence), d'un serveur annuaire implémentant le service de localisation des propriétaires des objets (c'est-à-dire des pages mémoires) et d'un serveur de *logs* proposant un service centralisé de journalisation. Dans un premier temps, nous pouvons considérer que l'intégration d'un tel service centralisé paraît étrange dans le cadre d'une application répartie puisque nous utilisons des ressources réseau pour journaliser des informations (accès aux objets, accès à l'annuaire, demandes en lecture et/ou écriture ...) dont la granularité est semblable à celle des informations qui sont échangées pour assurer le service de cohérence. Cependant, n'oublions pas que nous restons dans le domaine de la simulation, et ce service a été implémenté dans le seul but d'ajouter un nouvel

aspect non-fonctionnel à notre application et ainsi de montrer comment il est possible de l'intégrer au sein des trois plates-formes que nous avons étudiées. De plus, de nombreuses alternatives liées à la journalisation sont envisageables, par exemple une mise à jour des *logs* dans des fichiers locaux aux postes, une gestion d'un cache local aux postes avec envois périodiques à une base de données distante ...etc.
Les machines clientes sont celles qui effectuent les défauts en lecture et écriture.

10.1.2 Choix et simplifications au niveau de l'implémentation

Notre prototype d'application sera composé de trois machines clientes et de deux serveurs (annuaire et journalisation) :

- Un client est susceptible de lire et d'écrire sur les objets sur lesquels porte la cohérence, ainsi que d'"émettre" des défauts sur ces objets. Il dispose également d'une interface accessible aux autres sites pour répondre à des demandes de lecture et d'écriture sur des objets dont il est propriétaire (ou des invalidations).
- Le serveur annuaire dispose d'une interface permettant aux clients d'invoquer des requêtes de localisation des sites propriétaires des objets concernés.
- Le serveur de journalisation propose une simple interface proposant à l'annuaire ou aux clients d'envoyer une chaine de caractères qui sera alors traitée localement par le serveur (dans notre cas, un simple affichage sur la sortie standard !).

Les entités sur lesquelles portent la cohérence sont de simples entiers encapsulés dans un objet `Page` (dans la suite du rapport, lorsque nous parlerons de page mémoire, nous ferons référence à cet objet `Page`). Seules trois pages sont actives dans le système.

Le processus d'initialisation est très simple : chaque site client est propriétaire d'une page mémoire dont le numéro est celui du numéro du site (0, 1 et 2). Comme nous l'avons évoqué, nous considérons que le système est complètement statique, le nombre de pages n'évolue pas au cours de l'exécution. Chaque site a donc la connaissance de toutes les pages mémoire, ainsi que le serveur annuaire initialisé lui aussi avec sa structure `TableInfo` dont les éléments pointent vers les sites propriétaires des pages respectives.

Nous avons choisi d'utiliser JAVA RMI pour gérer la répartition de notre prototype. Remarquons qu'avec cette approche, les communications distantes se font en RPC et qu'il peut s'avérer compliqué d'implémenter de façon totalement asynchrone les communications entre sites (telles qu'elles le sont énoncées

dans l'algorithme de KAI LI et PAUL HUDAK dans un environnement concurrent). Schématisons les deux processus dans l'exemple d'un défaut de lecture :

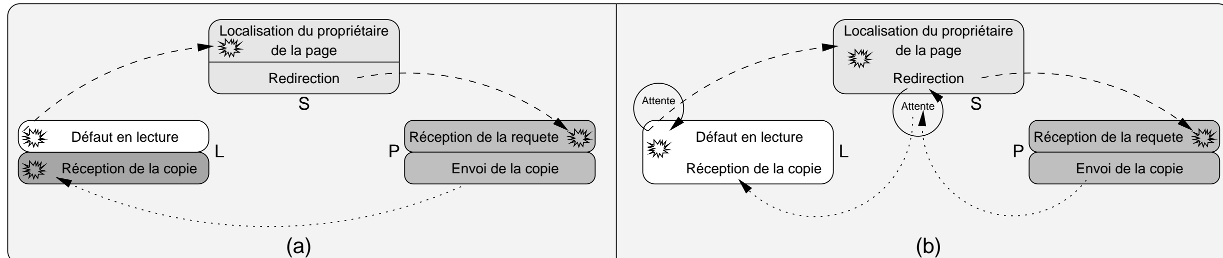

FIG. 15 – Comunications asynchrones (a) et synchrone (RPC de RMI) (b) dans le cas d'un défaut en lecture (algorithme centralisé)

Cette figure met en avant les différences en terme d'asynchronisme et d'accès concurrents aux pages mémoire entre les spécifications de LI et HUDAK (qui s'appliquent au cas réel d'utilisation d'un tel algorithme) et notre application de simulation.
Dans le cas (a) se rapprochant le plus des spécifications de l'algorithme énoncé, les communications se font de manière complètement asynchrones, c'est le propriétaire de la page sur laquelle a été effectué le défaut qui contacte directement le site source. Avec cette vision des choses, on peut imaginer une implémentation multi-threadée de l'application permettant (en plus de l'asynchronisme) une exécution concurrente des défauts sur les pages.
Avec l'utilisation de RMI, la réponse du site propriétaire est interceptée par le serveur annuaire qui à son tour redirige la page vers le site source. La chaîne d'événements de rapatriement du contenu de la page est donc atomique, les processus mis en cause sur le site lecteur et sur le serveur sont donc bloquants. On aurait cependant pu imaginer d'implémenter une version multi-threadée de manière à simuler des accès concurrents, mais nous avons déjà expliqué qu'ajouter cette complexité dans le prototype n'aurait pas apporter de résultats pertinents quant aux objectifs du stage (cf. 9.2.3 page 71). De plus, pour disposer d'un environnement de concurrence réaliste, il aurait fallu faire tourner l'application sur de "vrais" sites distants et non en simulation locale comme il est possible de le faire en RMI (et c'est cette méthode que nous avons utilisée).
Nous considérons donc acceptable ces simplifications d'implémentation dans le cadre de notre étude.

10.1.3 Les échanges RMI

Chaque site client dispose d'une interface `FonctionsClient` dans laquelle sont déclarées les méthodes `demandeLecture`, `demandeEcriture` et `invalider` (permettant respectivement de retourner le contenu d'une page, de retourner une liste de sites dont la page est à invalider et d'invalider une page particulière).
Le serveur annuaire implémente une interface `FonctionsServeur` avec les méthodes `lire` et `ecrire` permettant de localiser un site propriétaire pour une lecture ou une écriture.
Le serveur de journalisation dispose d'une interface `FonctionsServeurLogs` implémentant une méthode `recevoirLogs` pour réceptionner une chaîne de caractère.

Schématisons l'ensemble des communications RMI qui interviennent dans le cadre d'un défaut en écriture (avec les arguments de chaque méthode) :

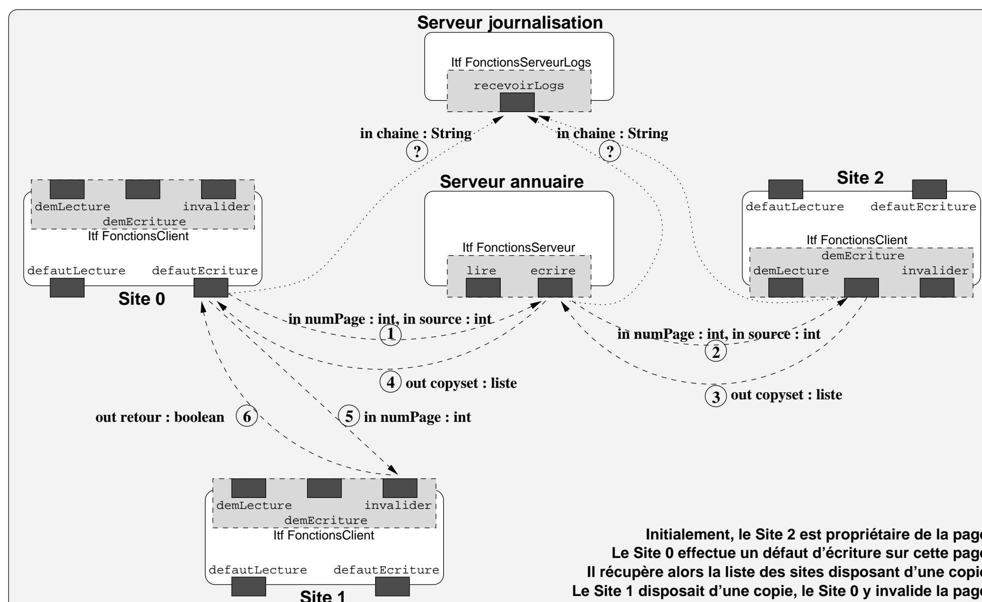

FIG. 16 – Ensemble des communications RMI dans le cadre d'un défaut en écriture (algorithme centralisé)

Nous n'avons volontairement pas signalé à quel moment interviennent les invocations à la méthode distante `recevoirLogs` car cela dépend des étapes que l'on veut journaliser.

10.2 La séparation des préoccupations

Il faut avouer que la séparation des préoccupations en sein de l'implémentation de l'algorithme de KAI LI et PAUL HUDAK n'a pas été facile. Encore une fois en sein de notre prototype, nous avons grandement simplifié les choses de manière à en limiter la complexité.

Les différentes fonctionnalités que nous pouvons attribuer à notre application sont les suivantes :

- La lecture et l'écriture des entités sur lesquelles porte la cohérence. Il s'agit donc de deux opérations très simples : la première est de type `lire (in numeroPage : int) : int` (sa signature dépend bien sûr des choix d'implémentation que nous avons expliqués, cf. 10.1.2 page 75). Cette opération effectue en une étape la lecture de la page mémoire (donnée en argument) dans une structure de données et renvoie son contenu. La deuxième est de type `ecrire (in numeroPage : int, in contenu : int) : void`. En une étape, cette opération écrit la nouvelle valeur qui doit être assignée à une page mémoire.

- L'entité sur laquelle porte la cohérence : nous avons expliqué que nous avons choisi un simple entier encapsulé dans un objet `Page`. Cette modélisation des données qui s'échangent dans notre système peut être interprétée de différentes manières si l'on transpose notre simulation à un cas réel : il peut s'agir d'un octet, d'une page mémoire, d'un fichier dans son ensemble ...

- La gestion de la cohérence des données. Elle est alors assurée par une grande partie de l'algorithme de LI et HUDAK. Une segmentation de cette fonctionnalité est encore possible si l'on considère que la cohérence est assurée par les mécanismes de sémaphore côté serveur et verrou côté clients (que nous n'implémentons pas puisque les accès aux données ne se font pas de manière concurrente) et le mécanisme d'invalidation (définissant la manière dont sont ordonnés les accès à une page).

- La localisation, c'est-à-dire le mécanisme qui permet de localiser le propriétaire de la copie de référence d'une page donnée. Nous avons expliqué (cf. 8.2 page 66) qu'il existe d'ailleurs plusieurs alternatives.

- Le nommage, c'est-à-dire comment nommer au sein du système les entités sur lesquelles porte la cohérence.

- Les communications distantes. Nous sommes en environnement réparti, la manière dont vont s'effectuer les communications entre sites constitue

donc une fonctionnalité.

- La journalisation, c'est-à-dire le mécanisme permettant de journaliser d'une certaine manière (dans notre cas, une simple écriture d'informations sur la sortie standard) certaines étapes de l'exécution de l'application.

Bien évidemment, nous n'avons pas pris en compte dans notre implémentation une séparation des préoccupations aussi fine que celle exposée ci-dessus. En ce qui concerne le premier point (cf. 3.1 page 17), nous avons décidé d'une séparation très simple entre fonctionnel et non-fonctionnel : Les opérations `lire` et `ecrire` sont du domaine du fonctionnel (métier), tout le reste sera considéré comme non-fonctionnel. Ce choix parraît très logique dans le sens où dans un cas réel, les applications utilisant le système de fichiers répartis décrit par l'algorithme de Li et Hudak n'auraient connaissance que de ces deux opérations, elles constitueraient donc l'interface entre le système en production et la gestion de la cohérence. Ces deux opérations constituent donc la préoccupation métier de notre application.

En ce qui concerne les aspects non-fonctionnels :

L'aspect cohérence des données demeure une préoccupation non-fonctionnelle essentielle qu'il est important de séparer du reste.
Nous considérons que le service de journalisation constitue une préoccupation à part entière.

La localisation des propriétaires des pages mémoire au sein de l'algorithme n'est assurée que par le service d'annuaire centralisé, cette fonctionnalité est donc propre au serveur.

L'entité sur laquelle porte la cohérence (la modélisation de la page, le contenu de type entier) est déterminée de manière statique, nous ne la considérons pas comme une préoccupation.
La nomination des pages mémoire est également statique (les clients et l'annuaire ont une connaissance globale du nombre de pages actives dans le système et de leur nommage par entier), nous considérons donc que cette fonctionnalité est trop intimement mêlée au coeur du programme pour pouvoir la séparer de manière intéressante.
Nous partons du principe que les communications distantes sont assurées par la couche middleware (Java RMI), toutes les fonctionnalités liées à la répartition sont donc assurées par cette couche. Lorsque nous parlons de répartition ici, nous parlons des fonctionnalités de bas niveau telles les envois de messages

avec paramètres, ordre sur les messages, type de protocole réseau utilisé (...) et non de la gestion de la répartition au sein de l'algorithme.

Pour revenir sur les problèmes exposés plus haut (cf. 3.1 page 17), nous pouvons donc dire que nous avons choisi une granularité importante dans le choix de nos séparations puisque nous considérons certaines fonctionnalités comme étant trop mêlées au programme pour les séparer.
Notre vision de la séparation des préoccupations peut être schématisée de la manière suivante :

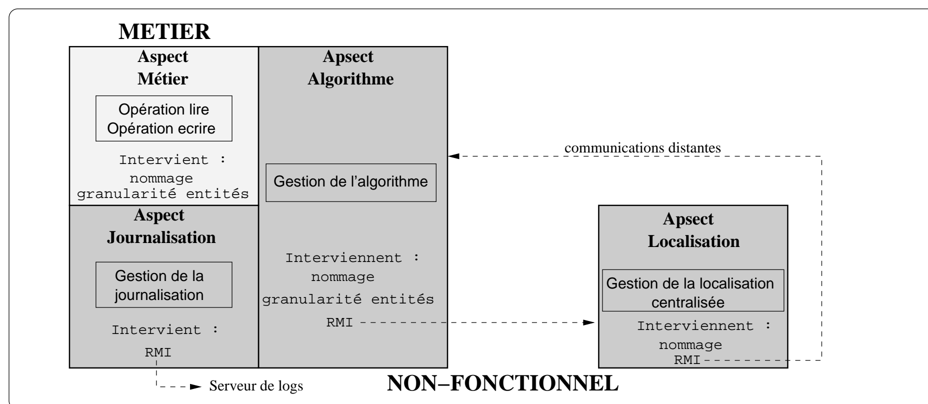

FIG. 17 – Schématisation de la séparation des préoccupations de notre application (côté client) de l'algorithme centralisé

Avec cette vision des choses, nous aurions pu également intégrer un aspect de journalisation sur le serveur annuaire de manière à journaliser les accès à ce serveur mais nous avons décidé de ne pas compliquer les choses davantage avec cette redondance (puisque l'aspect journalisation est pris en compte sur client). Par contre, on peut faire une différence entre la journalisation métier et la journalisation liée à l'algorithme. Dans le premier cas, on journalise les accès aux informations métier, dans le deuxième cas, les accès aux informations manipulées par l'aspect algorithme.

10.3 Version implémentée en JAVA

Dans un premier temps, nous nous sommes concentrés sur une implémentation de notre prototype en JAVA pur.

10.3.1 Modélisation

Le modèle de notre prototype d'application est donné par le schéma de la figure 18.

Comme nous l'avons déjà expliqué, nous simulons une page mémoire par la classe `Page` qui ne possède qu'un seul attribut correspondant à la valeur contenue par cette page. Nous avons volontairement introduit la classe intermédiaire `TablePage` car pour certaines implémentations nous avons trouvé intéressant de pouvoir partager une référence d'une instance de cette classe entre deux composants logiciels.
La classe `RetourEcriture` est utilisée pour retourner les informations nécessaires au site ayant provoqué un défaut en écriture.
La classe `ServeurImpl` détient toutes les références RMI des sites clients stockées dans le vecteur `fonctionsClient`.
Les méthodes `lookupClients` des classes `Site` et `ServeurImpl` permettent d'invoquer les *lookups* RMI après les instanciations de toutes les classes, puisqu'elles doivent être toutes *bindées* auprès du *registry* RMI avant d'effectuer ces liaisons. Les classes qui héritent de `UnicastRemoteObject` sont celles pour lesquelles il faudra construire un *stub* RMI, celles réalisant `Serializable` correspondent aux objets qui transiteront sur le réseau.
Les méthodes `lectureLocale` et `ecritureLocale` permettent de simuler les défauts en lecture et écriture.

Toutes les instanciations et simulations sont effectuées à partir de la classe `Main`.

Remarquons que dans cette modélisation, aucune séparation des préoccupations n'a été effectuée. En effet, la classe `Site` rassemble à elle seule tous les aspects évoqués (figure 17 page 80).
Nous aurions pu nous y prendre de manière différente, en tentant de diviser les fonctionnalités de la classe `Site` en différentes classes. En quelque sorte, cette modélisation offre une séparation des préoccupations liée à la répartition : à chaque site réparti est attribuée une classe modélisant toutes les fonctionnalités de ce site.

10.3.2 Remarques sur l'implémentation

Nous avons donc implémenté les fonctionnalités de l'algorithme de gestion de la cohérence dans la classe `Site` côté client, et dans la classe `ServeurImpl` pour le mécanisme de localisation des propriétaires des pages. Les appels RMI consistant à envoyer une chaîne de caractères au serveur de logs sont dispersés

FIG. 18 – Modélisation de l'application implémentée en JAVA pur (algorithme centralisé)

dans le code de la classe `Site`.
Pour effectuer les *lookups* RMI, nous nous servons de la chaîne `nomClient` (dans les classes `Site` et `ServeurImpl`) à laquelle on concatène le numéro du site (c'est-à-dire de 0 à 2).

10.3.3 Exécution et déploiement

Dans la fonction `main`, on instancie un objet `ServeurImpl` et un objet `ServeurLogsImpl` que l'on *bind* dans le *registry* RMI (lancé avant l'exécution). On instancie trois objets `Site` que l'on *bind* également. On appelle la méthode `lookupClients` sur le serveur et sur chaque site de manière à mettre à jour toutes les références RMI croisées. On peut alors commencer notre simulation.

10.4 Version implémentée avec FRACTAL

10.4.1 Le modèle d'implémentation

Le modèle d'implémentation de notre application (figure 19 page 84) avec la plate-forme FRACTAL est relativement proche de celui proposé pour l'implémentation en pur JAVA (figure 18 page 82) (les classes pour lesquelles les attributs ou méthodes sont notés par "..." expriment le fait que leurs contenus sont identiques à ceux présentés dans la modélisation en JAVA pur).

