Laboratoire d'Informatique Fondamentale de Lille

Un framework pour les composants logiciels distribués et parallèles

THÈSE

présentée et soutenue publiquement le 20 décembre 2006

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Iyad AL SHABANI

Composition du jury

Président :	Lionel SEINTURIER	Université de Lille1
Rapporteurs :	Jean-Christophe LAPAYRE	Université de Franche-Comté
	Jean-Louis PAZAT	INSA de Rennes
Examinateur :	Jaafar GABER	Université de Belfort-Montbéliard
Co-encadrant	Richard OLEJNIK	Université de Lille1
Directeur :	Bernard TOURSEL	Université de Lille1

Université des Sciences et Technologies de Lille - Lille 1

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible, par leurs soutiens, leurs aides et leurs contributions. J'aimerais remercier en particulier :

Lionel Seinturier, Professeur à l'Université des Sciences et Technologies de Lille, pour m'avoir fait l'honneur de présider le jury de thèse.

Jean-Christophe Lapayre, Professeur à l'Université de Franche-comté et Jean-Louis Pazat, Professeur à l'INSA Rennes pour avoir accepté de juger mon travail de thèse.

Jaafar Gaber, Maître de conférence à l'Université de Belfort-Montbéliard pour avoir accepté de faire partie de mon jury de thèse.

Bernard Toursel, Professeur à l'Université des Sciences et Technologies de Lille pour m'avoir accueilli au sein de l'équipe *PALOMA* et pour la confiance qu'il m'a accordée. Ses précieux conseils et son aide ont été déterminants pour la réalisation de cette thèse.

Richard Olejnik, Ingénieur de Recherche CNRS au LIFL pour m'avoir soutenu pendant ces années d'études. Ses multiples conseils, encouragements et sa gentillesse m'ont souvent aidé à traverser des périodes difficiles et à surmonter les problèmes qui se sont présentés.

Je tiens, par ailleurs, à exprimer ma gratitude à tous mes amis ; tous ceux qui m'ont accompagné, qui m'ont soutenu et qui m'ont encouragé durant ces années d'études. Je remercie plus particulièrement Ammar, Ahmad et Rachid pour leur gentillesse et leur soutien.

Un remerciement spécial à Sonia pour sa gentillesse et son encouragement dans les moments difficiles et pour avoir lu avec un regard extérieur certains chapitres.

Enfin, je remercie plus particulièrement ma famille à commencer par mon frère Ayman qui a partagé trois ans avec moi en France, mes frères, mes sœurs et mes parents, pour leur aide et encouragement malgré la distance qui nous sépare. Leurs lettres, e-mails, appels téléphoniques, ne manquaient pas et m'ont été d'un précieux secours. Je ne peux mesurer la dette que j'ai envers eux conscient des sacrifices qu'ils ont fait pour m'offrir l'opportunité de faire des études.

Iyad

Je dédie cette thèse
à mes parents

Table des matières

Table des figures

Introduction générale

La montée en puissance régulière des équipements de communication et de traitement de données rend possible le développement d'applications de calcul distribué à grande échelle. Il devient envisageable de déployer une application exploitant des ressources de calcul disséminées sur des grands réseaux inter-connectés. Ces ressources pouvant être elles-mêmes des architectures parallèles et/ou des systèmes distribués.

L'état d'avancement dans ces technologies intéresse le monde du calcul parallèle et distribué. Les forces conductrices dans ce domaine sont : les avancées technologiques dans le monde du calcul, la disponibilité des réseaux très rapides et les efforts de recherche très intéressants dirigés vers le développement de logiciels qui supportent les environnements de programmation pour le calcul parallèle et/ou distribué. De plus, avec la demande croissante de puissance de calcul et la diversité de demandes de calcul, il est clair qu'il n'existe pas une seule plate-forme qui réponde à l'ensemble des besoins. Par conséquent, les environnements de calculs ont besoin de mettre l'accent sur l'utilisation effective des ressources informatiques hétérogènes. Seuls les systèmes parallèles et distribués offrent un potentiel pour effectuer une telle intégration de ressources et de technologies en gardant la flexibilité désirée lors de l'utilisation. Cependant, la réalisation de ce potentiel demande une avancée technique sur plusieurs fronts : la technologie de traitement, la technologie des réseaux et les environnements et les outils logiciels.