FRACTAL est un modèle de composants techniques, la séparation des préoccupations s'effectue donc par séparation de classes au niveau de la modélisation. Comme nous pouvons le faire remarquer sur la figure ci-dessus, du côté client, nous avons séparé les fonctionnalités comme nous l'avons déjà expliqué précédemment (figure 17 page 80) : L'aspect métier de notre composant client est implémenté par la classe `SiteFonctionnelImpl`, l'aspect non-fonctionnel noté algorithme par la classe `SiteNonFonctionnelImpl` et l'aspect non-fonctionnel de journalisation par le classe `ComposantLogsImpl`. La classe métier implémente donc les deux opérations métier de lecture et d'écriture, ainsi que les trois méthodes de l'interface `UserBindingController` qui permettent de spécifier au sein du modèle FRACTAL les opérations de liaisons des composants. Les deux classes non-fonctionnelles implémentent toutes les méthodes respectivement relatives à la gestion de l'algorithme et du processus de journalisation. La classe `SiteNonFonctionnelImpl` implémente également `UserBindingController` car (comme nous le verrons plus loin figure 20) le

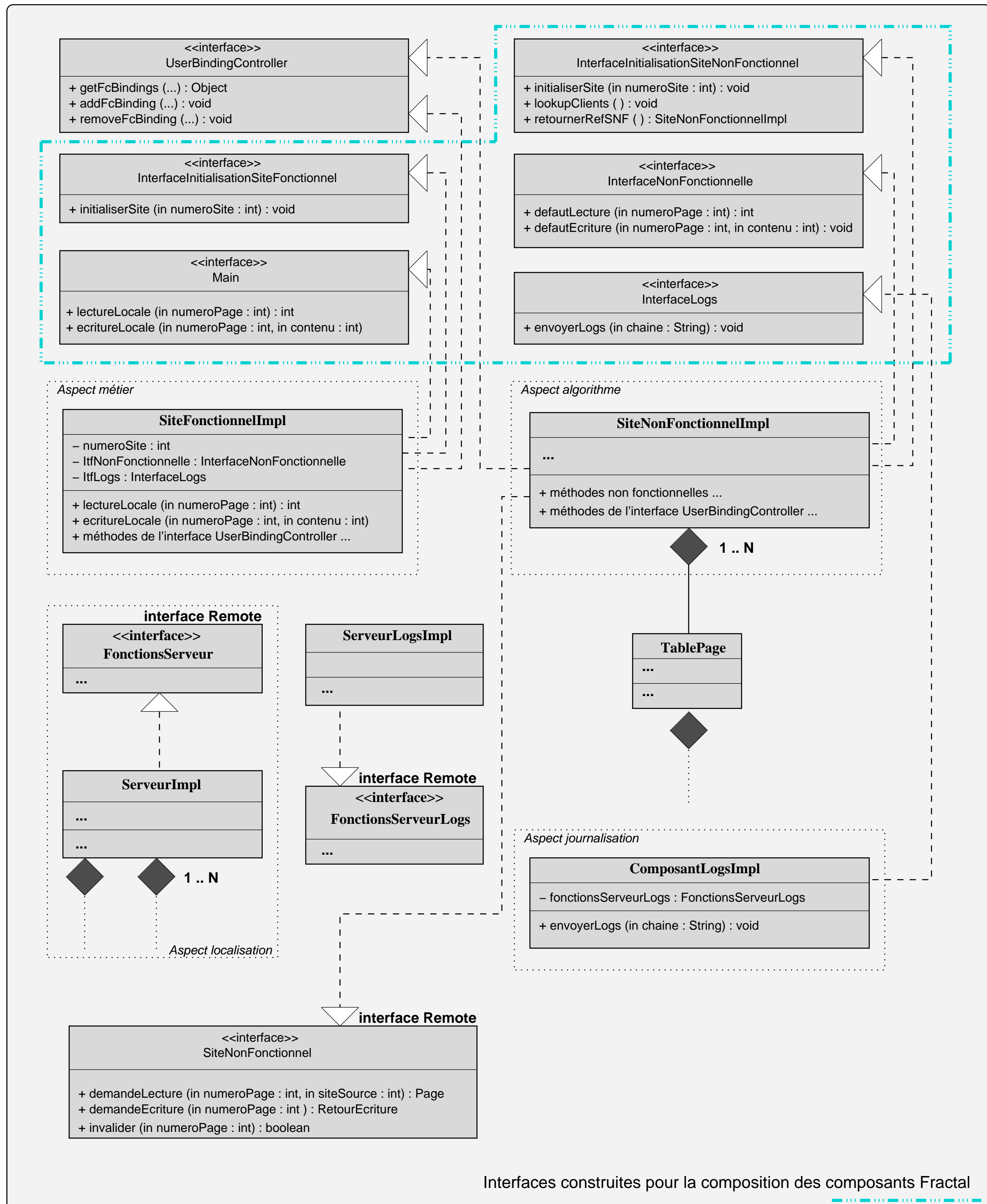

FIG. 19 – Modélisation de l'implémentation de l'application avec FRACTAL

composant qui sera associé à cette classe sera également assemblé.
Les composants FRACTAL sont assemblés par l'intermédiaire d'interfaces clientes et serveurs. Pour cela, nous avons modélisé cinq interfaces (notés dans le modèle `Interface*`). Pour comprendre quelles classes les implémentent, il faut se reporter au modèle de composition de notre application. On peut classer ces cinq interfaces en deux catégories : les trois interfaces `Main`, `InterfaceNonFonctionnelle` et `InterfaceLogs` qui sont nécessaires à la composition structurelle de notre "super" composant client, et les deux autres (`InterfaceInitialisation*`) nécessaires au processus d'initialisation de l'application.

10.4.2 Le modèle de composition

Nous rappelons que dans notre vision de l'assemblage des fonctionnalités (figure 17 page 80) nous ne nous intéressons qu'au client et nous ne nous préoccupons pas de la répartition (pas d'assemblage réparti). En effet, selon notre vision de l'application, l'aspect de localisation n'est assuré que par le serveur de façon centralisée, il est donc implémenté par un objet JAVA et n'est donc pas sujet à une composition structurelle FRACTAL.
La schématisation de notre composition est donc la suivante :

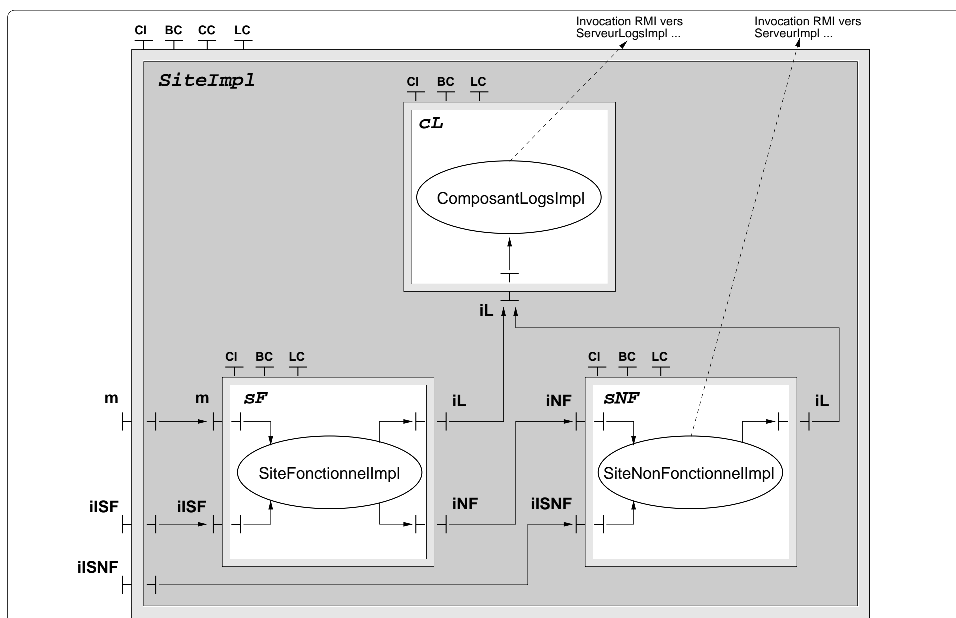

FIG. 20 – Modélisation de la composition avec FRACTAL

Le "super" composant composite `SiteImpl` est le composant racine du client. Il est composé de trois composants primitifs : `sF`, `sNF` et `cL`. Chacun

de ces composants est implémenté par leurs classes respectives tel que nous l'avons décrit dans la modélisation précédente.

Le composant primitif `cL` fournit l'interface :
- `iL` de type `InterfaceLogs`

Le composant primitif `sNF`
- fournit les interfaces :
  - `iNF` de type `InterfaceNonFonctionnelle`
  - `iISNF` de type `InterfaceInitialisationSiteNonFonctionnel`
- requiert l'interface :
  - `iL` de type `InterfaceLogs`

Le composant primitif `sF`
- fournit les interfaces :
  - `m` de type `Main`
  - `iISF` de type `InterfaceInitialisationSiteFonctionnel`
- requiert les interfaces :
  - `iL` de type `InterfaceLogs`
  - `iNF` de type `InterfaceNonFonctionnelle`

Le composant composite `SiteImpl` fournit les interfaces :
- `m`
- `iISF`
- `iISNF`

Nous ne revenons pas sur les interfaces `ComponentIdentity`, `BindingController`, `ContentController` et `LifeCycleController` partie intégrante du modèle FRACTAL et déjà présentées (cf. 5.3.1 page 35).

10.4.3 Remarques sur l'implémentation

Nous avons préservé les implémentations des interfaces `Remote` de notre implémentation en JAVA (ainsi que les implémentations de `ServeurImpl` et `ServeurLogsImpl`). Par contre, l'implémentation du client a été séparée en trois classes distinctes comme nous l'avons mentionné dans le modèle.
De manière à coller au modèle FRACTAL, nous avons dû implémenter les interfaces de type `Interface*`, ainsi que les méthodes implémentant l'interface `UserBindingController`. Dans le code de chaque classe côté client sont présentes les interfaces matérialisant l'assemblage des composants. Le code relatif à la journalisation des informations côté client est dispersé dans les classes `SiteFonctionnelImpl` et `SiteNonFonctionnelImpl` selon les accès que l'on veut journaliser.

10.4.4 Remarques sur l'assemblage

Nous avons utilisé l'ADL de FRACTAL pour définir la composition structurelle de notre composant `SiteImpl` et non les méthodes de l'API (comme dans l'exemple exposé plus haut, cf. 3 page 40). Nous décrivons donc nos assemblages dans des fichiers XML (extensions `.fractal`) : trois fichiers associés aux composants primitifs et un fichier associé au composant composite.
Donnons par exemple le fichier XML associé à la définition du composant principal `SiteImpl` :

```
 1 <component-type name="RootType">
 2   <provides>
 3     <interface-type name="m" signature="Main"/>
 4     <interface-type name="iISF" signature="InterfaceInitialisationSiteFonctionnel"/>
 5     <interface-type name="iISNF" signature="InterfaceInitialisationSiteNonFonctionnel"/>
 6   </provides>
 7 </component-type>
 8
 9 <composite-template name="Site" implements="RootType">
10   <composite-content>
11     <components>
12       <component name="sF" type="SiteFonctionnelType"/>
13       <component name="sNF" type="SiteNonFonctionnelType"/>
14       <component name="cL" type="ComposantLogsType"/>
15     </components>
16     <bindings>
17       <binding client="this.m" server="sF.m"/>
18       <binding client="sF.iNF" server="sNF.iNF"/>
19       <binding client="sF.iL" server="cL.iL"/>
20       <binding client="sNF.iL" server="cL.iL"/>
21       <binding client="this.iISF" server="sF.iISF"/>
22       <binding client="this.iISNF" server="sNF.iISNF"/>
23     </bindings>
24   </composite-content>
25 </composite-template>
26
27 <composite-template name="SiteImpl" extends="Site">
28   <composite-content>
29     <components>
30       <component name="sF" implementation="SiteFonctionnelImpl"/>
31       <component name="sNF" implementation="SiteNonFonctionnelImpl"/>
32       <component name="cL" implementation="ComposantLogsImpl"/>
33     </components>
34   </composite-content>
35 </composite-template>
```

Les lignes 1 à 7 nous permettent de définir un type de composant nommé `RootType` avec les trois type d'interfaces qu'il fournit. Des lignes 9 à 25, on définit le contenu du composant composite (la liste des types de composants primitifs qui le constitue) ainsi que la description des liaisons entre ces composants (par exemple, le ligne 17 permet d'exporter l'interface du composant métier à l'extérieur du composant composite). Enfin, des lignes 27 à 35, on explicite les implémentations des composants primitifs (l'attribut `implementation` ne correspond pas à la classe JAVA associée au composant mais à une référence

vers la description de ce composant défini dans un autre fichier).

10.4.5 Exécution et déploiement

Le déploiement de l'application par la fonction `main` est identique à celui exposé dans l'implémentation en JAVA pur (cf. 10.3.3 page 83), mis à part le fait qu'au lieu d'instancier trois objets de la classe `Site` précédente, nous instancions trois composants FRACTAL : on récupère trois *templates* associés au composant composite `SiteImpl`, on les instancie (JULIA se charge alors automatiquement d'effectuer les instanciations des objets et les liaisons définies dans l'ADL), et on lance le composant de manière à le rendre actif dans le système. On peut alors récupérer les interfaces d'initialisation pour invoquer nos méthodes d'initialisation de notre application. On démarre ensuite notre simulation. Voici le code permettant d'initialiser une instance d'un site :

```
1  public static void main (final String[] args) {
2
3       ...
4       Parser parser = Launcher.getBootstrapParser();
5       // On load le template
6       rTmpl0 = parser.loadTemplate("SiteImpl");
7       // Instanciation du template
8       ComponentIdentity rComp0 = Fractal.getTemplate(rTmpl0).instantiateFc();
9       // Lancement du composant racine
10      Fractal.getLifeCycleController(rComp0).startFc();
11      // Invocation de la méthode d'initialisation du site non-fonctionnel
12      ((InterfaceInitialisationSiteNonFonctionnel)rComp0.getFcInterface("iISNF")).initialiserSite (0);
13      // Invocation de la méthode d'initialisation du site fonctionnel
14      ((InterfaceInitialisationSiteFonctionnel)rComp0.getFcInterface("iISF")).initialiserSite (0);
15      // Binding RMI du composant associé au site non-fonctionnel
16      Naming.rebind ("rmi://localhost:40000/acpoa/Client0",
17           ((InterfaceInitialisationSiteNonFonctionnel)rComp0.getFcInterface("iISNF")).retournerRefSNF ());
18      ((InterfaceInitialisationSiteNonFonctionnel)rComp0.getFcInterface("iISNF")).lookupClients ();
19      // Début de la simulation ...
20      ((Main)rComp0.getFcInterface("m")).lectureLocale (0);
21
22      ...
23 }
```

10.5 Version implémentée avec KILIM

10.5.1 Le modèle d'implémentation

Le modèle d'implémentation utilisé pour la plate-forme KILIM ne change absolument pas en ce qui concerne le serveur annuaire et le serveur de logs par

rapport à ce que l'on a exposé précédemment. Du côté du client, la séparation des classes (selon nos préoccupations) est la même que celle exposée dans le modèle d'implémentation de FRACTAL (cf. 10.4.1 page 83). Bien entendu, notre modèle avec KILIM ne prend pas en compte les interfaces que nous avions eu besoin de modéliser avec le modèle FRACTAL.
Comme nous l'avons expliqué dans la présentation de KILIM, nous devons cependant modéliser les méthodes *setter* nécessaires à l'assemblage des composants. Notre classe `SiteFonctionnelImpl` dispose donc des méthodes `setSNF` et `setCL` (respectivement pour ajouter la référence d'un `SiteNonFonctionnelImpl` et d'un `ComposantLogsImpl`) et la classe `SiteNonFonctionnelImpl` d'une méthode `setCL`.

10.5.2 Le modèle de composition

L'assemblage de nos composants KILIM peut se schématiser de la manière suivante :

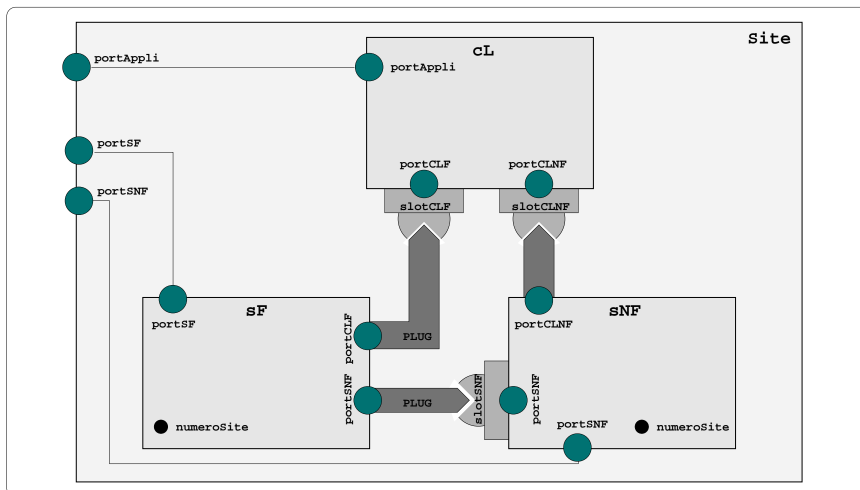

FIG. 21 – Le modèle de composition de notre prototype avec KILIM

Nous avons volontairement présenté une approche "boîte noire" de l'assemblage des composants (c'est-à-dire que nous n'avons pas schématisé les mécanismes internes de types providers et transformers). Nous avons en effet déjà expliqué l'utilisation de ces abstractions du modèle KILIM (cf. 7 page 48).
Le client de notre prototype est donc formé par le composant `Site` dont le contenu est formé par les composants `cL`, `sF` et `sNF`. Ce modèle de composition est complètement similaire à celui présenté dans le cadre du modèle FRACTAL.
Le composant `Site` dispose du port `portAppli` car c'est par son intermédiaire que sont instanciés de manière récurssive tous les composants de l'application.

Nous "récupérons" ensuite une référence du composant `sF` (pour y invoquer les méthodes métier) et du composant `sNF` pour terminer l'initialisation du site (tous les sites de l'application doivent effectivement être instanciés avant d'effectuer les *lookups* RMI).

10.5.3 Remarques sur l'assemblage

Il y a des dépendances dans le processus d'instanciation de l'application. En effet, étant donné que les composants `sF` et `sNF` ont besoin d'une référence vers le composant `cL`, ce-dernier doit être instancié en premier. On évoque ici le graphe de dépendance entre les exécutions des abstractions de KILIM décrit dans la présentation de la plate-forme (cf. 6.3.1 page 45). La description de l'assemblage des composants, et plus précisément la notion de dépendances entre providers et transformers influence donc notre vision de l'application KILIM. C'est donc à partir de la référence `portAppli` que s'instancient récursivement les objets associés aux composants. Un *bind* sur `portAppli` déclenche donc un provider sur le composant `cL` nous permettant d'obtenir une instance de la classe `ComposantLogsImpl`. A partir de cette liaison, une série de transformers est déclenchée. Un événement est généré de telle sorte qu'il provoque l'exécution d'un provider (qui est en fait un constructeur prenant la property `numeroSite` comme paramètre) sur le composant `sNF` et d'y invoquer la méthode `setCL` de l'instance de `SiteNonFonctionnelImpl` nouvellement créée. Cette méthode prend en argument l'instance encapsulée dans le composant `cL`. On procéde de la même manière avec un événement qui déclenche l'instanciation de la classe `SiteFonctionnelImpl` du composant `sF`. Le troisième transformer est responsable de l'établissement de la liaison entre le composant `sF` et le composant `sNF`. Cet événement est en fait généré par le composant `cL` puisqu'il détient les références des deux composants concernés.
De manière à bénéficier d'une vision plus pragmatique, donnons une partie de l'ADL définie dans les composants `cL` et `sNF` (figure 22 page 91).

10.5.4 Exécution et déploiement

Dans la fonction `main`, on récupère les trois *templates* associés aux trois sites clients. Ils ne différent que par leur propriété `numeroSite`. Grâce aux méthodes de l'API (expliquées dans la présentation de KILIM, cf. 5 page 49) nous effectuons un *binding* explicite sur le `portAppli` qui provoque les instanciations nécessaires au composant `Site`. Nous récupérons ensuite une référence de l'instance `SiteNonFonctionnelImpl` de manière à l'ajouter au *registry* RMI, et une instance de la classe `SiteFonctionnelImpl` pour y invoquer les méthodes métier.

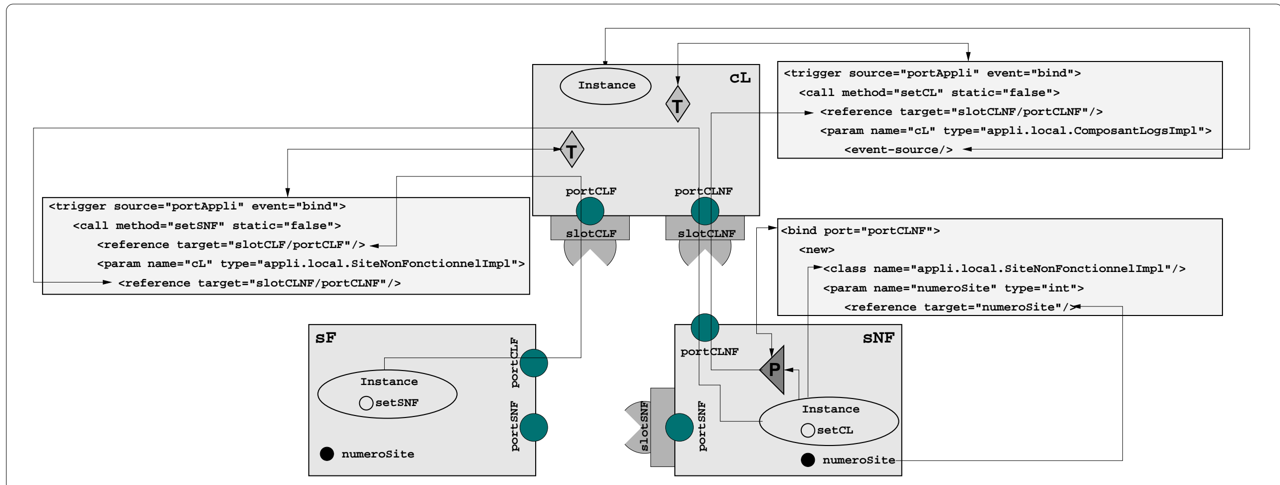

FIG. 22 – Schématisation de certains mécanismes mis en jeu dans l'instanciation de l'application au niveau du modèle KILIM

Les deux serveurs sont également instanciés de la même façon que celle déjà décrite dans l'implémentation JAVA et FRACTAL.

10.6 Version implémentée avec JAC

10.6.1 Le modèle d'implémentation

Dans le cadre de l'implémentation de notre prototype avec JAC, nous avons suivi les mêmes directives que celles que nous avons proposées en terme de séparation des préoccupations. De manière à transposer cette séparation dans la plate-forme JAC, nous sommes partis de l'idée que notre seul objet de base (les objets de base sont ceux qui correspondent à l'implémentation des fonctionnalités métier dans le paradigme de programmation par aspects) était bien sûr une instance de la classe `SiteFonctionnelImpl` puisqu'elle modélise les opérations métier de notre application. Cependant, contrairement aux modèles précédents, nous avons choisi d'agréger la classe `TablePages` à notre classe métier, de telle sorte que l'opération de lecture métier se charge elle-même d'accéder au contenu de la page de la structure de données `TablePages`. Bien entendu, une référence vers cette même instance de notre structure de données sera alors nécessaire dans les composants d'aspects, comme nous le verrons

plus loin.

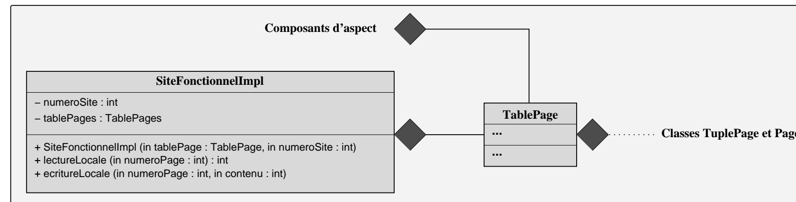

FIG. 23 – Une petite modification du modèle d'implémentation de notre prototype avec la plate-forme JAC

10.6.2 Le modèle de composition

Le modèle de composition que nous avons établi pour la plate-forme JAC peut être schématisé de la manière suivante (nous avons utilisé les notations UML définies dans [PDF+02a]) :

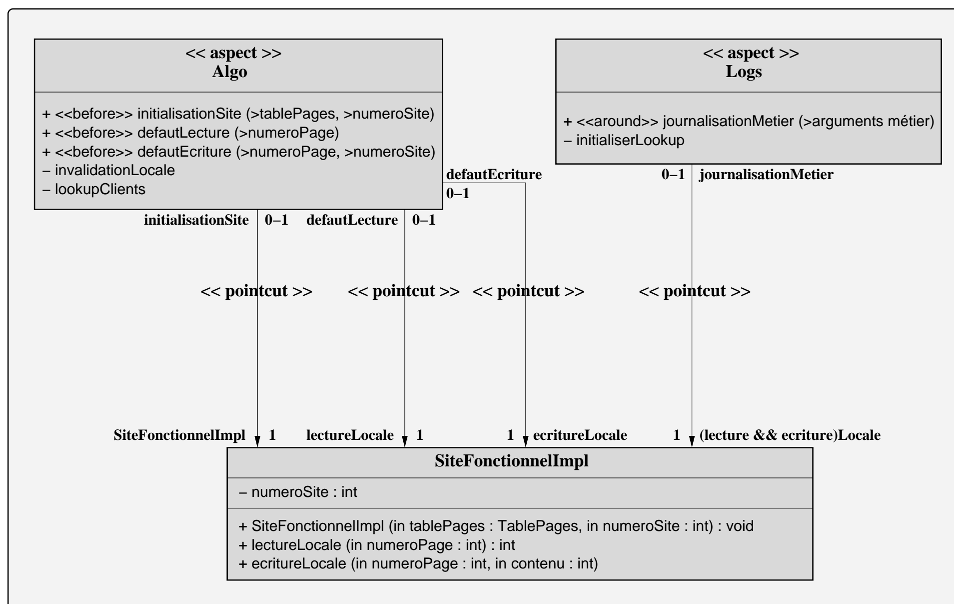

FIG. 24 – Le modèle de composition des aspects avec JAC

Dans l'objet de base de notre application, modélisé par la classe métier `SiteFonctionnelImpl`, nous ne trouvons que les deux opérations métier ainsi que le constructeur de cette classe prenant en argument la structure de données

`TablePages` (pour les raisons que nous avons évoquées plus haut). Comme le montre notre modèle de composition, l'aspect que nous avons nommé `Algo` est tissé autour de trois points de jointure de notre classe métier :

- La méthode `initialisationSite` sera exécutée avant le constructeur de la classe métier. Cette méthode initialise le composant d'aspect associé à la classe métier en récupérant les arguments `tablePages` et `numeroSite` du constructeur métier. Elle effectue les *lookups* RMI nécessaires pour récupérer les références distantes sur serveur de localisation et du serveur de journalisation. Elle créée également une instance de la classe `SiteNonFonctionnelImpl` (toujours avec les arguments `tablePages` et `numeroSite`) puis effectue le *bind* auprès du *registry* RMI pour cette instance.

- La méthode `defautLecture` sera exécutée avant la méthode métier `lectureLocale`. Elle correspond à l'implémentation des mécanismes nécessaires pour effectuer un défaut en lecture.

- La méthode `defautEcriture` fait de même mais pour effectuer un défaut en écriture avant la méthode métier `lectureLocale`.

La méthode `lookupClients` de l'aspect `Algo` permet d'effectuer les *lookups* RMI de toutes les instances de `SiteNonFonctionnelImpl` composant notre système.

L'aspect `Logs` est tissé autour des deux opérations métier. La méthode `journalisationMetier` permet donc de contacter le serveur de journalisation afin de lui envoyer les informations que l'on a décidé de journaliser.

Il faut remarquer que dans ce modèle de composition JAC, nous n'avons pas prévu de journalisation des accès aux méthodes non-fonctionnelles. Pour cela, il aurait fallu tisser l'aspect de journalisation autour des méthodes du composant d'aspect `Algo`. Cependant, cela ne pose pas de problème particulier puisque dans JAC les composants d'aspect sont des objets JAVA, il est donc possible de les aspectiser au même titre que des classes considérées comme métier.

10.6.3 Remarques sur l'implémentation

JAC est un framework de programmation orientée aspects : de nombreux aspects du framework sont donc utilisables et configurables pour les adapter

à une application donnée. Cependant, dans le cadre de cette implémentation, nous étions obligés d'implémenter nos propres aspects, du moins en ce qui concerne l'aspect `Algo` puisque celui-ci représente un besoin tout à fait particulier à notre prototype. Comme dans l'exemple exposé dans la présentation de JAC, nous avons programmé nos aspects de telle sorte qu'ils puissent être configurables (c'est-à-dire que le tissage des aspects est décrit indépendamment de l'implémentation et n'est donc pas codé en dur au sein de l'application).

L'implémentation de la classe métier est très succinte : le corps de la méthode `lectureLocale` se contente d'accéder à la page dont le numéro est donné en paramètre et y extrait le contenu. Il en est de même pour la méthode `ecritureLocale` qui écrit le nouveau contenu de la page dans la structure de données. C'est d'ailleurs pour cette raison que nous avons décidé d'exporter une référence de `TablePages` dans la classe métier. Si cela n'avait pas était le cas, le corps des méthodes métier serait tout simplement vide, puisque tout le traitement serait alors assuré par les méthodes wrappantes. Cette vision de l'application nous permet de mieux appréhender le paradigme de programmation par aspect. En effet, l'opération de lecture métier effectue réellement une lecture, tout comme l'opération d'écriture métier qui effectue une écriture dans la structure de données où sont stockées les pages. Cependant, la mis en oeuvre de l'AOP permet dans ce cas d'assurer que la cohérence des données est bien respectée mais de manière totalement transparente à notre classe métier.

Nous avons implémenté les composants d'aspects `AlgoAC` et `LogsAC`. Ils sont composés de méthodes qui nous permettent de définir les points de jointure tels que nous les avons spécifiés dans le modèle de composition de notre application. L'implémentation de ces méthodes nous permet d'utiliser le niveau configuration de JAC par la suite. Il nous suffira alors de définir dans des fichiers textes les méthodes des composants d'aspect que nous déciderons d'utiliser avec les arguments nécessaires (noms de méthodes, classes, instances ...etc) de manière à tisser au moment de l'exécution nos aspects sur notre objet de base.

Nous avons implémenté nos deux classes *wrapper* : `AlgoWrapper` et `LogsWrapper`. Dans ces classes sont implémentées les méthodes wrappantes de nos composants d'aspect (nous les avons décrites plus haut, cf. 10.6.2 page 92). Dans le cadre de notre prototype, nos aspects sont "avec état" dans le sens où ils sont composés d'un ensemble de champs déterminants tels le numéro du site, une référence vers une instance de la classe `SiteNonFonctionnelImpl`, une référence vers la structure de données `TablePages` ...etc. Ces champs doivent être persistants puisqu'ils sont nécessaires à la mise en oeuvre des mécanismes liés au bon déroulement de notre algorithme de gestion de la cohérence. L'aspect `Logs` est également "avec état" mais son état est partageable avec les différents

sites puisque son seul champ est statique : il s'agit de la référence distante vers le serveur de journalisation. Il est d'ailleurs possible au sein de la plate-forme JAC de partager un aspect, même sur des sites répartis. Cependant, dans ce premier contexte, nous n'avons pas utilisé l'aspect de déploiement de JAC. A chaque site "métier" est donc attribuée une instance de composant d'aspect `Algo` et une instance de `Logs`.
Il faut noter qu'au sein de JAC, il est possible par l'intermédiaire d'une méthode wrappante d'accéder à l'instance de l'objet wrappé : il est alors possible d'accéder aux attributs (dont la visibilité est rendue *public*) de cette instance et éventuellement d'en changer leurs états. Cette capacité d'introspection peut être très intéressante, comme par exemple pour notre aspect `Logs` pour lequel on pourrait décider de journaliser des attributs de l'instance wrappée (qui ne sont pas passés en argument de la méthode wrappée).

Le reste de l'implémentation de la logique non-fonctionnelle de notre prototype est identique à celle que nous avons déjà exposée dans le cadre des autres plates-formes.

10.6.4 Remarques sur l'assemblage

Etant donné que nous avons implémenté nos propres aspects, il est nécessaire de les déclarer dans le fichier `jac.prop`.
Puis il suffit de décrire la configuration de ces aspects dans les fichiers `algo.acc` et `logs.acc`. Dans le fichier `appli.jac`, nous définissons la classe de notre prototype dans laquelle est implémentée la fonction `main` ainsi que les aspects que nous décidons de tisser au moment de l'exécution.

10.6.5 Exécution et déploiement

Dans la fonction `main` nous nous contentons d'instancier le serveur annuaire et le serveur de journalisation puis de les *binder* auprès du *registry* RMI. Il suffit alors d'instancier les trois sites de la classe `SiteFonctionnelImpl` puis de commencer la simulation en invoquant des lectures et des écritures sur ces instances.
Au moment de l'exécution, le tissage dynamique des aspects se charge du reste.

11 Présentation de l'algorithme décentralisé

Donnons une description détaillée du fonctionnement de l'algorithme de KAI LI et PAUL HUDAK en version décentralisée et *copyset* distribué utilisant un mécanisme d'invalidation synchrone.

Dans ce cas, il n'y a pas de serveur de localisation, on dispose donc d'une heuristique qui consiste à interroger une liste des propriétaires probables de la copie de référence.

11.1 Les structures de données

Comme dans le cas précédent, chaque site dispose d'une structure de données que l'on nomme `TableDesPages` composée d'un ensemble de tuples permettant de stocker les informations relatives à toutes les pages mémoires actives dans le système. Les champs de ces tuples sont les suivants :

- **Numéro** : Le numéro de la page.
- **Verrou** : Pour gérer les accès à une page en exclusion mutuelle.
- **Copyset** : Liste des sites ayant une copie de la page concernée.
- **PropProb** : Le numéro du site qui est propriétaire probable de la page.
- **Droits** : Les droits d'accès à une page, `l`, `e` ou `i`.
- **Propriétaire** : Indique si le site est propriétaire ou non de la page concernée.

Par rapport à la version centralisée de l'algorithme proposé (cf. 9.1 page 68), cette structure est différente concernant les champs `Verrou` et `PropProb`. En effet, nous avons décidé dans cette deuxième implémentation de gérer l'exécution des traitements relatifs à une page en exclusion mutuelle (champ `Verrou`). Nous avons déjà expliqué que dans le cadre de notre simulation, la séquentialisation des accès ne pouvait corrompre la cohérence des données, mais nous avons tout de même décidé d'implémenter ce mécanisme de manière à dégager une nouvelle fonctionnalité de notre application comme nous le verrons plus loin.
Le champ `PropProb` est nécessaire pour la mise en place de notre heuristique de localisation. Cette notion de propriétaire probable met en avant le fait que le site n'a qu'une estimation du véritable propriétaire de la page.

La structure `TableDesPages` est la seule que nous avons à gérer puisque cette version de l'algorithme n'a pas besoin de serveur de localisation centralisé pour fonctionner. Le service de localisation est alors interne au client.

11.2 Déroulement du protocole

Dans cette version de l'algorithme, le protocole ne spécifie que deux types de fonctionnalités en lecture et en écriture :

- Le site effectuant un défaut (en lecture ou en écriture)
- Le site répondant à une demande générée par un demandeur

Bien entendu, dans le processus de défaut, il se peut que différents sites intermédiaires soient mis en cause. En effet, notre heuristique de localisation n'assure pas que le défaut généré sur un site soit "servi" en un saut, comme nous l'expliquons ci-dessous.

11.2.1 L'algorithmie des différentes fonctions

Les différentes fonctions du protocole sont décrites par les étapes suivantes. Dans ce cas, nous avons utilisé pour décrire l'algorithme une syntaxe d'invocation de méthodes de type RPC (Recevoir ...Retourner) pour simplifier sa compréhension, et de manière à être proche de l'implémentation que nous avons effectuée avec JAVA RMI :

Défaut en lecture sur un site lecteur :

```
1.  page.Verrou = occupe
2.  Envoyer a page.PropProb : demandeLecture [NumLecteur, page.Numero]
3.  Recevoir de page.PropProb : [contenu]
4.  page.Contenu ← contenu
5.  page.Droits ← lecture
6.  page.Verrou = libre
```

Réception d'une demande en lecture sur le propriétaire probable :

```
7.  Recevoir du lecteur : [NumLecteur, NumPage]
8.  page.Verrou ← occupe
9.  Si page.Droits == invalide
10.     Envoyer a page.PropProb : demandeLecture [NumLecteur, page.Numero]
11.     Recevoir de page.PropProb : [contenu]
12.     Retourner au lecteur : [contenu]
13. Sinon
14.     page.Copyset ← page.Copyset + lecteur
15.     Retourner au lecteur [page.Contenu]
16. page.Verrou ← libre
```

Défaut en écriture sur un site écrivain :

```
17.  page.Verrou ← occupe
18.  Envoyer a page.PropProb : demandeEcriture [NumEcrivain, page.Numero]
19.  Recevoir de page.PropProb le chemin suivi par la requete
20.  Pour tous les sites j de page.Copyset et du chemin suivi par la requete :
21.      Envoyer a j : demandeInvalidation [NumPage]
22.      Recevoir de j un acquitement
23.  page.Contenu ← nouveau contenu
24.  page.Droits ← ecriture
25.  page.Proprietaire ← vrai
26.  page.Verrou ← libre
```

Réception d'une demande en écriture sur le propriétaire probable :

```
27.  Recevoir du demandeur : [NumEcrivain, NumPage]
28.  page.Verrou ← occupe
29.  Si page.Proprietaire == vrai :
30.      page.Proprietaire ← faux
31.      page.PropProb ← NumEcrivain
32.      Retourner au demandeur le numero de ce site
33.  Sinon :
34.      Envoyer a page.PropProb : demandeEcriture [NumEcrivain, NumPage]
35.      Recevoir de page.PropProb un acquitement
36.      Retourner au demandeur le chemin suivi par la requete + le numero de ce site
37.      page.PropProb ← NumEcrivain
38.  page.Verrou ← libre
```

Réception d'une demande d'invalidation :

```
39.  Recevoir du demandeur : [NumPage]
40.  page.Verrou ← occupe
41.  page.Droits ← invalide
42.  page.Contenu ← vide
43.  Pour tous les sites j de page.Copyset :
44.      Envoyer a j : demandeInvalidation [NumPage]
45.      Recevoir de j un acquitement
46.  Retourner au demandeur un acquitement
47.  page.Verrou ← libre
```

Dans le cas d'un défaut en lecture, n'importe quel site qui détient un exemplaire de la page peut fournir cette page au demandeur (ligne 15), on ne se préoccupe pas de savoir si ce site est le vrai proriétaire ou non. C'est d'ailleurs en ce sens que le *copyset* est considéré comme distribué puisque l'on ajoute le demandeur au *copyset* d'un site qui peut ne pas être le propriétaire (ligne 14).

Dans le cas d'un défaut en écriture, la requête initiée de l'écrivain est propagée de propriétaires probables en propriétaires probables (lignes 33 à 37) jusqu'à atteindre le vrai propriétaire (ligne 29). Lorsque l'on déroule récursivement ces appels à demandeEcriture, chaque site ajoute à la structure de donnée de retour son numéro de site (lignes 32 et 36). Ainsi, l'écrivain sera en mesure d'invalider le contenu de la page pour ces sites (ligne 20). Le processus d'invalidation se fait alors pour tous les sites qui disposaient d'un accès à cette page (schématisé plus loin).

Il faut remarquer que par souci de simplifier le protocole, nous ne gérons pas le fait qu'un défaut n'aboutisse pas selon notre heuristique de localisation. Cela pourrait arriver dans le cas d'un défaut en lecture si le chemin des propriétaires probables forment un cycle de site dont les accès à la page concernée sont invalides, ou dans le cas d'un défaut en écriture si le chemin n'aboutit pas au vrai propriétaire de la page. Pour pallier à ces problèmes, il suffirait de fournir aux invocations distantes une structure de données contenant le chemin déjà parcouru afin de détecter les cycles et de retourner une erreur avec le chemin déjà parcouru. Le site initiateur du défaut pourrait alors diffuser sa demande sur les sites non parcourus.

On peut considérer que la cohérence associée à ce protocole est forte (de l'ordre d'une cohérence séquentielle). En effet, l'écriture proprement dite du contenu d'une page ne se fait qu'après le processus d'invalidation (ligne 23). Cependant, on peut considérer une ambiguïté dans le sens où des accès en lecture peuvent se faire sur d'autres sites (sur la page concernée) après la demande de l'écrivain et avant la réception de la requête d'invalidation, puisque les verrous associés à cette page sont relâchés entre ces deux étapes.

11.2.2 Schématisation du déroulement du protocole

Les différentes étapes associées au déroulement d'un défaut en lecture/écriture et du mécanisme d'invalidation peuvent être schématisées de la manière suivante :

Le site 6 effectue un défaut en lecture, il est servi par le site 2. Le graphe des propriétaires probables ne change pas entre les schémas (a) et (b). Le site 2 ajoute le site 6 à son *copyset* (figure 25 page 100).

Le site 6 effectue un défaut en écriture. La requête passe par le site 2, est propagée sur le site 1. Le site 6 est alors servi par le site 1. Le graphe des propriétaires probables est mis à jour (figure 26 page 100).

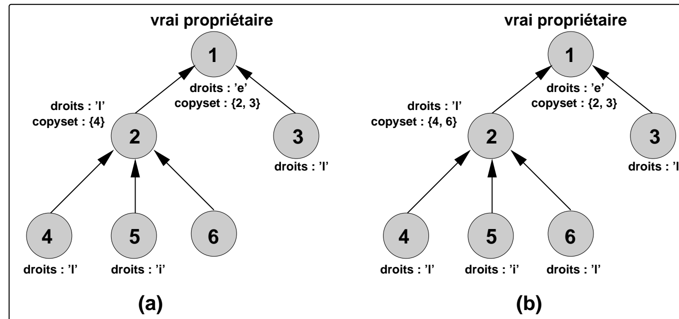

FIG. 25 – Graphe des propriétaires probables dans le cas d'un défaut en lecture initié par le site 6 dans le cadre de l'algorithme décentralisé

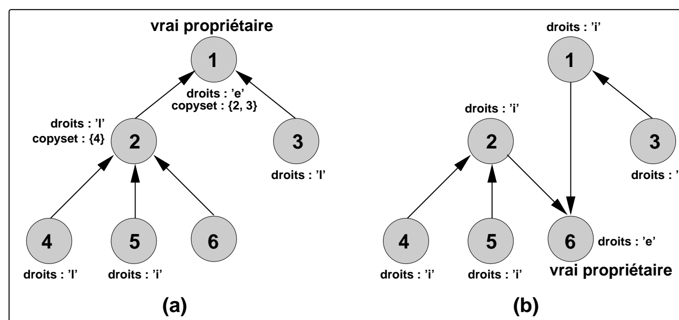

FIG. 26 – Graphe des propriétaires probables dans le cas d'un défaut en écriture initié par le site 6 dans le cadre de l'algorithme décentralisé

Entre les étapes (a) et (b) de la figure précédente, le site 6 exécute le mécanisme d'invalidation, en envoyant une requête sur le site 2 et sur le site 1. Ils diffusent à leur tour la demande en fonction de leurs *copysets* :

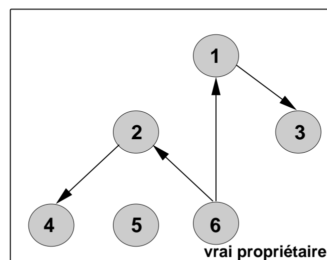

FIG. 27 – Propagation des requêtes de demande d'invalidation initiée par le site 2 dans le cadre le l'algorithme décentralisé

Dans le cas de ce protocole, la distribution du *copyset* organise l'ensemble des noeuds en arbre, le graphe des propriétaires probables va des feuilles vers la racine, celui des invalidations va de la racine vers les feuilles. En effet, c'est un résultat classique de cette version de l'algorithme de KAI LI et PAUL HUDAK.

12 Deuxième implémentation

Cette partie traite de l'implémentation de l'algorithme de KAI LI et PAUL HUDAK en version décentralisée avec heuristique de localisation (notion de propriétaire probable) et gestion décentralisée des *copysets*.

12.1 Contexte et hypothèses

Le contexte général (en ce qui concerne l'algorithme) est le même que celui explicité dans le cadre de l'algorithme centralisé (cf. 10.1.1 page 74) en ce qui concerne les communications fiables et le déploiement statique de notre système.
Dans ce deuxième cas, seules les machines clientes interviennent puisque le service de localisation est interne à celles-ci.