1 La problématique

Le développement des applications parallèles et distribuées n'est pas un processus trivial et demande une compréhension totale de l'application et de l'architecture. La complexité de ce type d'applications est toujours plus importante. Cependant, les approches classiques de programmation rendent relativement difficiles des propriétés comme l'évolutivité, la réutilisation, la maintenance et la qualité dans les gros logiciels. Dans le monde du calcul parallèle et distribué, le besoin en capacité pour le calcul à haute performance et le parallélisme intensif ajoute des difficultés pour gérer la complexité des logiciels. Le respect des applications existantes est l'une des principales considérations des réalisateurs de logiciels pour le calcul performant. Une des solutions est de réécrire les applications avec de nouvelles technologies. Mais tout cela conduit à un cycle de ré implémentation au fur et à mesure que de nouvelles technologies apparaissent.

Diverses technologies sont apparues pour permettre une meilleure construction des applications. La programmation orienté-objets a donné certaines réponses sur ces aspects mais elles n'étaient pas suffisantes. En effet, l'utilisation des modèles objets dans la pro-

grammation distribuée a été motivée par la modélisation et la facilité de conception des applications ainsi que la réutilisation du code. Mais, dans la programmation orienté-objets, l'objet n'est pas totalement encapsulé. Il est vrai que l'objet est manipulé par des méthodes mais son corps (code) n'est pas tout a fait masqué. La réutilisation de l'objet se fait au niveau du langage et non au niveau du binaire. L'approche composants est la continuité de l'approche orienté-objets. Elle vient répondre plus clairement à des aspects comme la réutilisabilité et l'encapsulation de morceaux de logiciels. Les implémentations de composants sont réutilisables sous forme binaire. La manipulation du composant se fait au travers de ses interfaces. Plusieurs termes comme architecture logicielle à base de composants ou framework de composants ont été introduits et utilisés en relation avec la construction des applications à base de composants. L'évolution du développement à base de composants a été poussée par la demande croissante de nouvelles technologies par l'industrie du logiciels pour contrôler le coût de développement des produits logiciels et améliorer leurs qualités. Dans le monde du développement à base de composants l'accent n'est pas seulement mis sur les aspects techniques et la disponibilité de la construction mais il est aussi mis sur la garantie de la qualité de ses composants. Cette qualité des composants est liée à la définition des règles architecturales et à la définition des contrats sur les composants. Les règles décrivent la manière de construire des applications par composition de composants logiciels et comment les composants sont fabriqués. Les architectures logicielles à base de composants les plus répandues sont les architectures conduites par un modèle (MDA).

Les aspects cités ci-dessus doivent être standardisés pour permettre plus d'interopérabilité entre les applications qui sont écrites en utilisant certains standards. Plusieurs questions se posent : comment les interfaces sont-elles spécifiées ? Comment les références d'objets sont-elles traitées ? Comment les services sont-ils placés et fournis ? Comment l'évolution du composant est-elle traitée ? Toutes ces questions ont amené à définir des modèles de composants pour l'interopérabilité, la connexion et la construction des applications à base de composants. Dans cette perspective, des modèles de composants ont été définis et standardisés. Les grands industriels du logiciel, comme SUN et MicroSoft, et des institutions, comme l'OMG, ont défini des standards et des modèles selon les besoins communs de l'industrie du logiciel et des besoins spécifiques dont chaque standard essaye d'offrir une solution. Les EJB de SUN, les DCOM/COM+ de Microsoft et le CORBA CCM de l'OMG sont les standards de composants les plus répandus dans l'industrie. Les universités, les laboratoires de recherche ainsi que des institutions spécialisées ont défini des modèles plus au moins adaptées aux besoins spécifiques de la recherche. Dans le monde du calcul parallèle et distribué à haute performance, plusieurs modèles de composants ont été définis ou dérivés des modèles de l'industrie afin d'offrir des plates-formes ou des environnements spécifiques pour le calcul parallèle et distribué. Des projets de recherche comme GridCCM, Proactive ou CCA sont de bons exemples. Le premier étend le modèle CCM, le deuxième implémente un modèle de recherche (Fractal) défini par FranceTelecom et le troisième est une nouvelle définition de standard spécifique qui vise le calcul scientifique à haute performance.

Notre contribution se situe dans le domaine des applications Java parallèle/distribuées. Le projet *DG-ADAJ (Desktop-Grid Adaptive Distributed Applications in Java)* vise à fournir des réponses aux problèmes de conception et d'exécution efficaces des trai-

tements répartis sur des réseaux. Le langage Java offre des caractéristiques importantes pour la gestion de la distribution, de l'hétérogénéité et de la sécurité. DG-ADAJ offre un système d'observation d'objets répartis et des machines. Cette contribution met l'accent sur le développement des applications à base de composants logiciels indépendants et réutilisable. Dans ce but nous avons intégré un framework de composants dans la plateforme DG-ADAJ pour permettre aux applications conçues avec DG-ADAJ de profiter de l'apport de la technologie composants en gardant une efficacité de développement et d'exécution. Cela permettra aussi à d'autres applications de profiter des caractéristiques de la plat-forme DG-ADAJ en terme d'observation et de régulation de charge. Nous souhaitons aussi rendre plus facile la construction des composants et des applications à base de composants dans un environnement parallèle et distribué. Certes, la construction des applications en utilisant un framework de composants a des avantages au niveau de la construction mais il y a un coût à cela. Ce dernier est plus important quand il s'agit d'une application parallèle et distribuée. Pour cela, dans la construction du framework, il faut choisir le modèle et l'architecture logicielle adéquates pour les applications dans ce type d'environnement.