Les choix et simplifications au niveau de l'implémentation sont les mêmes que dans le cas centralisé (cf. 10.1.2 page 75) en ce qui concerne la granularité de l'entité sur laquelle porte la cohérence (c'est-à-dire la `Page`), le processus d'initialisation, l'utilisation de JAVA RMI pour les communications distantes...

Contrairement au cas précédent, nous considérons que la journalisation se fait localement à chaque site et non par l'intermédiaire d'un site distant.
Notre système sera composé de trois entités (sites). Cependant par souci de simplification, nous ne considérons qu'une page mémoire active au sein du système (même si, comme nous le verrons, nous avons proposé une modélisation dont les méthodes définies permettent la gestion de plusieurs pages).

12.2 La séparation des préoccupations

La programmation d'une autre version de l'algorithme de gestion de la cohérence est bien sûr un objectif de cette deuxième implémentation. Cependant, nous avons décidé de changer également notre vision des différentes fonctionnalités, de séparer différemment les préoccupations de notre application.

Comme nous l'avons déjà expliqué (cf. 10.2 page 79), nous ne prenons pas en compte l'aspect concernant l'entité sur laquelle porte la cohérence, ni le nommage.

Notre aspect métier demeure inchangé. Il est constitué des deux opérations fonctionnelles `lire` et `ecrire`.

En ce qui concerne les aspects non-fonctionnels, les différentes fonctionnalités que nous avons décidé d'isoler sont les suivantes :

- La gestion de la **cohérence**. Nous considérons que la cohérence des données est assurée par la mise en oeuvre du mécanisme d'invalidation et par la gestion du verrouillage des accès aux données (exclusions mutuelles). Dans le cadre de ce deuxième exemple, nous avons décidé de diminuer la granularité des fonctionnalités : L'aspect cohérence sera donc lui-même composé de deux aspects : l'**invalidation** et le **verrouillage**.
- La **localisation**. Le service de localisation est maintenant interne au client, il en est partie intégrante. Nous considérons la fonction de localisation comme un aspect à part entière.
- La gestion de l'**algorithme**. Dans cet aspect, nous considérons tous les traitements liés à la mise en oeuvre de l'algorithme décentralisé en dehors des aspects localisation et cohérence.
- La **journalisation** : comme dans notre première proposition de séparation, nous la considérons comme un aspect à part entière.
- Les **communications distantes**. Nous avons décidé de les isoler en partant du principe que la gestion des invocations de méthodes RMI n'avait pas à être mélangée avec la gestion des traitements liés à l'algorithme.

Notre nouvelle vision de la séparation des préoccupations peut donc être schématisée par la figure 28.

Comme nous le montre cette figure, nous avons isolé sept fonctionnalités (ou aspects) : un aspect fonctionnel et six aspects non-fonctionnels. Sur cette représentation, deux fonctionnalités partageant une même frontière sont en interaction l'une de l'autre (cette interaction est illustrée par une flêche double). En ce sens, nous représentons les dépendances qui existent entre les différents aspects, leur manière de s'interfacer. Cependant, cette représentation n'illustre pas l'intégralité de la sémantique de l'application. Par exemple, la fonctionnalité `Journalisation` utilise un service à la fonctionnalité de `Communications distantes` mais ce n'est pas réciproque. La fonctionnalité `Moulinette Algo` utilise l'aspect `Invalidation` seulement dans des cas bien précis, par exemple, réception d'une requête de demande de lecture sur un propriétaire probable ou exécution du mécanisme d'invalidation lors d'un défaut en écriture ...Il nous manque donc des informations relatives au flux d'exécution d'une opération donnée, inhérentes à la modélisation de l'application (que nous expliciterons

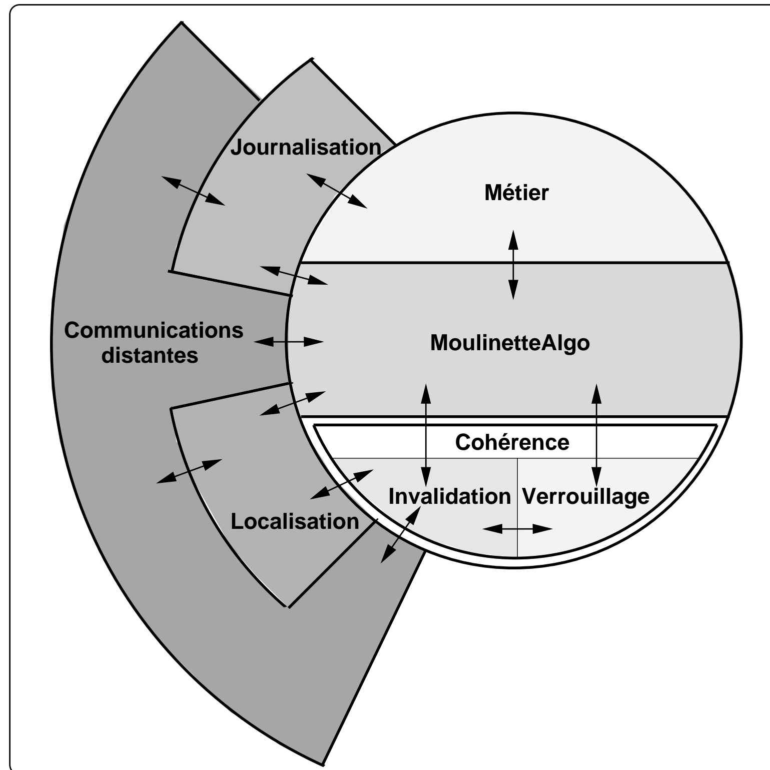

FIG. 28 – Schématisation de la séparation des préoccupations au sein de l'algorithme décentralisé

plus tard).

Tous ces aspects se partagent des "interfaces", mais également la structure de données `TableDesPages` (cf. 11.1 page 96) dans le sens où chaque fonctionnalité accède aux champs relatifs au service qu'elle assure :

- L'aspect `Moulinette Algo` manipule les champs `Numéro` et `Droits`.
- L'aspect `Invalidation` manipule les champs `Numéro`, `Copyset` et `Proprietaire`.
- L'aspect `Verrouillage` manipule le champ `Verrou`.
- L'aspect `Localisation` le champ `PropProb`.

12.3 Modélisation liée aux différentes fonctionnalités

Dans cette partie, nous allons présenter la modélisation de notre application selon la séparation des préoccupations que nous avons choisie.

12.3.1 Schématisation de la modélisation

Cette modélisation est indépendante de la plate-forme d'accueil. Elle illustre de façon précise la vision de notre séparation (figure 29 page 106).

La page mémoire utilisée dans le système est modélisée par la classe `Page`. Contrairement à la modélisation liée à l'algorithme centralisé, cette classe est directement référencée par les classes qui en ont besoin car nous avons décidé d'instancier qu'une seule page au sein du système dans cette nouvelle modélisation.
La classe `CheminEcriture` est utilisée pour retourner les informations nécessaires au site ayant provoqué un défaut en écriture. Elle dispose simplement d'un vecteur qui sera constitué des numéros de sites ayant été traversés par la requête du défaut associé. Elle implémente la classe `Serializable` car ses instances sont véhiculées sur le réseau.
La classe `CommDistImpl` réalise l'interface `FonctionsDistantes` car elle implémente les trois méthodes d'invocations distantes de RMI.

12.3.2 Fil d'exécution

Pour clarifier la manière dont interagissent les différentes fonctionnalités de notre application, présentons le fil d'exécution lié à une demande en lecture et en écriture. Nous supposons que la demande en lecture engendre un défaut en lecture (ce qui est le cas lorsque la valeur demandée a été invalidée). De plus, nous considérons que le premier propriétaire probable répond au défaut (c'est-à-dire qu'il possède une copie de référence valide, et qu'il est le propriétaire de la page concernée par notre exemple), comme le montre le schéma (figure 30 page 107).

- Dans le cas d'un défaut en lecture :

Sur le site 2 :

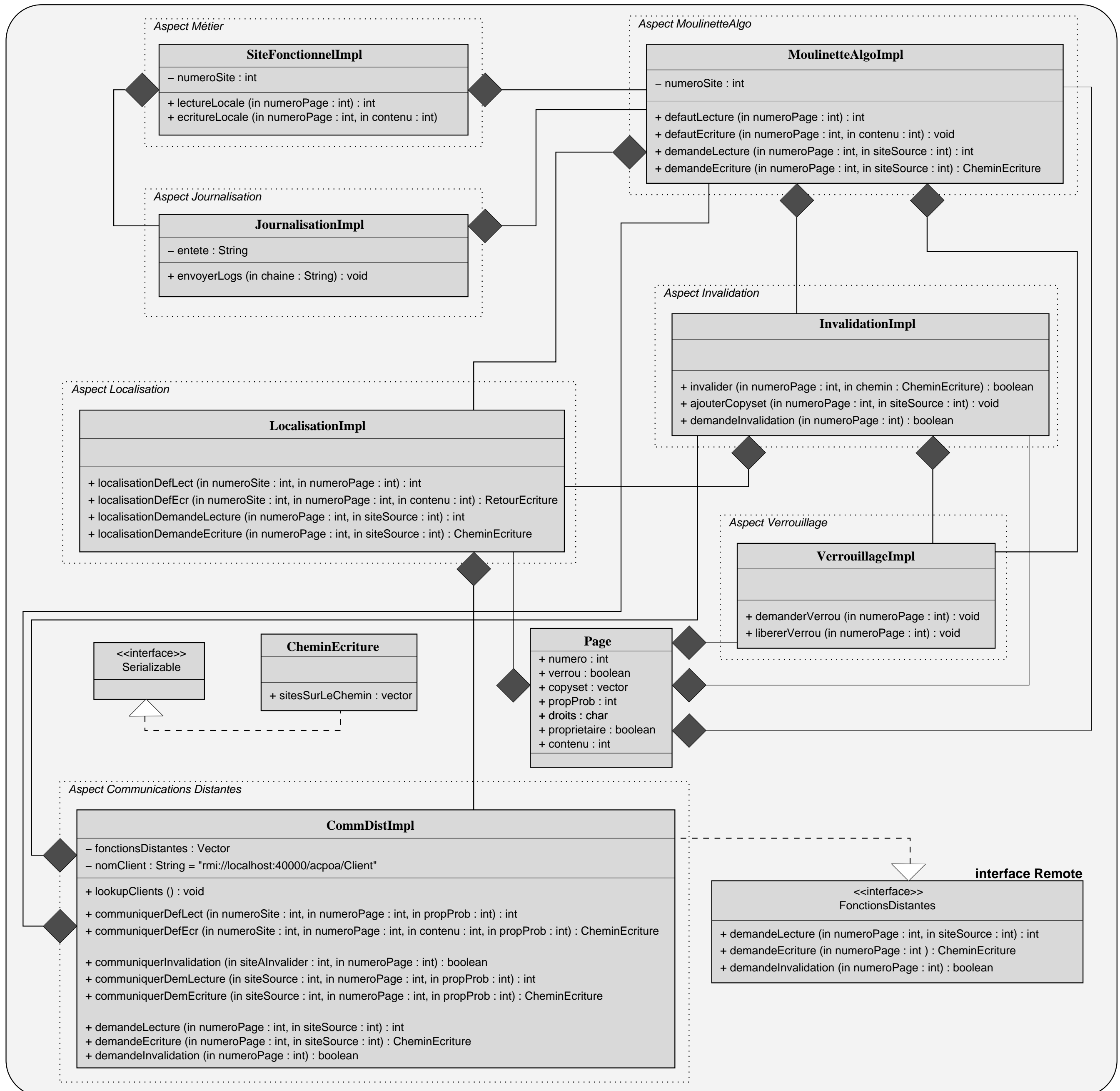

FIG. 29 – Modélisation de l'implémentation (et de notre séparation des préoccupations) de l'algorithme décentralisé

SiteFonctionnel : lectureLocale (numeroPage = 0)
    Journalisation : envoyerLogs (chaine = "logs ...")
    MoulinetteAlgo : defautLecture (numeroPage = 0)

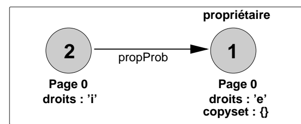

FIG. 30 – Le contexte du site 1 et du site 2 pour illustrer notre exemple

Verrouillage : demanderVerrou (numeroPage = 0)
Localisation : localisationDefLect (numeroSite = 2, numeroPage = 0)
CommDist : communiquerDefLect (numeroSite = 2, numeroPage = 0, propProb = 1)
Invoquation RMI sur le site 1 : demandeLecture (numeroPage = 0, siteSource = 2)
Recuperer le contenu de la page
Verrouillage : libererVerrou (numeroPage = 0)

Sur le site 1 :

CommDist : demandeLecture (numeroPage = 0, siteSource = 2)
MoulinetteAlgo : demandeLecture (numeroPage = 0, siteSource = 2)
Verrouillage : demanderVerrou (numeroPage = 0)
Invalidation : ajouterCopyset (numeroPage = 0, siteSource = 2)
Verrouillage : libererVerrou (numeroPage = 0)
Retourner le contenu de la page
Retourner le contenu de la page

- Dans le cas d'un défaut en écriture :

Sur le site 2 :

SiteFonctionnel : ecritureLocale (numeroPage = 0, contenu = x)
Journalisation : envoyerLogs (chaine = "logs ...")
MoulinetteAlgo : defautEcriture (numeroPage = 0, contenu = x)
Verrouillage : demanderVerrou (numeroPage = 0)
Localisation : localisationDefEcr (numeroSite = 2, numeroPage = 0, contenu = x)
CommDist : communiquerDefEcr (numeroSite = 2, numeroPage = 0, contenu = x, propProb = 1)
Invocation RMI sur le site 1 : demandeEcriture (numeroPage = 0, siteSource = 2)
Recuperer la structure CheminEcriture et la retourner
Invalider : invalider (numeroPage = 0, chemin = struture recue)
Pour tous les sites a invalider (ici seul le 1) :
CommDist : communiquerInvalidation (siteAInvalider = 1, numeroPage = 0)
Invocation RMI sur le site 1 : demandeInvalidation (numeroPage = 0)
Mettre a jour le contenu
Verrouillage : libererVerrou (numeroPage = 0)
Retourner

Sur le site 1 (première requête RMI) :

CommDist : demandeEcriture (numeroPage = 0, siteSource = 2)
    MoulinetteAlgo : demandeEcriture (numeroPage, siteSource = 2)
        Verrouillage : demanderVerrou (numeroPage = 0)
        Retourner une structure CheminEcriture contenant le numero 1
        Verrouillage : libererVerrou (numeroPage = 0)

Sur le site 1 (deuxième requête RMI) :

CommDist : demandeInvalidation (numeroPage = 0)
    Invalidation : invalider (numeroPage = 0)

S'il y avait des sites intermédiaires entre le site 2 et le site 1, nous aurions invoqué les méthodes `demandeLecture` et `demandeEcriture` de l'aspect MoulinetteAlgo, elles mêmes associées aux méthodes `communiquer*` de l'aspect CommDist. Et dans ce cas, comme nous l'avons expliqué, l'invalidation se fait récursivement entre les sites ...

12.4 Transposition de la modélisation

Il est intéressant de voir comment transposer notre séparation des préoccupations et sa modélisation associée dans un modèle d'assemblage de composants techniques et dans un modèle de composition d'aspect (dans le sens AOP).

12.4.1 Modèle d'assemblage de composants techniques

La transposition de notre modèle générique en un modèle d'assemblage de composants techniques est relativement simple (c'est d'ailleurs la méthode que nous avons utilisée dans le cadre de la première implémentation) : à chaque préoccupation, on associe un composant technique qui "encapsule" un objet JAVA qui implémente la fonctionnalité associée. Cependant, dans le cadre de notre exemple, nous avons séparé l'aspect Cohérence en deux "sous-apsects". Nous utiliserons donc un composant associé à cet aspect, qui sera lui-même composé de deux sous-composants. Il nous suffit ensuite d'assembler tous les composants ensemble selon leurs dépendances. On obtient alors un composant de plus haut niveau, "encapsulant" la sémantique de notre application.

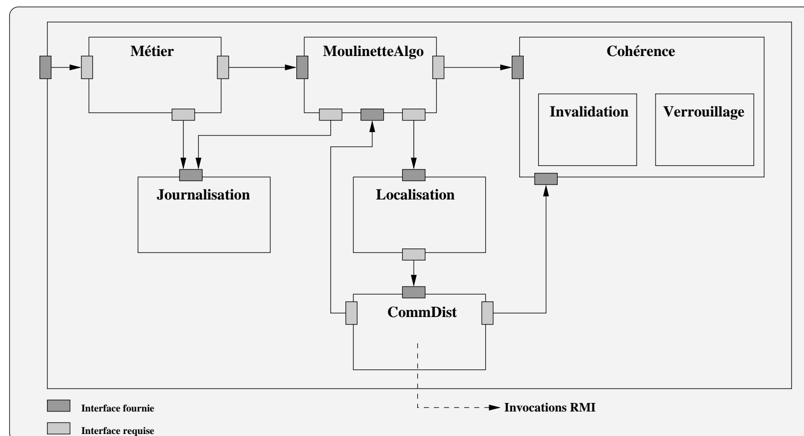

FIG. 31 – Modéle d'assemblage générique avec les composants techniques (algorithme décentralisé)

Un modèle d'assemblage générique pourrait être schématisé de la manière ci-dessus.

De notre modélisation précédente, il est possible de déterminer quelles interfaces sont fournies/requises et par quels composants. Cela traduit les dépendances entre les fonctionnalités.

12.4.2 Modèle de composition d'aspects

Il est difficile de donner un modèle de composition des aspects de notre application sans utiliser la notation UML définie par l'équipe de JAC [PDF+02a]. Etant donné que nous donnons plus loin notre vision de la composition de manière détaillée (figure 34 page 114) nous n'allons pas présenter de figure dans cette partie.
De plus, il faut remarquer que la transposition de notre séparation des préoccupations en un modèle de composition d'aspects peu s'avérer complexe, comme nous le verrons plus tard. A ce stade de la réflexion, nous pouvons dire que nous connaissons les dépendances entre fonctionnalités, ce qui peut se transposer à l'ordre d'exécution des wrappers dans le modèle de JAC. Par exemple, intuitivement, nous pouvons nous douter que le wrapper associé à la fonctionnalité `MoulinetteAlgo` devra s'exécuter avant le wrapper associé à la fonctionnalité `Localisation` ... etc. Par contre, pour une vue d'ensemble plus détaillée de toute cette composition, il nous manque quelques informations que nous expliquerons plus tard dans la partie relative à l'implémentation avec JAC.

12.4.3 Modèles d'implémentations liés aux différentes plates-formes

Dans la prochaine partie, nous allons expliquer l'implémentation de l'algorithme en version décentralisée sur les plates-formes FRACTAL, KILIM et JAC. Cependant, nous ne redonnerons pas le modèle lié à l'implémentation de notre application sur ces plates-formes puisque pour chacune d'entre elles, nous suiverons celui proposé plus haut (figure 29 page 106). Nous avons en effet intégré notre vision de la séparation des préoccupations au sein de ce modèle générique.

12.5 Version implémentée avec FRACTAL

Le modèle utilisé dans le cas de l'implémentation avec FRACTAL est donc le même que celui proposé, aux interfaces nécessaires à la composition près. L'étape de modélisation avec ces interfaces a déjà été expliquée (cf. 10.4.1 page 83).

12.5.1 Le modèle de composition

L'assemblage des différents composants de notre application selon notre séparation des préoccupations peut être schématisé de la manière suivante (figure 32 page 111).

Nous avons décidé d'"encapsuler" la fonctionnalité de cohérence de l'algorithme dans un composant `Coherence`. Il s'agit donc d'un composant composite, composé de deux composants primitifs (`Invalidation` et `Verrouillage`). Les interfaces fournies par ces deux composants primitifs sont exportées sur le composant composite de manière à les rendre visibles pour les autres composants. Il en est de même pour les interfaces que requierent les composants primitifs.
Notre "méta-composant" site correspond à l'application qui sera déployée sur chaque site de notre système. Il encapsule tous les composants nécessaires à la mise en oeuvre de l'algorithme.
Les dépendances entre fonctionnalités sont matérialisées par les assemblages entre les interfaces clientes et serveurs.
Pour configurer notre application, nous avons ajouté des interfaces d'initialisation. L'interface `iInitCOM` permet d'invoquer la méthode `lookupClients` sur l'instance de `CommDistImpl`.

FIG. 32 – Modèle de composition lié à l'algorithme décentralisé avec FRACTAL

L'interface `iInitPage` permet de configurer la référence de l'instance de la classe `Page` (qui sera instanciée de la méthode `main`) auprès des composants qui en ont besoin.
Il en est de même pour l'interface `iInitNumSite` pour configurer l'attribut `numeroSite` nécessaire aux composants `Metier` et `MoulinetteAlgo`. Pour ce dernier attribut, remarquons que nous aurions pu utiliser l'interface FRACTAL `ServiceAttributes` sur ces deux composants. Dans ce cas, le numéro du site aurait été assigné au sein de l'ADL décrivant l'assemblage de notre application (cette solution ne fonctionne qu'avec des types de base, elle n'est pas utilisable pour configurer l'attribut précédent `Page`).
Nous exportons également l'interface métier sur notre composant principal de manière à invoquer les méthodes métier pour notre simulation.

12.5.2 Remarques liées à la plate-forme

Nous avons défini toutes les interfaces nécessaires à la composition de notre application. Toutes les classes qui offrent des interfaces FRACTAL implémentent bien-sûr ces interfaces. Toutes les classes qui requierent des interfaces implémentent l'interface `UserBindingController`. Dans chacune d'elles, il faut donc implémenter les trois méthodes `getFcBindings`, `addFcBinding` et

`removeFcBinding` en fonction des interfaces à lier.

Nous avons utilisé l'ADL de FRACTAL pour décrire notre assemblage.

L'exécution et le déploiement est similaire à la première implémentation que nous avons proposée (cf. 10.4.5 page 88).

12.6 Version implémentée avec KILIM

12.6.1 Le modèle de composition

Le modèle de composition de notre application avec KILIM est bien entendu très similaire à celui de FRACTAL, nous le schématisons de la manière suivante :

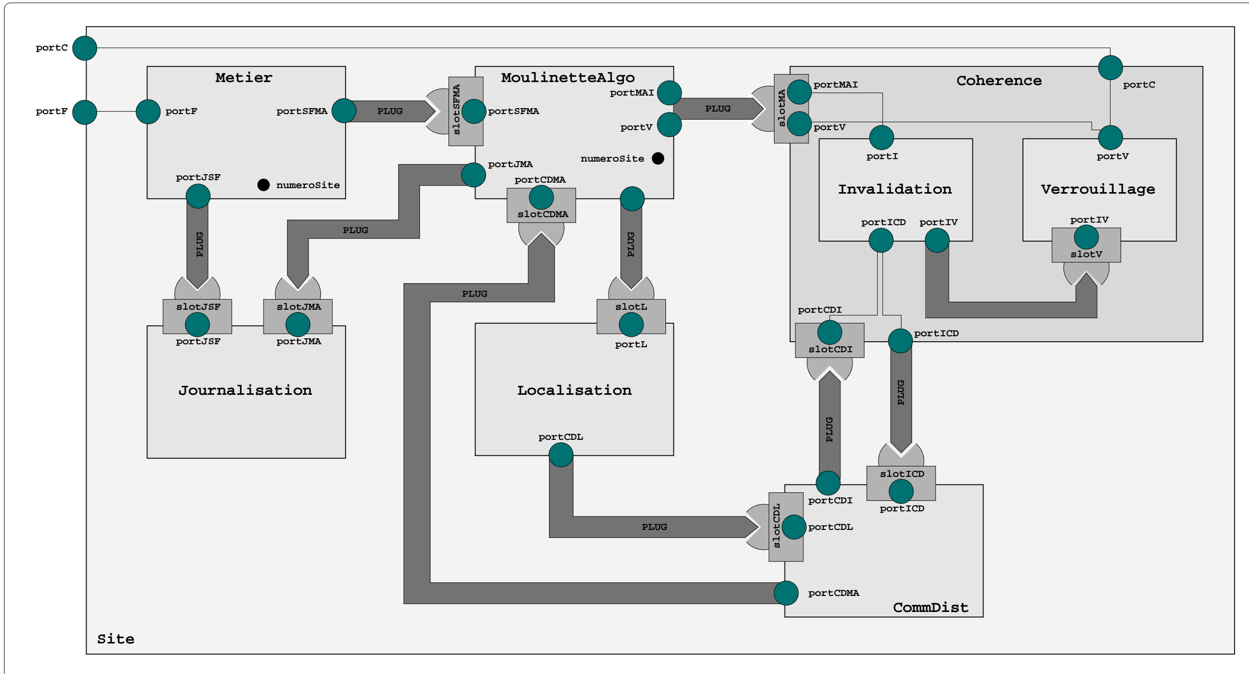

FIG. 33 – Modèle de composition lié à l'algorithme décentralisé avec KILIM

C'est par l'intermédaire du port portC que sera instanciée toute l'application, d'où l'intention de l'exporter sur le composant principal. Il en est de même pour le portF de manière à invoquer les méthodes métier sur l'instance de la classe SiteFonctionnelImpl encapsulée dans le composant Metier.
Deux ports sont associés au slot slotMA de façon à rendre atomique l'opération plug entre les deux composants MoulinetteAlgo et Coherence.
Les lignes liant certains ports du composant Coherence correspondent à des références d'instances qui sont "exportées" de ce composant.
Nous ne revenons pas sur les explications des mécanismes permettant d'instancier l'ensemble des entités du composant principal. Cela a déjà été donné dans le cadre de la première implémentation (cf. 10.5.3 page 90).

12.6.2 Remarques liées à la plate-forme

L'implémentation de l'application avec KILIM se fait de la même manière qu'en JAVA "pur". Il suffit d'implémenter dans chaque classe les méthodes *setters* permettant de configurer les références d'instances nécessaires à l'assemblage.

L'exécution et le déploiement se font de la même façon que celle décrite dans le cadre de la première implémentation (cf. 10.5.4 page 90).

12.7 Version implémentée avec JAC

12.7.1 Le modèle de composition

Le modèle de composition des aspects au sein de la plate-forme JAC se schématise de la manière suivante (toutes les flèches correspondent à des points de jointure dont la cardinalité est identique : à chaque objet de base est associé un et un seul composant d'aspect) (figure 34 page 114).

Notre choix de prendre une granularité plus fine en ce qui concerne notre séparation des préoccupations complique beaucoup notre modèle de composition des aspects par rapport à l'implémentation précédente.

Notre objet de base est représenté par la classe SiteFonctionnelImpl. Elle implémente nos deux opérations métier de lecture et d'écriture.

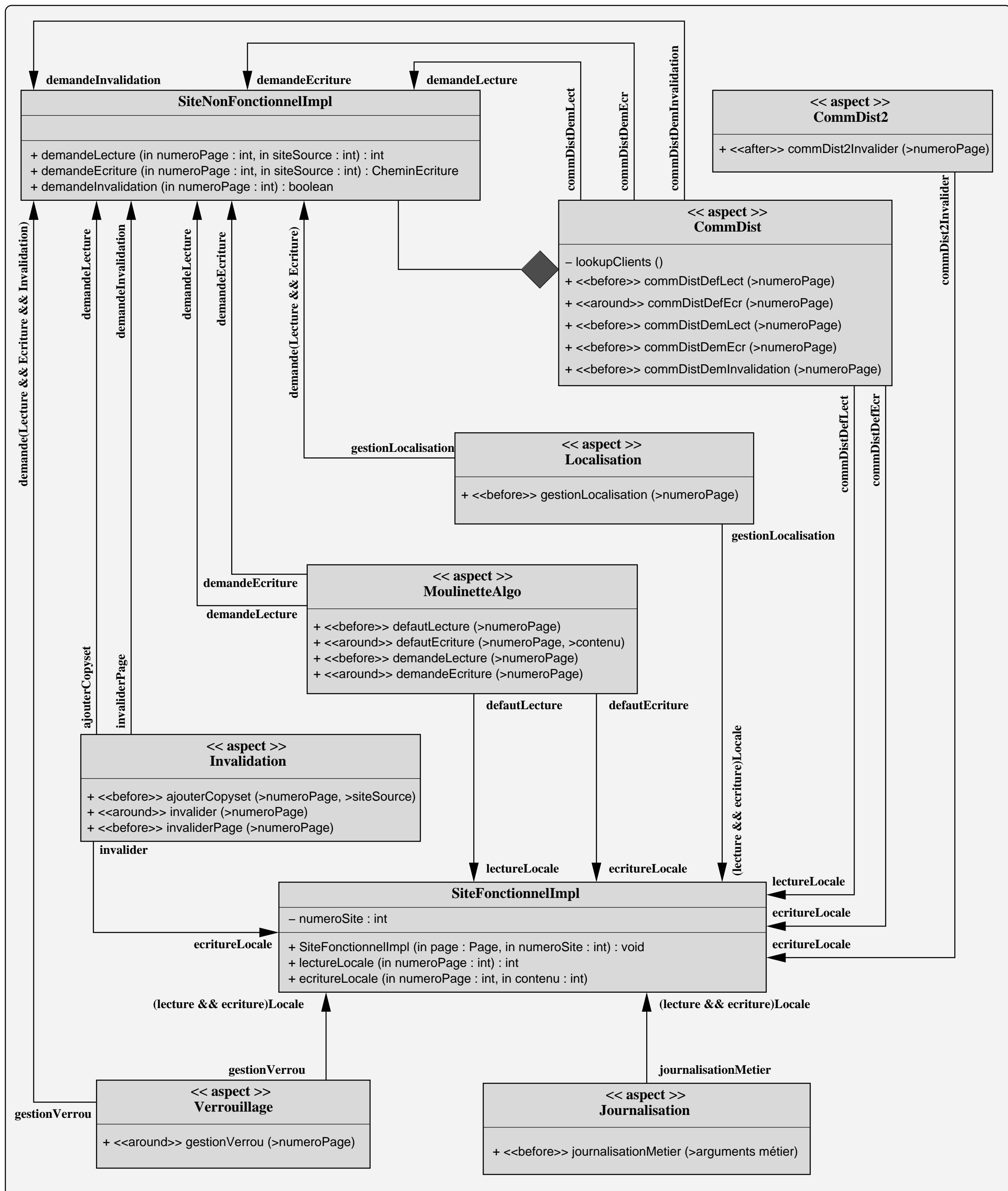

FIG. 34 – Le modèle de composition des aspects de l'algorithme décentralisé avec JAC

Tous les composants d'aspects que nous avons définis seront alors tissés autour de ces deux méthodes comme le montre les points de jointure représentés sur la figure. Nous expliquerons plus loin pourquoi certaines méthodes d'aspects sont exécutées avant, autour ou après les invocations métier, ces paramètres dépendent en effet de la sémantique de l'application.
Nous tisssons également des aspects autour des méthodes distantes invoquées par les requêtes RMI (les méthodes de la classe `SiteNonFonctionnelImpl`). Ces méthodes ne correspondent pas à la logique fonctionnelle que nous avons définie de notre application. En effet, nous aurions pu les intégrer à l'aspect `CommDist` et tenter d'effectuer le tissage nécessaire sur ce composant d'aspect. Même si cette alternative est envisageable au sein de la plate-forme JAC, cette vision des choses aurait compliqué d'avantage l'implémentation. De plus, nous pouvons nous imaginer que ce genre de pratique (tissage autour d'un composant d'aspect) n'est pas vraiment souhaitable dans le paradigme AOP car il introduit de la complexité et de fortes dépendances entre les aspects.

Nous avons donc considéré que le composant d'aspect `CommDist` disposait d'une référence vers la classe `SiteNonFonctionnelImpl` et que sur les instances de cette classe seraient tissés les composants d'aspect nécessaires.
Nous expliquerons plus loin la présence de l'aspect `CommDist2` que nous avons été obligé d'introduire pour faire fonctionner notre application.

Remarquons que sur cette modélisation ne sont pas montrées la dépendance entre les aspects. En effet, nous imaginons bien que les méthodes wrappantes doivent être exécutées dans un ordre qui correspond à la sémantique de l'application.

Les points de jointure associés à l'initialisation de l'application n'ont pas été représentés sur la modélisation (comme cela avait été le cas dans la première modélisation). En effet, les composants d'aspects qui ont besoin d'une référence sur l'instance `Page` (c'est-à-dire `Verrouillage`, `MoulinetteAlgo`, `Localisation` et `Invalidation`) tissent une méthode `initialisation` autour du constructeur de la classe `SiteFonctionnelImpl` de manière à récupérer cette référence (alors en argument du constructeur de la classe).

12.7.2 Remarques liées à la plate-forme

En ce qui concerne l'implémentation, nous avons donc développé les sept composants d'aspects avec leurs classes *wrapper* associées. Comme dans le cadre de la première implémentation, nous nous sommes arrangés de telle sorte que les points de jointure soient définis dans des fichiers texte, à l'extérieur des sources. Le tissage se fait alors dynamiquement au moment de l'exécution de

notre application.

L'ordre dans lequel doivent être tissés les *wrappers* se définit dans le fichier `jac.prop`. Cet ordonnancement est très important, la sémantique de l'application en dépend. Cet ordre est le suivant :

1. Journalisation
2. Verrouillage
3. MoulinetteAlgo
4. CommDist2
5. Invalidation
6. Localisation
7. CommDist

A partir de cet ordre, expliquons quels traitements sont assignés à chaque *wrapper* dans le cadre d'un défaut en lecture (il s'agit là de la transposition du fil d'exécution explicité plus haut, cf. 12.3.2 page 105). Notons que si deux méthodes wrappantes `A` et `B` encapsulent une même méthode métier, le corps `«before»` des wrappers sera exécuté dans l'ordre `A` puis `B`, le corps `«after»` dans l'ordre `B` puis `A`. On se place dans le cadre d'un défaut en lecture effectué par un site dont le propriétaire probable de la page concernée détient en détient une copie :

• Sur le site effectuant le défaut en lecture :
0. «before» Journalisation : On journalise l'accès.
1. «before» Verrouillage : On pose un verrou sur la page concernée.
2. «before» MoulinetteAlgo : Si l'attribut `Droits` associé à la page n'est pas invalide, on continue
le fil de l'exécution, sinon on effectue un "saut" de wrappers jusqu'à la méthode métier.
3. «before» Localisation : On récupère l'attribut `PropProb` associé à la page.
4. «before» CommDist : On invoque la méthode distante RMI pour récupérer la valeur sur le propriétaire probable.
5. Metier : Exécution de la méthode `lectureLocale`.
6. «after» MoulinetteAlgo : On met à jour le contenu de la page mémoire.
7. «after» Verrouillage : On libère le verrou sur la page concernée.

• Sur le site recevant la requête RMI :
0. «before» Verrouillage : On pose un verrou sur la page concernée.
1. «before» MoulinetteAlgo : Si l'attribut `Droits` n'est pas invalide, on continue le fil de l'exécution, sinon on exécute le wrapper associé à l'aspect `Localisation`.
2. «before» Invalidation : On ajoute le site source au *copyset* de la page concernée

puis “saute” l’exécution des autres wrappers pour exécuter la méthode de base.
3. «before» Localisation : On récupère l’attribut `PropProb` associé à la page.
4. «before» ComDist : On invoque la méthode distante RMI pour récupérer la valeur sur le propriétaire probable.
5. Méthode de base : Exécution de `demandeLecture`.
6. «after» Verrouillage : On libère le verrou.

Schématisons le fil de l’exécution (en fonction de son contexte) du site recevant la requête RMI :

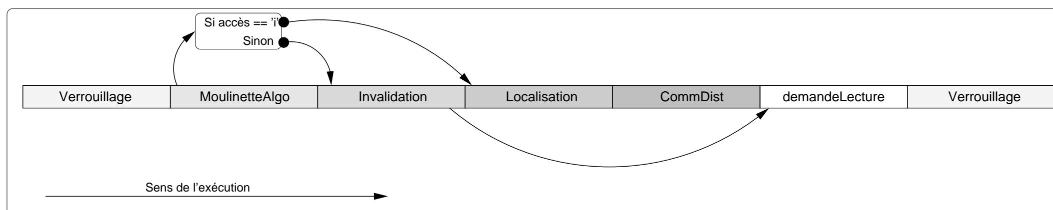

FIG. 35 – Schématisation du fil d’exécution des wrappers lors de la réception de notre requête RMI

Bien entendu, le retour “réel” de notre méthode de base se fait à la fin du fil d’exécution.

De cette exemple, nous pouvons donc tirer deux remarques très importantes :

1. Selon le contexte, le fil d’exécution peut changer. C’est bien ce que nous avons mis en évidence avec cet exemple : si le propriétaire probable qui reçoit la requête détient un exemplaire de la page, il doit alors ajouter le site source au *copyset* lié à cette page (d’où le passage par le wrapper `Invalidation`) et “il” exécute alors la méthode de base sans passer par les wrappers `Localisation` et `ComDist` puisque il n’y a pas d’autres propriétaires probables à contacter. Dans le cas contraire, “il saute” le wrapper `Invalidation` (il n’y a rien à ajouter au *copyset* puisqu’il ne détient pas d’exemplaire de la page) et sous-traite le reste de l’exécution aux wrappers gérant la localisation et les communications distantes.
   Au sein des wrappers, il faut donc effectuer des tests permettant d’adapter le fil d’exécution en fonction du contexte.

2. Nous ne l'avons pas explicité dans l'exemple mais un autre point important consiste au fait que les wrappers doivent s'échanger des informations le long du fil d'exécution. Nous avons déjà expliqué comment les wrappers peuvent récupérer les arguments de la méthode de base, mais dans notre cas, nous avons besoin d'informations "non-fonctionnelles", ne faisant donc pas partie des arguments métier. Par exemple, le wrapper `CommDist` doit disposer du numéro du propriétaire probable qu'il doit éventuellement contacter. Et seul le wrapper `Localisation` peut lui fournir.
Il s'agit là d'un point important à considérer lorsque l'on utilise le paradigme AOP. Au sein de JAC, nous avons utilisé la classe `Collaboration` définie dans le package `jac.core` qui permet donc d'échanger des attributs, de faire collaborer les wrappers.

Nous n'allons pas expliciter toutes les étapes liées à un défaut en écriture, nous allons nous contenter de les schématiser :

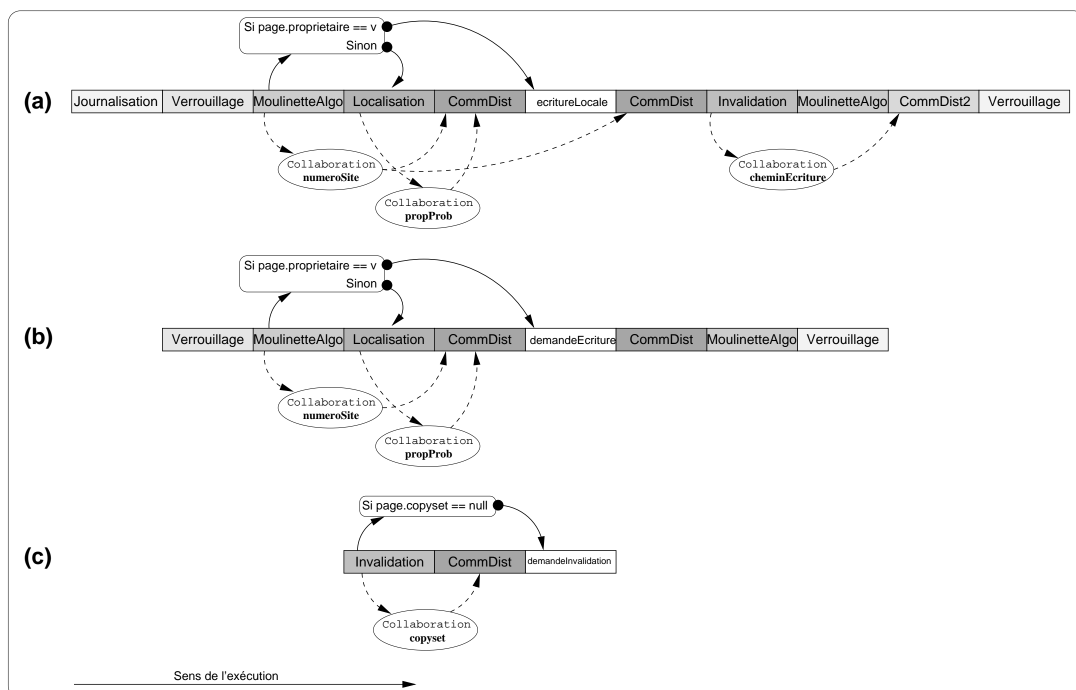

FIG. 36 – Schématisation des différents fils d'exécution des wrappers lors d'un défaut en écriture

Sur cette figure, la partie (a) représente les wrappers qui entourent la méthode `ecritureLocale`, la partie (b) la méthode `demandeEcriture` (invoquée

par une requête RMI) et la partie (c) la méthode `demandeInvalidation`.
En plus des tests à effectuer en fonction du contexte d'exécution, nous avons ajouté certaines `Collaborations`. Une flêche pointant vers la `Collaboration` indique que le wrapper associé l'initialise, une flêche dans le sens opposé indique que le wrapper l'utilise. Bien entendu, les wrappers utilisent également les arguments donnés par la méthode métier qu'ils encapsulent mais nous ne les avons pas représentés sur le schéma.
Par exemple, pour le premier cas (a), l'aspect `MoulinetteAlgo` initialise une `Collaboration` avec le numéro du site associé (le wrapper associé à l'aspect connaît la valeur de l'attribut `numeroSite` depuis le processus d'initialisation). Cette information est nécessaire à l'aspect `CommDist` puisque celui-ci invoque une méthode RMI qui demande en argument le site source.
En ce qui concerne les tests effectués : par exemple, dans le cas (a), l'aspect `MoulinetteAlgo` teste si le site est propriétaire de la page sur laquelle s'effectue le défaut. Si c'est le cas, les aspects `Localisation` et `CommDist` ne doivent pas être exécutés puisque le changement de contenu se fait localement au site (l'aspect `CommDist` dans ce cas sera tout de même sollicité plus tard pour gérer le mécanisme d'invalidation).
Une remarque est d'ailleurs à faire en ce qui concerne ce mécanisme d'invalidation : en effet, lorsque l'on spécifie un ordre d'exécution pour les wrappers, il n'est pas possible de le changer au moment de l'exécution de la chaîne wrappante (ou alors en modifiant certains attributs du package `jac.core` mais cela n'est pas satisfaisant et peut être dangereux). Nous avons donc ce problème dans le cas (a) : on ne peut lancer le mécanisme d'invalidation dans le `«after» CommDist`, puisque dans ce wrapper, nous n'avons pas connaissance des sites à invalider (c'est dans l'aspect `Invalidation` que nous avons décidé de gérer le mécanisme). D'où la nécessité d'ajouter un aspect `CommDist2`. Dans notre cas, l'aspect `Invalidation` récupère la structure de données `CheminEcriture` et la stocke dans un vecteur au sein d'une `Collaboration` et ainsi transmettre ces informations à l'aspect `CommDist2`.
Bien entendu, au niveau de l'implémentation, nous aurions pu nous arranger pour ne pas avoir à créer l'aspect `CommDist2`, mais cela permet de mettre le doigt sur des problèmes que l'on est susceptible de rencontrer dans le paradigme de programmation par aspects.

L'exécution et le déploiement de notre prototype avec JAC se fait de la même manière que ceux décrits dans la première implémentation (cf. 10.6.5 page 95).

Dans cette troisième partie, nous avons détaillé les étapes liées à l'implémentation de deux prototypes d'applications réparties : le premier relatif à une version centralisée de l'algorithme de Kai Li et Paul Hudak (avec une séparation des préoccupations de granularité élevée) et le deuxième relatif à une version décentralisée de cet algorithme (avec une séparation des préoccupations de granularité beaucoup plus fine). Ces implémentations nous ont permis d'expliciter les différentes étapes de notre réflexion en terme de vision de la séparation des fonctionnalités, en terme de vision de l'application et de l'assemblage de ces différentes fonctionnalités relativement aux plates-formes d'accueil et aux paradigmes de programmation étudiés.

La dernière partie de ce rapport est dédiée à la comparaison entre les plates-formes Fractal, Kilim et Jac.

Quatrième partie

Autres implémentations et comparaisons

Dans cette dernière partie, nous tentons de comparer les différentes plates-formes étudiées dans le cadre de ce stage.
Dans un premier temps, nous étudions les capacités des plates-formes en terme d'assemblage réparti des composants et en terme d'évolutivité.
Dans un deuxième temps, les concepts et abstractions des différents modèles sont comparés relativement aux points énoncés dans notre modèle générique de composant (cf. 2.2 page 12). Notre comparaison se poursuit ensuite en terme de cycle de développement, d'impact de la séparation des préoccupations sur le code, de dispersion de code, de vision de l'application, de quantification en terme de lignes de code et de performances.

13 D'autres implémentations ...

Dans cette partie, nous donnons quelques explications sur d'éventuelles autres implémentations que nous aurions pu imaginer mais que nous n'avons pas eu le temps de réaliser.

13.1 La composition répartie

Dans le cadre des deux implémentations que nous avons proposées, nous n'avons pas considéré d'assemblage réparti de composants. En effet, nous avons toujours assemblé les entités logicielles site par site en considérant que la répartition ne faisait pas partie intégrante de notre modèle de composition. Le déploiement de l'application s'est toujours effectué "manuellement" par l'intermédiaire de la fonction `main` et par des invocations explicites de méthode RMI pour effectuer les *lookups* nécessaires à la mise en oeuvre des mécanismes répartis.
Cependant, sur certaines plates-formes étudiées, il est possible d'intégrer la répartition au modèle de composition de l'application.

13.1.1 Avec FRACTAL

Il est possible d'utiliser FRACTAL RMI [Fra]. Il s'agit de l'implémentation d'un ensemble de composants FRACTAL qui fournissent une "usine de liaison" (*binding factory*) permettant d'assurer des liaisons distantes entre les composants FRACTAL de notre application. Ces composants sont basés sur le framework de JONATHAN (ORB générique écrit en JAVA) mais l'équipe de FRACTAL a développé sa propre personnalité RMI au-dessus de JONATHAN.

Par exemple, si nous reprenons le modèle de composition que nous avons proposé dans la cadre de la première implémentation (figure 20 page 85), nous aurions procédé de la manière suivante : le composant `cL` aurait exporté une interface cliente sur la membrane du "super" composant `SiteImpl` (nommé `iSL`. Le composant `sNF` aurait exporté une interface cliente `iCSL` et une interface serveur `iSCL`. Nous aurions implémenté deux nouveaux composants FRACTAL : un composant `ServeurLogs` avec une interface serveur `iSL` et un composant `ServeurLocalisation` avec une interface serveur `iCSL` et une interface cliente `iSCL`. Cinq sites auraient été actifs dans notre système, un par site client, un pour le serveur de localisation et un dernier pour le serveur de journalisation. Pour chaque site, nous aurions effectué les liaisons entre les composants distants suivants :

- `iSL` de `SiteImpl` vers `iSL` de `ServeurLogs`.
- `iCSL` de `SiteImpl` vers `iCSL` de `ServeurLocalisation`.
- `iSCL` de `ServeurLocalisation` vers `iSCL` de `SiteImpl`.

Néanmoins, nous n'avons pas étudié les capacités de déploiement d'une telle méthode.

13.1.2 Avec KILIM

Il n'est pas possible avec KILIM de concevoir un assemblage réparti de l'application (de manière automatique et transparente) car dans l'implémentation de la version actuelle, on ne peut pas récupérer un `port` associé à un *template* distant.

13.1.3 Avec JAC

Au sein de la plate-forme JAC sont implémentés des mécanismes permettant (par l'intermédiaire d'un aspect de déploiement) de déployer les instances de l'application sur des sites distants automatiquement et d'assurer également un tissage des composants d'aspects sur des objets distants.

Pour cela, il suffit de spécifier les sites distants actifs par l'intermédiaire du fichier `jac.prop`, d'initialiser un serveur JAC sur chacun de ces sites et de configurer l'aspect de déploiement pour spécifier quelles instances doivent être déployées.

Avec cette méthode, nous bénéficions donc d'une vue générale de notre application répartie au sein du modèle de composition.

13.2 Evolutivité : cohérence causale

Nous aurions pu implémenter une version de l'algorithme de KAI LI et PAUL HUDAK maintenant une cohérence causale des données du système avec une gestion par horloge vectorielle, plutôt que le mécanisme d'invalidation que nous avons implémenté.
Chaque site maintient une horloge vectorielle, celle-ci évolue au cours du temps, en fonction des demandes de pages. La gestion de la cohérence causale s'effectue grâce à ces estampilles. Chaque site effectuant une demande de page (en

lecture ou en écriture) récupère l'horloge vectorielle du site ayant répondu à la requête. Toutes les pages dont les estampilles sont inférieures à l'estampille la plus récente sont alors invalidées.
Nous n'allons pas rentrer dans les détails du déroulement d'un tel algorithme, nous nous contenterons d'évaluer la possibilité d'implémenter ce mécanisme à la place de celui proposé dans le cadre de la deuxième implémentation.

La structure de donnée relative à une page mémoire est identique à celle proposée (cf. 11.1 page 96) mis à part le champ `Copyset` qu'il faut supprimer et le champ `Estampille` qu'il faut ajouter. Ce champ contient la valeur de l'horloge vectorielle du site sur lequel s'est produit la dernière écriture au moment où celle-ci s'est produite, elle sert donc à matérialiser la relation de dépendance causale entre une version d'une page (dernière écriture sur celle-ci) et toute copie [GS02].

Par rapport à l'implémentation (et la séparation des préoccupations) que nous avons proposée), la mise en place de ce nouveau mécanisme de gestion de la cohérence touche la fonctionnalité `MoulinetteAlgo` (c'est elle qui maintient l'horloge vectorielle c'est-à-dire récupère l'estampille lors d'une demande de page, et l'incrémente lors d'un défaut en écriture), la fonctionnalité `Invalidation` (puisque le mécanisme se base sur une nouvelle méthode), sur les méthodes relatives aux défauts en lecture et écriture (puisqu'il faut ajouter en paramètre l'estampille dont la valeur est véhiculée sur le réseau).

Sur les trois plates-formes, il faudrait donc modifier les trois méthodes de l'interface `FonctionsDistantes` (cf. 29 page 106) comme cela :

• `demandeLecture (in numeroPage : int) : nvlStruct`
• `demandeEcriture (in numeroPage : int, in siteSource : int) : long`
• suppression de `demandeInvalidation`.

Nous n'avons plus besoin du paramètre `siteSource` de la méthode `demandeLecture` puisque nous n'avons plus de *copyset* à gérer. Le paramètre de retour `nvlStruct` de cette méthode serait une structure (un objet) composée du contenu de la page et de l'estampille du site receveur. Nous n'avons plus besoin de la méthode `demandeInvalidation` puisque le processus d'invalidation est local à chaque site. Il est exécuté après chaque exécution d'un défaut.

Pour les trois plates-formes, il faut donc :

- Modifier la fonctionnalité `CommDist` dont la classe associée implémente cette nouvelle interface.

- Modifier la fonctionnalité `Invalidation` pour adapter le code lié au nouveau mécanisme d'invalidation par horloge vectorielle.
- Modifier la fonctionnalité `MoulinetteAlgo` pour adapter le code lié à la réception de l'estampille dans le cas de demande de page et son incrémentation dans le cas de défaut en écriture.

Pour FRACTAL et KILIM, il s'agit donc de modifier le code de trois classes JAVA, dans le cas de JAC le code d'une classe JAVA et de trois classes *wrapper*.

La fonctionnalité `Invalidation` n'a donc plus de lien de dépendance avec la fonctionnalité `CommDist`. Dans le cas de FRACTAL il faut donc supprimer le code lié aux interfaces de composition entre les deux fonctionnalités, dans KILIM supprimer la méthode *setter* associée ainsi que les abstractions au niveau du modèle permettant cet assemblage. Avec JAC, on se passe donc de l'aspect `CommDist2` et il suffit de supprimer le point de jointure qui était défini par `CommDist` sur la méthode `demandeInvalidation`. Dans ce cas, la méthode wrappante `gestionInvalidation` sera exécutée en `«after»`, c'est-à-dire après l'exécution de la méthode métier.

Sur FRACTAL et KILIM, le fait de changer les paramètres des méthodes `demandeLecture` et `demandeEcriture` a un impact très important sur le reste de l'application. En effet, beaucoup de méthodes de `CommDistImpl`, `LocalisationImpl` et `MoulinetteAlgoImpl` doivent être modifiées. Etant donné que les composants de ces deux plates-formes sont assemblés par l'intermédiaire des appels de méthodes, nous avons beaucoup de code à modifier.
Par contre, dans le cas de JAC, ce changement n'influe pratiquement pas sur l'assemblage des composants d'aspect, il suffit d'utiliser une `Collaboration` ou de récupérer simplement dans certaines méthodes wrappantes le paramètre de retour `estampille`.

Beaucoup d'efforts d'implémentation auraient donc été nécessaires pour modifier le code relatif à la mise en place de ce nouveau mécanisme de gestion de la cohérence dans le cadre des plates-formes de programmation orientées composant. Avec la plate-forme JAC, les efforts d'implémentation auraient été plus réduits.

14 Remarques, comparaisons, conclusions

14.1 Comparaisons des points du modèle

Dans cette partie, nous allons tenter de faire les rapprochements entre le modèle de composants que nous avons donné en introduction (cf. 2.2 page 12) et les modèles des différentes plates-formes que nous avons présentés/utilisés.

14.1.1 Fractal

1. Encapsulation, abstraction, indentité

   Dans le modèle concret de Fractal, l'encapsulation est bien respectée puisque les points d'entrée des composants ne peuvent être que les interfaces de contrôle (formulées par les spécifications) ou les interfaces métier spécifiées par le programmeur de l'application. L'implémentation de ces interfaces est complètement masquée à l'environnement dans lequel sera plongé le composant.

   La notion d'identité de composant est très forte dans Fractal. En effet, tout composant Fractal doit implémenter l'interface `ComponentIdentity` qui permet de récupérer toutes les références d'interfaces externes du composant. L'identité d'un composant est donc caratérisée par l'ensemble des interfaces externes qu'il propose. De plus, comme nous l'avons expliqué dans les spécifications, les interfaces sont fortement typées. Les interfaces de type client permettent de matérialiser un service que requiert un composant, les interfaces serveur, un service que fournit un composant. Les modes de coopération entre interfaces peuvent être de type synchrone (*two way*) ou de type asynchrone (*one way*). Comme il est spécifié dans Fractal, les propriétés configurables du composant sont gérées par l'intermédiaire d'une interface spéciale (nommée `AttributeController`), permettant d'accéder à des attributs internes du composant via les méthodes *setter/getter*. Cependant, les attributs qu'il est possible de configurer par l'intermédiaire de cette méthode ne peuvent être que des types de "base", tels des chaînes de caractères, des entiers, des *long* ... etc. Il n'est pas possible en niveau de l'ADL de spécifier des instances d'objets Java.

2. Composabilité

   Quelle que soit la méthode utilisée pour assembler les différents composants de l'application (en utilisant les méthodes de l'API ou par le langage ADL de Fractal), les règles d'assemblage, basées sur les liaisons entre interfaces clientes et serveur sont très explicites. De plus, une vérification de la sémantique d'assemblage est rendue possible grâce aux typages des interfaces (seules deux interfaces de même type peuvent-être liées entre

elles). Le modèle de composant de Fractal est récursif, un premier assemblage de composant devient donc une nouvelle unité de composition (comme nous avons pu le voir dans le cadre de l'implémentation de l'algorithme décentralisé avec le composant `Coherence` qui encapsulait deux composants primitifs) susceptible d'être utilisée en sein d'un composant de plus haut niveau.
L'assemblage peut se faire de manière dynamique au cours de l'exécution de l'application. Pour le moment, la sémantique de l'assemblage entre composants dans le modèle concret n'est que structurelle mais l'équipe de FRACTAL travaille sur d'autres sémantiques de composition.

3. Caractéristiques non-fonctionnelles
   Les spécifications de Fractal tendent à vouloir normaliser les aspects non-fonctionnels que l'on est susceptible d'utiliser dans une approche de programmation orientée composant, mêmes si des efforts sont encore à faire dans ce sens.
   Les caractéristiques non-fonctionnelles des composants sont donc gérées par les interfaces de contrôle du contrôleur (le contrôleur peut donc être associé à un méta-objet permettant de contrôler le contenu du composant) : comme par exemple la gestion du cycle de vie des composants (interface `LifeCycleController`), la gestion des mécanismes de liaison entre interfaces (`BindingController`, la gestion du contenu des composants (`ContentController`) ... etc.
   La configurabilité des aspects non-fonctionnels peut s'imaginer au travers des alias de composants génériques. En effet un composant générique sera associé à un type de comportement bien précis (par exemple composant avec état ou sans état, composant primitif ou composite ...). Cependant, les types de composants sont immuables, ils ne peuvent être modifiés à l'exécution.
   En ce sens, on peut imaginer une certaine séparation des préoccupations interne au sein de la plate-forme, d'un côté les interfaces métier du composant, de l'autre les interfaces de contrôle du contrôleur du composant. Cependant la vision des aspects non-fonctionnels se résument à des mécanismes relatifs à la gestion des composants. Pour le développeur de l'application, ses objets "fonctionnels" et "non-fonctionnels" seront considérés au même titre dans la plate-forme.

14.1.2 KILIM

1. Encapsulation, abstraction, identité
   Pour un composant Kilim, les points d'entrée (à l'exécution) sont donnés par les `ports`, les `properties` et éventuellement les `providers`. Cependant, les `transformers` peuvent être également considérés comme

des points d'entrée mais du point de vue du mécanisme d'instanciation (cette remarque caratérise l'ambiguïté entre la vision d'un composant décrit par un ADL et celle d'un composant exécutif). A ce niveau du modèle, ces abstractions permettent une bonne encapsulation du composant Kilim, mais lorsque l'on atteint le niveau du langage et les règles de mapping entre ces deux niveaux, l'encapsulation peut paraître plus ambigüe, comme par exemple si l'on considère qu'un `provider` peut être sous forme de constructeurs, de méthodes de type *getter* ou de méthodes distantes. En ce sens, les points d'entrée du composant peuvent être de différentes formes, et donc toucher de nombreux points du code. Pour clarifier la vision de l'application, il faudrait s'astreindre à des régles méthodologiques de modélisation et d'écriture de code, comme par exemple la séparation (au sein d'une classe) entre interfaces fonctionnelles et interfaces de configurations et ne gérer que ces-dernières au niveau de la méta-représentation KILIM de l'application.
L'identité d'un composant Kilim se caractérise par une instance de *template* (au niveau de l'ADL) et par une classe `RtComponent` (à l'exécution, au niveau de la méta-représentation de l'application). Les notions d'interfaces requises et fournies sont bien intégrées dans le modèle via les `ports` (*offered* ou *required*). Le mode de coopération entre interfaces se fait par appels de méthodes. Les propriétés configurables des composants sont gérées par les `transformers`, dont les formes les plus communes sont les méthodes de type *setter*. Comme nous l'avons expliqué ci-dessus concernant la plate-forme FRACTAL, les propriétés configurables au sein de KILIM ne peuvent être que des types de base.

2. Composabilité
   L'assemblage des composants se fait par l'intermédiaire des opérateurs `plug` et `bind` au sein de l'ADL. Les composants seront alors liés par leurs `slots`. Les règles de composition sont explicitement décrites au sein de la méta représentation (c'est-à-dire la représentation de l'assemblage avec les abstractions de KILIM) de l'application par l'utilisation des `providers` et des `triggers` qui permettent respectivement de fournir des références d'instances et de se les échanger. Cependant, il n'y a pas de vérification sémantique dans la version actuelle de KILIM (on peut tout de même considérer une certaine vérification dans le sens où les événements déclenchant des appels de méthodes retourneront une erreur si cette méthode n'existe pas sur l'instance concernée ou si les paramètres fournis ne correspondent pas à la signature de cette méthode, mais cette vérification se fera au moment du mécanisme d'instanciation et non au moment de l'analyse de l'ADL), la sémantique de l'assemblage devra donc être à la charge du programmeur. L'assemblage peut-être modifié dynamiquement, au cours de l'exécution de l'application. La sémantique d'assemblage est structurelle.

3. Caractéristiques non-fonctionnelles

   Les caractéristiques non-fonctionelles de gestion des composants ne sont pas prises en compte dans le modèle de KILIM. Mis à part le fait qu'il est possible de reconfigurer à l'exécution l'assemblage d'une application ou une propriété (dans ce cas, la plate-forme a connaissance de la méta représentation de l'application et lors d'un changement d'un composant, tous les `triggers` concernés seront réexécutés de manière transparente) il n'y a pas dans KILIM la notion de cycle de vie d'un composant (un composant est uniquement dans l'état initialisé ou non initialisé, c'est-à-dire instancié ou non), de contrôle de contenu (...) comme on peut le trouver dans le modèle FRACTAL.

   En effet, KILIM est un framework orienté configuration et assemblage de classe, les objectifs ne sont donc pas les mêmes que FRACTAL. La notion de séparation des préoccupations peut se faire via les interfaces des objets contenus dans les composants. En ce sens, au niveau du modèle, les `slots` peuvent refléter l'aspect d'un composant, avec d'un côté les `slots` traitant de la logique métier de l'application et de l'autre les `slots` associés à la logique non-fonctionnelle. Mais cette séparation doit alors se faire au moment de la conception de l'application, et ne concerne pas vraiment la plate-forme KILIM.

   Cependant, sans parler d'aspects non-fonctionnels, on peut imaginer une certaine séparation des préoccupations dans KILIM, entre les interfaces métiers de l'application (associées aux `ports`) et les interfaces de configuration (pouvant être associées aux `transformers`).

   La flexibilité de déploiement est assurée par le fait que les `templates` sont des unités opérationnelles car ils contiennent toutes les informations nécessaires à la plate-forme pour initialiser l'application, la configurer et instancier ses composants.

14.1.3 JAC

JAC n'est pas une plate-forme d'assemblage de composants mais implémente les concepts de programmation par aspect. Aussi, il est parfois difficile de trouver une correspondance entre les concepts de l'AOP et les critères du modèle de composant que nous avons élaboré.

1. Encapsulation, abstraction, indentité

   Dans le paradigme AOP, les points de jointure peuvent-être considérés comme des points d'accès aux aspects de base (les objets de base qui implémentent la logique métier de l'application). L'abstraction d'un composant (c'est-à-dire que celui-ci ne doit pas révéler plus de détails qu'il

n'en faut pour qu'il soit intégré à son environnement) n'est pas vraiment transposable dans le paradigme AOP puisque les points de jointure peuvent être associés à n'importe quelle méthode des objets de base. La notion d'identité de composant est également difficile à appréhender. On peut dire qu'il existe deux identités de composants : les composants de base et les composants d'aspects, et il semble difficile d'être plus précis. Les notions d'interface (services) requises ou fournies n'ont pas vraiment leur place, mis à part le fait que l'on peut considérer que les méthodes et exceptions implémentées dans les composants d'aspect sont susceptibles de fournir un service non-fonctionnel aux composants de base. Les communications entre les deux types de composants se font dans JAC par l'intermédiaire des trois types de méthodes d'aspect (*wrapping method*, *role method* et *exception handler*). Les propriétés configurables des composants ne sont pas gérables au niveau du modèle, elles sont à la charge du programmeur (notons que nous ne parlons pas ici de la configuration des aspects, synonyme de déploiement des aspects, mais d'éventuelles propriétés configurables au sein même des composants d'aspect).

2. Composabilité
   L'assemblage des composants d'aspect et des composants de base se fait par l'intermédiaire du déploiement des aspects, "le long" des coupes transversales associées à l'aspect. Les règles d'assemblage de composants sont explicites, puisqu'un ensemble de points de jonction doit être spécifié de manière explicite pour assurer le déploiement. On ne peut pas dire qu'une vérification de la sémantique d'assemblage soit effectuée au sein de la plate-forme, dans le sens où un aspect peut être tissé autour de points de jonction qui n'ont rien à voir avec l'aspect en question. Cependant, à un autre niveau d'abstraction, on peut imaginer que la sémantique de l'assemblage est vérifiée, lorsque par exemple on utilise un aspect de composition qui se chargera d'ordonnancer les aspects (dans le cas de plusieurs aspects tissés autour d'un même point de jonction) et donc d'assurer la cohérence de la composition de l'aspect, et donc de l'assemblage entre les composants de base et les composants d'aspect. L'assemblage entre les composants est complètement dynamique au sein de JAC puisque les aspects peuvent être tissés et détissés au cours de l'exécution de l'application. La notion d'assemblage récursif n'est pas transposable dans le modèle de composant de JAC, l'équivalent pourrait-être de considérer qu'il est possible de tisser un nombre illimité d'aspects autour d'un point de jonction.
   La sémantique de composition de JAC peut être considérée comme comportementale, dans le sens où les manifestations du tissage d'un aspect se feront en fonction du comportement de l'application (par exemple lors de l'appel d'une méthode encapsulée par une méthode *wrappante*).

3. Caractéristiques non-fonctionnelles

   Dans le cas de JAC, nous sommes au centre de l'ambigüité énoncée dans la partie d'introduction du rapport (cf. 3.2 page 18).

   En ce qui concerne les caractéristiques de gestion des composants, on peut imaginer qu'elles sont assurées par la notion de tissage (issue du paradigme AOP) dynamique. En effet, à l'exécution de l'application, il est possible de tisser/détisser des aspects autour des objets de base. Si l'on utilise les aspects pré-programmés du framework, on peut considérer qu'il est possible de bénéficier d'un certain contrôle de comportement de nos objets de base et en quelque sorte, faisant partie intégrante de la plate-forme puisqu'il s'agit de fonctionnalités développées au sein du framework. Il est également possible de (re)configurer dynamiquement ces aspects. Les objets de base et les composants d'aspect sont instanciés dans un containeur JAC, assurés de manière transparente par la plate-forme. Il en est de même avec la notion de cycle de vie que l'on peut associer au mécanisme de tisssage/détissage. Un aspect de déploiement a été également développé de manière à automatiser les instanciations réparties sur différents sites (utilisant JAVA RMI).

14.2 Tableau récapitulatif

Récapitulons nos remarques au sein d'un tableau commun (page suivante) permettant de comparer les différents concepts mis en œuvre dans les trois plates-formes étudiées.

Caractéristiques		FRACTAL	KILIM	JAC
Encapsulation (points d'entrée)		Interfaces de contrôle et interfaces métier	`ports`, `properties` et `providers`	Points de jonction
Identité		Référence sur un ensemble de types d'interfaces supportées	Instances de `templates` et `RtComponent`	Objets de base vs. composants d'aspect
	Interfaces	Clientes et serveurs	`ports` *offered* ou *required*	-
	Modes de coopération	Opération synchrone (*tow way*) ou asynchrone (*one way*)	Appels de méthodes	Types de méthodes d'aspect
	Propriétés (métier) configurables	Par une interface de contrôle (méthodes *setter*)	Par les `transformer` (méthodes *setter*)	Non pris en compte dans le modèle
Composabilité				
	Vérification sémantique	Assurée par le fort typage des interfaces	A la charge du programmeur (pas de typage)	Possibilité d'utiliser un aspect de composition pour assurer la cohérence de tissages inter-aspects
	Récursivité	Assurée par les composants composites	Assurée par composition de `templates`	Composition illimitée d'aspects
	Sémantique	structurelle	structurelle	comportementale
	Dynamicité	Assurée	Assurée	Assurée par le tissage et détissage
Caractéristiques non-fonctionnelles		Prises en charge par les interfaces de contrôle du méta-objet contrôleur et "configurables" par les alias de composants génériques	Non prises en charge par le modèle	Prises en charge par tissage/détissage des composants d'aspect et configurables

14.3 Les étapes du développement

Dans cette partie, nous allons expliquer quelles sont les étapes de développement à effectuer avec chacune des plates-formes étudiées. Nous partons

de l'hypothèse que le modèle de conception de l'application a été élaboré ainsi que la séparation des fonctionnalités. Certaines fonctionnalités sont considérées comme représentant la logique métier de cette application, d'autres la logique non-fonctionnelle. Les dépendances entre ces fonctionnalités ont été déterminées (c'est-à-dire que pour chaque opération métier nous connaissons le fil d'exécution associé).

14.3.1 Avec FRACTAL

FRACTAL est un paradigme de programmation orienté composant. Nous ne faisons donc pas de différence entre les fonctionnalités métier et celles que nous considérons comme non-fonctionnelles.
A priori, à chaque fonctionnalité nous assignons un composant. Mais il est tout à fait possible d'imaginer des composants encapsulant plusieurs fonctionnalités considérées “proches” comme nous l'avons proposé dans le modèle de composition de l'algorithme décentralisé.
D'après le modèle, nous connaissons les dépendances entre toutes les entités du système (qu'elles soient fonctionnelles ou non), nous connaissons donc pour chaque composant les services qu'il offre et qu'il requiert.
A partir de là, il est facile d'établir un schéma d'assemblage des composants et donc de déterminer les types d'interfaces FRACTAL dont chaque entité a besoin.
Il suffit ensuite de définir toutes les interfaces FRACTAL en interfaces JAVA avec les signatures des méthodes associées à chaque service. Bien évidemment, si un même composant offre plusieurs services à d'autres, il est possible de séparer les différentes interfaces de manière à rendre plus lisible au sein du code source la séparation de ces services.
Nous développons ensuite toutes les fonctionnalités sous forme de classes JAVA. Ces classes implémentent les interfaces correspondant aux services offerts par l'objet (le composant primitif) associé. Les services que requiert l'objet sont les interfaces associées sous forme d'attributs. Au sein de ces classes, à chaque fois que nous voulons déléguer un traitement à un autre composant (objet), nous nous servons de l'interface associée avec laquelle nous appelons la méthode désirée. Dans ce cas, la classe doit également implémenter l'interface `UserBindingController` de FRACTAL qui définit trois méthodes pour gérer le processus de liaison entre composants :

```
public class SiteFonctionnelImpl implements InterfaceMetier,
     UserBindingController {

     private InterfaceMoulinetteAlgo ItfMoulinetteAlgo;

     // Methodes de l'interface InterfaceMetier :
```

```
    public int lectureLocale (...) {
        ...
        ItfMoulinetteAlgo.lecture (...);
        ...
    }
    public void ecritureLocale (...) {
        ...
        ItfMoulinetteAlgo.ecriture (...);
        ...
    }

    // Methodes de l'interface UserBindingController :
    public Object getFcBindings (final String cItf) {
         if (cItf.equals(``iMA'')) {
              return ItfMoulinetteAlgo;
         }
    }
    public void addFcBinding (final String cItf, final Object sItf) {
         if (cItf.equals(``iMA'')) {
              ItfMoulinetteAlgo = (InterfaceMoulinetteAlgo)sItf;
         }
    }
    public void removeFcBinding (final String cItf, final Object sItf) {
        if (cItf.equals(``iMA'')) {
              ItfMoulinetteAlgo = null;
        }
    }
}
```

Dans cet exemple, le composant primitif implémenté par la classe `SiteFonctionnelImpl` offre un service `Metier`, il implémente donc l'interface `InterfaceMetier`. Il requiert un service `MoulinetteAlgo`, il dispose donc de l'interface associée à ce service en tant qu'attribut qui sera alors référencé par le mécanisme de liaison de FRACTAL : les trois méthodes de l'interface `UserBindingController`. Dans ces méthodes on trouve la chaîne de caractères (`IMA`) qui correspond au nom de l'interface que l'on utilisera plus tard dans l'ADL.
Il en est donc de même avec tous les composants.

Nous nous servons ensuite de l'ADL de FRACTAL pour décrire la composition de notre application. Pour chaque composant primitif nous déterminons un `type` de composant (c'est-à-dire les interfaces qu'il offre et requiert) ainsi que la classe permettant l'instanciation. Il en est de même pour les composants composites (et donc pour le composant racine de l'application) pour lesquels on spécifie les composants primitifs qu'il contient.

Pour l'exécution, il nous suffit alors d'invoquer les méthodes de l'API de FRACTAL on chargeant le *template* associé à notre composant racine.

14.3.2 Avec Kilim

KILIM est également un paradigme de programmation orienté composant. Comme dans le cas de FRACTAL, nous ne faisons donc pas de différence entre les fonctionnalités métier et non-fonctionnelles.
Comme dans le cas d'une implémentation avec FRACTAL, chaque fonctionnalité sera assignée à un composant, ou plutôt à une classe JAVA puisque les abstractions du modèle de KILIM n'interviennent pas au niveau de l'implémentation, mais au niveau de la description des *templates*. Ainsi, une application assemblée avec KILIM se développe comme une application JAVA "normale". Il suffit alors de prévoir l'assemblage des classes par l'intermédiaire des constructeurs et des méthodes *setter*. Les dépendances entre fonctionnalités seront assurées par les références des objets au sein des différentes classes concernées. KILIM est un framework d'assemblage et de configuration de classe. De manière à rendre plus lisible le code de l'application, il est possible de spécifier deux interfaces JAVA par classe. L'une pour décrire les méthodes fonctionnelles, l'autre pour décrire les méthodes de configuration des instances de la classe (mais nous n'utilisons pas cette séparation dans l'exemple ci-dessous).
Reprenons l'exemple précédent :

```
public class SiteFonctionnelImpl {

     private MoulinetteAlgoImpl mAI;

     public int lectureLocale (...) {
         ...
         mAI.lecture (...);
         ...
     }
     public void ecritureLocale (...) {
         ...
         mAI.ecriture (...);
         ...
     }

     public void setMoulinetteAlgo (MoulinetteAlgoImpl mAI) {
         this.mAI = mAI;
     }

}
```

La classe (fonctionnalité) `SiteFonctionnelImpl` requiert le service assuré par la classe `MoulinetteAlgoImpl`, elle dispose donc d'une référence sur une instance de cette dernière classe. Le sous-traitement lié à une fonctionnalité se fait simplement par une invocation de méthode. La méthode `setMoulinetteAlgo` permet d'attribuer la référence de l'instance à l'attribut `mAI`, donc d'assurer

l'assemblage de l'application.

Il faut ensuite décrire la méta représentation de l'application avec les mécanismes de KILIM (les providers, transformers, tiggers ...). A chaque classe est associé un *template* KILIM décrivant le mécanisme d'instanciation, les événements à déclencher pour assurer l'assemblage des instances. C'est donc à ce stade qu'est décrite l'application en "vision" composants.

A l'exécution, on utilise les méthodes de l'API de KILIM pour charger le *template* racine de l'application et récupérer la valeur d'un port qui déclenchera la série d'événements permettant l'instanciation et l'assemblage de l'application.

14.3.3 Avec Jac

JAC est un paradigme de programmation orienté AOP. Comme nous l'avons déjà expliqué, au sein du modèle de notre application, nous ferons la différence entre les aspects fonctionnels (implémentés par les objets de base) et les aspects non-fonctionnels (implémentés par les composants d'aspect).

Avant de commencer l'implémentation de notre application, il faut donc modéliser notre vision de la composition des aspects autour des objets de base. Il nous faut définir les points de jointure pour chaque aspect, les attributs des objets de base dont ont besoin les composants d'aspects, l'ordre dans lequel doivent être exécutés les *wrappers* de manière à correspondre à la sémantique de l'application, les éventuelles `Collaboration` entre aspects. Cette étape peut s'avérer complexe. Il peut paraître parfois difficile de transposer notre vision de la séparation de l'application vers ces concepts. En réalité, il faut définir de manière très précise le fil d'exécution des méthodes non-fonctionnelles associé à chaque opération métier.

L'implémentation des classes (objets) de base est sans difficulté puisque nous ne nous préoccupons pas à ce stade des aspects non-fonctionnels et de leurs mise en œuvre :

```
public class SiteFonctionnelImpl {

     public int lectureLocale (...) {
          ...
     }
     public void ecritureLocale (...) {
```

```
        ...
    }
}
```

Aucune trace d'éventuels aspects non-fonctionnels qui interagissent sur cette "classe de base" n'est présente dans l'implémentation.

Il faut ensuite implémenter les composants d'aspect, en commençant par définir les points de jointure que nous avons spécifiés. A chaque point de jointure, nous associons une méthode *wrappante*. L'étape suivante consiste donc à implémenter les classes `Wrapper`. Il faut penser au fait que ces classes doivent être configurées, elles peuvent avoir besoin de certaines références, d'attributs à initialiser. Ces mécanismes d'initialisation peuvent être déclenchés (par exemple) au moment de l'instanciation de l'objet de base auquel sera rattaché le *wrapper*. Il faut donc prévoir des points de jointure autour des constructeurs des objets de base.
L'implémentation des méthodes *wrappantes* se fait pas à pas : il suffit de suivre le fil d'exécution associé à une opération métier (donc dans l'ordre de l'exécution). On implémente les sections `«before»` de chaque fonctionnalité. Il est ainsi plus facile de se retrouver dans le processus de développement, de bénéficier d'une vision claire des éventuelles `Collaboration` entre aspects, de coder les tests permetant de "sauter" l'exécution de certains aspects en fonction du contexte, de se retrouver dans les éventuels aspects susceptibles de modifier la valeur de retour de l'opération fonctionnelle associée ... On procède de manière similaire concernant les implémentations des sections `«after»` de chaque fonctionnalité.

Pour l'exécution, nous spécifions les aspects à tisser autour de notre application ainsi que l'ordre d'exécution des *wrappers*, comme nous l'avons déjà expliqué.

14.4 Impact de la séparation des préoccupations

Il est évident que la vision de la séparation des préoccupations détermine la manière d'implémenter notre application, qu'elle soit développée avec des composants techniques ou avec les paradigmes de la programmation par aspects. En ce sens, deux visions de cette séparation pour une même application rendent la phase de développement complètement différente entre ces deux versions.

Dans le cycle de vie du logiciel, la séparation des préoccupations doit donc intervenir ou moment de la modélisation de l'application.

14.5 Dispersion de code

L'objectif de la séparation des préoccupations est de rendre une application le plus modulaire possible et d'éviter la dispersion du code d'une même fonctionnalité au sein de cette application. Il s'agit là bien sûr des objectifs de la programmation orientée aspects, l'isolation des fonctionnalités transverses.

Pour les plates-formes orientées composants, nous avons également cherché à isoler les fonctionnalités, en nous servant de l'assemblage de briques logicielles. Cependant, pour lier les fonctionnalités entre elles, nous sommes obligés de référencer "en dur" dans le code ces liaisons. Avec FRACTAL, nous utilisons des références d'interfaces, avec KILIM des références d'objets. Avec ces deux approches, le code d'un composant devant sous-traiter un service à un autre doit faire des appels de méthodes explicites sur ces références. Il y a donc une dépendance relativement forte entre les différentes fonctionnalités. En quelque sorte, on peut dire que nos assemblages de fonctionnalités évitent une certaine dispersion de code dans le sens où l'on modularise au maximum le code source. Cependant, avec ces approches, nous avons toujours une dispertion des invocations, ce qui ne fait que déplacer le problème.
Par exemple, si l'on veut ajouter un point de journalisation dans un composant, nous sommes obligés de modifier le code source de ce composant.

Comme nous l'avons dit, isoler les fonctionnalités transverses à une application est l'objectif de l'AOP. Nous n'avons donc pas ce genre de problème avec JAC. Un objet de base, une fois implémenté ne sera jamais modifié quels que que soient les aspects non-fonctionnels que l'on veut tisser ou détisser autour de cet objet. Le code d'un objet de base n'est en aucun cas dépendant des autres fonctionnalités (non fonctionnelles).
Pour reprendre l'exemple ci-dessus avec JAC, il suffit de définir un nouveau point de jointure et la plate-forme se chargera alors de le tisser à l'exécution.
Cependant, dans le paradigme AOP, on peut trouver de fortes dépendances entre aspects et leurs gestions peuvent être assimilables au problème que nous avons soulevé dans l'approche orienté composant. En effet, nous avons vu dans nos implémentations que nous devions parfois utiliser des `Collaborations` entre les aspects. On peut donc imaginer que dans certains cas, le fait de modifier un fil d'exécution non-fonctionnel peut entraîner une modification du code d'autres aspects adjacents (comme par exemple récupérer l'attribut stocké dans une `Collaboration` pour effectuer un nouveau traitement).

14.6 Remarque de dernière minute

Au moment où nous écrivons ces lignes, nous nous sommes rendu compte d'une petite erreur de modélisation en ce qui concerne les approches orientées composants.
En effet, dans le cadre de l'implémentation de l'algorithme décentralisé, avec FRACTAL et KILIM, nous avons été obligés de mettre en œuvre des mécanismes d'initialisation de l'application en ce qui concerne l'objet `Page` qui modélise une page mémoire de notre système. En effet, dans FRACTAL nous avons ajouté une interface **`iInitPage`** à tous les composants qui avaient besoin d'une référence sur l'instance `Page`, de même pour KILIM où dans notre méthode **`main`** nous avons récupéré les instances des composants, également pour leur fournir cette référence de notre `Page`. Il s'agit là d'un défaut de conception de l'application. En effet, dans le paradigme de ces deux plates-formes, nous aurions dû prévoir un composant encapsulant une instance de `Page` et effectuer les liaisons nécessaires au niveau de la composition pour que ce nouveau composant soit assemblé avec les autres.

Cette erreur est en fait intéressante. En effet, au moment de la conception, notre priorité était la séparation des préoccupations et leurs assemblages. Naïvement, nous n'avons donc pas pensé à intégrer un composant `Page` au sein du modèle puisque dans notre vision de l'application, cette instance ne correspondait à aucune fonctionnalité, elle n'avait donc pas sa place dans le modèle de composition.
En utilisant JAC, cela n'a pas posé de problème. Puisque notre objet métier prenait une référence de `Page` en argument de son constructeur, il nous a donc suffi de récupérer cette référence dans les composants d'aspect qui en avaient besoin par l'intermédiaire de points de jointure définis autour du constructeur. Cette anecdote a donc le mérite de mettre le doigt sur une différence de la vision de l'application entre le paradigme composant et le paradigme AOP. En orienté composant, toutes les instances ont le même "poids" au sein du modèle, c'est-à-dire sans "personnalité". En AOP, les entités de l'application ont un certain "caractère". Les objets de base et les composants d'aspects ne répondent pas du tout aux mêmes besoins. Au sein de ce paradigme, nous nous concentrons sur le fil d'exécution non-fonctionnel, sur la composition des fonctionnalités. La nécessité de faire connaître la référence de notre `Page` aux composants d'aspects est placée en second plan, de par la vision de l'application que l'on appréhende en utilisant le paradigme AOP.

14.7 La vision de l'application

Lorsque l'on travaille sur la séparation des fonctionnalités d'une application, il est très important que le développeur (ou le responsable du projet)

dispose d'une vision claire de son application, c'est-à-dire de son assemblage.

14.7.1 Avec Fractal

La vision de l'assemblage d'une application avec Fractal est très claire. En effet, les spécifications sont très poussées : définitions de type d'interface, de type de composant, liaisons de type client/serveur, spécifications de la mise en œuvre des liaisons, spécifications de gestion du contenu des composants ... etc. Chaque entité logicielle est bien définie de par les interfaces qu'elle fournit, qu'elle requiert, de par les sous-composants qu'elle encapsule (dans le cas de composants composites). Les liaisons de type client/serveur permettent de montrer le sens de la dépendance entre les composants. Dans un composant composite, l'encapsulation est très bonne, des interfaces internes au composant peuvent être totalement masquées pour l'extérieur ou au contraire exportées. L'assemblage de l'application au niveau de l'ADL (et aussi au niveau du code source puisque l'utilisation de Fractal commence au moment de l'implémentation des classes) est très intuitif. Une fois habitué au modèle de composition Fractal, il paraît simple de s'y retrouver même à la lecture d'un assemblage assuré par un autre développeur, puisque les mécanismes de composition ne s'appréhendent que d'une seule manière.

14.7.2 Avec Kilim

La vision de l'assemblage d'une application avec Kilim est ambigüe. En effet, la description des mécanismes d'instanciation de l'application peut s'effectuer de différentes manières selon la vision du développeur. Un développeur débutant n'aura pas la même façon de définir son assemblage qu'un développeur confirmé. En ce sens, il est nécessaire d'être habitué aux abstractions de Kilim pour les manipuler "proprement". Lorque l'on découvre une application, la compréhension de l'assemblage peut s'avérer fastidieuse. L'opération plug n'est pas asymétrique, son "sens" de liaison ne reflète pas forcément le sens de la dépendance entre deux composants (cela dépend sur quels composants ou *templates* les triggers sont exécutés).

14.7.3 Avec Jac

La vision de l'assemblage d'une application avec Jac est complexe. Cependant, cette complexité dépend très fortement du nombre d'aspects non-fonctionnels à gérer et de leurs éventuelles interactions. Par exemple, dans

le cadre de notre première implémentation, la composition des aspects non-fonctionnels est simple, il est assez facile d'avoir une vision cohérente de l'application et du fil d'exécution lié aux wrappers. Par contre, lorsque nous avons beaucoup d'aspects non-fonctionnels à gérer et que des dépendances fortes existent entre eux (comme dans le cadre de notre deuxième implémentation) il est plus difficile d'avoir une vision claire de la composition de ces aspects. Les dépendances entre aspects se "comprend" par rapport à l'ordre d'exécution qu'on leur donne au sein de la plate-forme. La visualisation du fil de l'exécution se fait également par l'intermédiaire de cet ordre spécifié mais aussi par les conditions qui sont traitées dans les wrappers pour tester si ceux-ci doivent être exécutés ou non en fonction du contexte. L'ensemble de ces informations peut donc rendre complexe l'appréhension de la vision de l'application.
La plate-forme JAC (ou plutôt le paradigme AOP) manque d'une méthode de spécification du fil d'exécution associée aux opérations métier (à travers laquelle on pourrait modéliser les ordres d'exécutions des wrappers, les éventuelles `Collaborations` et les tests à réaliser au début de l'exécution des méthodes wrappantes).

14.7.4 Dans une équipe de développement

Imaginons une équipe de développement composée de différentes sous-équipes chargées de développer une application. Une sous-équipe chargée du métier implémente la partie correspondant à la logique fonctionnelle de l'application, et à chaque aspect non-fonctionnel est attribué également une sous-équipe (par exemple un aspect de journalisation, de transaction et de communications distantes).

Dans un paradigme de programmation orienté composant, une équipe A qui requiert des services à d'autres fonctionnalités devra connaître les interfaces externes de ces fonctionnalités développées par une autre équipe (disons B). Les invocations de méthodes sur l'interface de l'équipe B seront effectuées par l'équipe A aux points spécifiés dans le code de A. Toutes les équipes interagissent donc entre elles en fonction des dépendances entre les fonctionnalités. Le code d'une fonctionnalité peut changer sans limite et sans conséquences pour d'autres fonctionnalités tant que les interfaces spécifiées n'évoluent pas.

Dans un paradigme de programmation orienté aspect, l'équipe chargée du métier implémente sa partie de l'application avec sa vision métier, sans se soucier des aspects non-fonctionnels. Une équipe développant un aspect non-fonctionnel n'aura pas à interagir avec l'équipe du fonctionnel mais devra connaître tout de même le code fonctionnel pour savoir où et comment tisser

les aspects. Par contre, il peut y avoir des interactions entre équipes "non-fonctionnelles", car certains aspects peuvent avoir à échanger des informations, peuvent être interdépendants. Une modification du code métier peut éventuellement provoquer une modification de code non-fonctionnel. Un concepteur devra également surveiller la cohérence de la sémantique globale de la composition des aspects de l'application.

14.8 Nombre de lignes de code

Comparer nos différentes implémentations/plates-formes en fonction du nombre de lignes de code ne paraît pas très formel. En effet, l'écriture d'un code source dépend de nombreux facteurs tels le niveau du programmeur, ses habitudes d'écriture ...
Nous proposons donc cette comparaison "par curiosité" et non comme un argument de comparaison entre les plates-formes. Néanmoins, toutes les implémentations ont été effectuées par la même personne, donc avec les mêmes habitudes d'écriture.
De plus, les ADL de Fractal et Kilim utilisent tous deux une syntaxe XML. Nous n'avons pas considéré les fichiers de configuration de Jac qui regroupent une quantité d'information négligeable par rapport à l'implémentation.

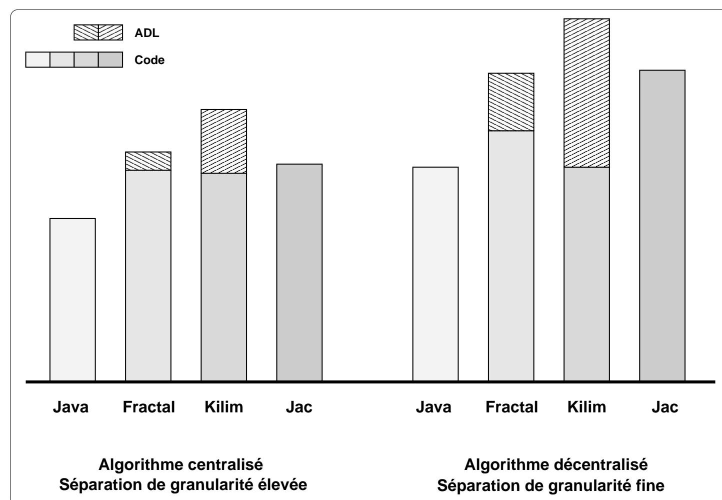

Fig. 37 – Quantité de lignes de code en fonction de la plate-forme et de la séparation des fonctionnalités

Les différences quantifiées sur la figure entre les deux implémentations concernent la séparation des préoccupations dont la granularité a été volontairement modifiée. En effet, le code relatif aux deux algorithmes entre la version

centralisée et la version décentralisée est à peu près similaire.

Nous pouvons tout de même tirer quelques informations de cette comparaison :

- L'implémentation qui demande le moins de code est la version JAVA de l'algorithme centralisé. En effet, dans ce cas, comme nous l'avons vu, aucune séparation des préoccupations n'a été effectuée, le programme a été écrit de manière complètement monolithique.
- Dans le cadre de la première implémentation, nous avons implémenté notre première vision de notre séparation avec les trois plates-formes. Le code engendré est donc plus élevé qu'avec la version JAVA monolithique. Cela s'explique en partie par le fait que la fragmentation des classes engendre plus de code qu'il faut alors "reconstituer" : en utilisant les interfaces FRACTAL, les méthodes *setter/getter* avec KILIM et les composants d'aspects avec JAC. Cette remarque s'applique également (de manière plus globale) lorsque l'on compare le code engendré par une séparation des préoccupations de granularité fine (dans le cas de l'algorithme décentralisé) plus important que celui engendré par une séparation de granularité élevée (dans le cas de l'algorithme centralisé).
- Dans le cadre de la deuxième implémentation, la quantité de code JAVA est identique à celle de KILIM. En effet dans l'implémentation en JAVA "pur", nous avons programmé avec notre vision de la séparation des préoccupations, il s'agit donc du même code que celui utilisé avec KILIM.
- Toujours dans cette deuxième implémentation, on voit que le code FRACTAL et JAC est plus élevé que celui de KILIM. Cela est logique : l'implémentation avec KILIM est du JAVA "pur". La gestion des interfaces FRACTAL et des wrappers JAC a bien évidemment un coût en terme de code.
- La deuxième implémentation avec JAC est la plus coûteuse. En effet, une séparation des préoccupations fine engendre beaucoup de traitements supplémentaires lorsque l'on programme en orienté AOP. De plus, toute la "reconstruction" de l'application se fait en JAVA.
- On voit bien que la définition des assemblages des composants techniques au niveau des ADL demande plus d'effort avec KILIM qu'avec FRACTAL. Cela est tout à fait compréhensible dans le sens où l'on écrit une application KILIM "normalement", en JAVA "classique". Les abstractions énoncées dans KILIM n'interviennent qu'au niveau de la description des *templates* et non au niveau du code. Nous avons donc plus d'informations à fournir au niveau de l'ADL pour fournir à KILIM les mécanismes d'instanciation de l'application (tels les providers, les transformers...). En ce qui concerne l'ADL de FRACTAL, la description de l'assemblage est plus simple, nous nous contentons de donner les liaisons entre les interfaces clientes et serveurs FRACTAL et les implémentations des com-

posants. Nous ne gérons que des noms d'interfaces dans l'ADL, leur "réelle" gestion se fait dans le code (ce qui explique d'ailleurs le surplus de code FRACTAL par rapport au code KILIM.

Bien évidemment, ces conclusions sont à relativiser pour les raisons déjà énoncées mais aussi en ce qui concerne la description des assemblages en niveau des ADL. En effet, il existe des mécanismes d'héritages dans les ADL de FRACTAL et KILIM qui, selon leurs utilisations peuvent avoir un impact important sur la quantité d'informations que stockent les fichiers de description.

Néanmoins, cette petite comparaison nous permet de mettre le doigt sur certains points qui différencient les différents paradigmes de programmation et modèles de composition que nous avons utilisés.

14.9 Performances

Il est évident que séparer les fonctionnalités d'une application a un coût en terme de performances à l'exécution.
En ce qui concerne les composants techniques, nous fragmentons l'application en différents objets qui communiquent alors par l'intermédiaire d'invocations de méthodes. Le fil d'exécution de l'application est donc réparti sur l'ensemble des composants, ralentissant les performances.
Dans KILIM, on ne manipule que les instances des objets de l'application, le fil d'exécution ne concernera donc que ces instances. Cependant, dans FRACTAL, certains objets intercepteurs sont parfois instanciés par la plate-forme (comme par exemple au niveau de la membrane d'un composant composite pour rediriger le fil d'exécution sur le bon composant) augmentant encore le nombre d'indirections.
Au sein de JAC demeurent également de nombreuses indirections dûes à la gestion de la chaîne wrappante, c'est-à-dire la chaîne des méthodes wrappantes qui entourent une méthode d'un objet de base.

Bien évidemment, plus la granularité choisie pour séparer les fonctionnalités est fine, plus les indirections sont importantes au sein du fil d'exécution, causant une baisse d'autant plus importante des performances.

Dans le cadre de nos différentes implémentations, nous avons tenté de quantifier les pertes de performances sur un défaut en lecture et un défaut en écriture. Cependant, les résultats n'ont pas été concluants dans le sens où parfois, certaines petites optimisations au niveau de l'algorithme ont été effectuées selon les plates-formes. Il a donc été rendu difficile d'apprécier à sa juste valeur

les diminutions de performances engendrées uniquement par ces indirections.

Intuitivement, nous pouvons tout de même supposer qu'une application développée avec KILIM sera la plus performante puisqu'elle manipule directement les instances des objets associés aux composants. Puis viendrait l'application développée avec JULIA (implémentation de FRACTAL). Il est d'ailleurs possible de spécifier une composition très statique de l'application, limitant sa capacité à être reconfigurée dynamiquement mais diminuant les indirections et donc augmentant les performances. Puis viendrait l'application développée avec JAC où les indirections font diminuer fortement les performances [PDF+02b] (surtout lorsque l'on tisse de nombreux aspects autour d'une même méthode de base comme nous l'avons fait dans le cadre de la deuxième implémentation).

15 Conclusion

Dans le cadre de notre travail, nous avons présenté trois approches basées sur l'assemblage de composants logiciels, chacune fondée sur la séparation des préoccupations. Chacune d'entre elles présente de réelles avancées par rapport au paradigme de programmation orientée objet, et représente un pas en avant vers plus de modularité, de factorisation de code et de réutilisabilité.
Néanmoins, ce gain de modularité que nous avons cherché à mettre en avant est directement dépendant du principe de séparation des fonctionnalités appliqué à l'étape de modélisation d'une application, et donc indépendant d'une plate-forme d'accueil et nous avons montré que de nombreux obstacles sont susceptibles de rendre cette phase complexe.

D'un point de vue plus pragmatique, nous pouvons dire que les spécifications du projet FRACTAL semblent donner une vision assez satisfaisante, concise, générique et mûre d'un modèle de composant et de la problématique d'assemblage de ceux-ci. Certains mécanismes liés à la configurabilité mis en œuvre dans le modèle de KILIM permettent d'appréhender, avec un regard de haut niveau, la problématique de paramétrisation d'applications. Enfin, la séparation de la logique système et de la logique métier, telle qu'elle est énoncée dans le paradigme de programmation orientée aspect, et en particulier dans JAC grâce à la mise en œuvre des composants d'aspect, est essentielle en terme d'adaptabilité et de modularité. Ces trois modèles étudiés, par complémentarité, semblent représenter une vision exhaustive des besoins en terme de construction d'applications réparties et il paraît indéniable, qu'à terme, ces trois approches nécessitent d'être fusionnées au sein d'un modèle commun pour servir de base aux développements des middlewares adaptatifs qui seront la base des architectures réparties futures.

Table des figures

1 Encapsulation d'un objet 22

2 Encapsulation d'une méthode 23

3 Modèle de composant - sous-composant - composants partagés dans FRACTAL 31

4 Composants et objets primitifs dans FRACTAL 31

5 Visibilité des interfaces des composants dans FRACTAL 32

6 Deux composants primitifs dans un composant composite dans FRACTAL . . . 39

7 Modélisation de la composition de notre application avec KILIM 48

8 Schématisation de l'architecture de KILIM 50

9 Etapes de la programmation par aspects 53

10 Modélisation de la composition de l'objet de base et du composant d'aspect dans le cadre de notre exemple avec JAC 59

11 Schématisation de la vue d'ensemble de notre application JAC 61

12 Schématisation de l'architecture de JAC 62

13 Défaut en lecture effectué sur le site L dans le cadre de l'algorithme centralisé 71

14 Défaut en écriture effectué par le site E dans le cadre de l'algorithme centralisé 71

15 Comunications asynchrones (a) et synchrone (RPC de RMI) (b) dans le cas d'un défaut en lecture (algorithme centralisé) 76

16 Ensemble des communications RMI dans le cadre d'un défaut en écriture (algorithme centralisé) 77

17 Schématisation de la séparation des préoccupations de notre application (côté client) de l'algorithme centralisé 80

18 Modélisation de l'application implémentée en JAVA pur (algorithme centralisé) 82

19 Modélisation de l'implémentation de l'application avec FRACTAL 84

20 Modélisation de la composition avec FRACTAL 85

21 Le modèle de composition de notre prototype avec KILIM 89

22 Schématisation de certains mécanismes mis en jeu dans l'instanciation de l'application au niveau du modèle KILIM 91

23 Une petite modification du modèle d'implémentation de notre prototype avec la plate-forme JAC 92

24 Le modèle de composition des aspects avec JAC 92

25 Graphe des propriétaires probables dans le cas d'un défaut en lecture initié par le site 6 dans le cadre de l'algorithme décentralisé 100

26 Graphe des propriétaires probables dans le cas d'un défaut en écriture initié par le site 6 dans le cadre de l'algorithme décentralisé 100

27 Propagation des requêtes de demande d'invalidation initiée par le site 2 dans le cadre le l'algorithme décentralisé 100

28 Schématisation de la séparation des préoccupations au sein de l'algorithme décentralisé . . . 104

29 Modélisation de l'implémentation (et de notre séparation des préoccupations) de l'algorithme décentralisé . . . 106

30 Le contexte du site 1 et du site 2 pour illustrer notre exemple . . . 107

31 Modéle d'assemblage générique avec les composants techniques (algorithme décentralisé) . . . 109

32 Modèle de composition lié à l'algorithme décentralisé avec FRACTAL . . . 111

33 Modèle de composition lié à l'algorithme décentralisé avec KILIM . . . 112

34 Le modèle de composition des aspects de l'algorithme décentralisé avec JAC . 114

35 Schématisation du fil d'exécution des wrappers lors de la réception de notre requête RMI . . . 117

36 Schématisation des différents fils d'exécution des wrappers lors d'un défaut en écriture . . . 118

37 Quantité de lignes de code en fonction de la plate-forme et de la séparation des fonctionnalités . . . 144

Glossaire

ADL : Acronyme de Architecture Description Language, langage de description d'architecture. *page 46*

AOP : Acronyme de *Aspect Oriented Programming. page 52*

Aspect : Propriété transverse à une application. *page 52*

CCM : Acronyme de Corba Component Model, le modèle de composant proposé par l'OMG (Object Management Group). *page 16*

Conteneur : Structure d'hébergement et d'exécution de composants. *page 12*

Framework : Environnement de développement (litérallement : cadre de travail) (cf. 4.1 page 19). *page 44*

Intercepteur : Permet de capturer un événement précis afin d'organiser la passage du niveau de base au niveau méta. *page 22*

Introspection : Possibilité pour un système de pouvoir interroger sa structure interne (code, données) ou les paramètres de son environnement d'exécution. *page 15*

MOP : Acronyme de *Meta Object Protocol*, interface entre le programme et le niveau de base d'un système d'un système réflexif. Il offre la possibilité aux utilisateurs de modifier le comportement du système au cours de l'exécution. *page 22*

POO : Programmation Orientée Objet. *page 13*

SoC : Acronyme de *Separation of Concerns* - Séparation des préoccupations. Principe qui consiste à séparer les définitions d'entités (concerns) distinctes d'une même application. *page 17*

Index

étapes de développement, 134
évolutivité, **125**
FRACTAL, **28**
JAC, **52**
KAI LI et PAUL HUDAK, **68**, 96
KILIM, **44**

abstraction, **13**, 128
ADL, **42**, 46
AOP, **52**, 54
architecture, 43, 50, 61
aspect, **52**

binding, liaison, **33**

canevas, 19
cellule, 29
cohérence, 70, 96, 125
composabilité, **14**
composant, **12**
composant composite, **37**
composant primitif, **37**
composition répartie, **124**
contrôleur, **33**
copyset, 68

design pattern, **20**
dispersion de code, 140

encapsulation, **13**, 128

fil d'exécution, **105**
framework, **19**

identité, **13**, 128
intercepteur, **21**

Julia, **28**, 43

lignes de code, 144

méta-objet, **21**
méthode wrappante, 55
membrane, 29
modélisation, **81**, 105
modèle de composant, **12**
MOP, 22

objet de base, 56

performances, 146
plasme, 29
points de jointure, 53
ports, **45**
programmation par aspects, **52**
protocole, 68, 97
prototype, 64, 74
provider, **45**

RMI, **75**, 77
RPC, **76**

séparation des préoccupations, **17**, 78, 102
slot, **45**
SoC, **17**

template, 46
transformer, **45**

vision de l'application, 141

Références

[BCS02] E. BRUNETON, T. COUPAYE, et J.B. STEFANI. The fractal composition framework. 2002. Spécifications de l'interface Fractal. http ://fractal.objectweb.org/current/Fractal1.0-0.pdf.

[Bri02] Jean-Pierre BRIOT. Adaptation logicielle : Réflexion, composants, agents, 2002. Cours magistral dans le cadre du tronc commun du DEA SIR.

[Chi00] Shigeru CHIBA. Load-time structural reflection in java. 2000.

[Del03] Fabien DELPIANO. The kilim2 framework, 2003. Cours de présentation de Kilim, DEA SIR.

[FAT03] Fractal adl tutorial, 2003. Tutoriel sur l'ADL de Fractal. http ://fractal.objectweb.org/current/doc/tutorials/adl/index.html.

[Fra] Fractal rmi documentation. API du projet Fractal RMI. http ://fractal.objectweb.org/current/doc/javadoc/fractal-rmi/index.html.

[FT002] Fractal tutorial, 2002. Tutoriel Fractal. http ://fractal.objectweb.org/current/doc/tutorials/fractal/index.html.

[GS02] Eric GRESSIER-SOUDAN. Cohérence des données réparties partagées, 2002. Cours du tronc commun / DEA SIR.

[HD03] François HORN et Fabien DELPIANO. Kilim2 tutorial, 2003.

[Hor03] François HORN. Kilim2 : configuration and component framework, avril 2003. Objectweb Architecture Meeting, Université de Jussieu.

[JUT02] Julia tutorial, 2002. Tutoriel sur le framework Julia. http ://fractal.objectweb.org/current/doc/javadoc/julia/index.html.

[LH86] Kai LI et Paul HUDAK. Memory coherence in shared virtual memory systems. pages 229–239, 1986. Symposium on Principles of Distributed Computing (PODC).

[MP02] Raphaël MARVIE et Marie-Claude PELLEGRINI. Modèles de composants, un état de l'art. *L'objet*, 2002. P. 61 à 89.

[Paw] R. PAWLAK. Jac programmer's guide. Guide de programmation pour JAC - http ://jac.aopsys.com/doc/programmer_guide.html.

[Paw02] Renaud PAWLAK. La programmation par aspects interactionnelle pour la construction d'applications à préoccupations multiples, 2002. Thèse de doctorat.

[PDF+02a] R. PAWLAK, L. DUCHIEN, G. FLORIN, F. Legond-Aubry aned L. SEINTURIER, et L. MARTELLI. A uml notation for aspect-oriented software design. 2002. http ://jac.aopsys.com/papers/uml/uml.html.

[PDF+02b] R Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, et L. Martelli. Jac : An aspect-based distributed dynamic framework. 2002.

[PDF+02c] R. Pawlak, L. Duchien, G. Florin, L. Martelli, et L. Seinturier. Une approche pour la programmation répartie : les composant d'aspect. *L'objet - Coopération dans les systèmes à objets*, 8 :39 à 59, 2002.

[PDFS01] R. Pawlak, L. Duchien, G. Florin, et L. Seinturier. Dynamic wrappers : Handling the composition issus with jac. 2001. TOOLS USA 2001.

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien, et G. Florin. Jac : A flexible solution for aspect-oriented programming in java. September 2001. Conférence Reflection 2001.

[Sou02] Jean-Marc Soucé. Modèle de spécification et d'assemblage de composants logiciels. Rapport technique, CNAM, 2002. Mémoire de diplôme d'ingénieur du CNAM.

[Teb02] Luc Teboul. Méta-programmation, réfléxion et programmation par aspect, 2002. Rapport bibliographique / stage DEA SIR.

[TUT02] Jonathan tutorial, 2002. Tutoriel de la plate-forme Jonathan, http ://jonathan.objectweb.org/doc/tutorial/index.html.