Le framework proposé est basé sur le modèle *CCA (Common Component Architecture)* : un standard dédié aux applications à base de composants dans le monde du calcul parallèle et distribué. CCA promet une efficacité plus importante que les standards industriels avec un framework qui a un surcoût négligeable. Nous avons implémenté le noyau CCA dans notre framework et l'avons intégré dans la plate-forme DG-ADAJ. Nous étendons le modèle de base de CCA pour prendre en compte une construction des composants à base d'autres composants. De plus, nous offrons des composants spéciaux pour aider à la construction des applications parallèles.

2 Organisation du mémoire

Ce mémoire est articulé suivant cinq chapitres. Le premier traite du développement à base de composants et des différentes technologies proposées par l'industrie de logiciels. Le deuxième concerne les environnements pour les applications parallèles et distribuées. Il cite quelques propositions du monde de la recherche dans le domaine du calcul parallèle et distribué. Le troisième présente la plate-forme DG-ADAJ et introduit le framework de composants. Le quatrième chapitre donne des propositions de composants spéciaux pour aider à la construction des applications parallèles et distribuées. Le cinquième traite des questions de performance du framework et explique quelques expérimentations autour de notre proposition.

- Dans le premier chapitre, nous présentons la problématique générale du développement des applications distribuées/parallèles ainsi que les technologies utilisées pour rendre cette tâche plus facile pour utiliser les architectures matérielles utilisées. Nous présentons la technologie héritant de la technologie objets distribués qui est la technologie composants. Dans la suite, nous présentons les notions spécifiques au développement basé sur des composants. Nous donnons des définitions du composant et de l'architecture logicielle à base de composants et les termes liés à ces définitions. Nous présentons par la suite les trois standards industriels qui sont *JB/EJB*,

DCOM/COM+/.NET et *CORBA CCM*. A la fin de ce chapitre nous faisons une comparaison entre ces trois standards.

- Le deuxième chapitre concerne les environnements de développement et d'exécution des applications parallèles/distribuées. Dans ce chapitre nous présentons quelques plates-formes de développement et d'exécution pour des applications parallèles et distribuées. Ces environnements sont des environnements spécifiques qui ont été créés afin de répondre aux besoins du développement des applications du calcul parallèle/distribué. La spécificité de ce type d'applications demande une extension ou une adaptation des standards industriels ou simplement une définition des standards spéciaux à ce type d'application. Nous avons présenté trois projets GridCCM, ProActive et CCA. Nous avons souligné les points intéressants dans chacune des propositions et les différences entre elles.
- Dans le troisième chapitre, nous décrivons la plate-forme DG-ADAJ et notre proposition, le framework CCADAJ. Nous expliquons l'implémentation de ce framework en utilisant les spécifications de l'architecture logicielle CCA. Nous détaillons aussi l'apport du framework CCADAJ par rapport au modèle CCA, notamment dans la composition hiérarchique par la définition du super-composant. Un super-composant permet d'encapsuler plusieurs composants de base dans un seul composant. Ensuite, nous décrivons l'intégration de ce framework dans la plate-forme DG-ADAG en utilisant des composants distants et la définition des ports distants et la généricité. Les composants distants sont des composants déployables dans un environnement DG-ADAJ sur les machines participantes à cet environnement.
- Le quatrième chapitre traite des composants de contrôle pour aider le développeur d'une application parallèle. Ce développeur utilise certaines structures encapsulées en forme de composants pour contrôler le parallélisme de l'application et des données. Nous décrivons des composants de parallélisation comme le distributeur, le collecteur et le pipeline. Le distributeur est un composant avec un port (interface) multiple qui permet à chaque connexion d'offrir une donnée à un composant de calcul. Ainsi il distribue un ensemble de données aux composants de calcul. Les ports donc sont créés dynamiquement à chaque connexion avec un composant de calcul. Le collecteur a le même principe, mais, il collecte les données offertes par les composants de calcul. Le pipeline est un composant qui est composé de plusieurs étages et qui envoie les données en mode pipeline aux composants de calcul. Ces composants de contrôle profitent de la spécificité du framework CCADAJ en terme de dynamicité de la création des ports et des composants.
- Le cinquième chapitre décrit les questions de performance d'une architecture basée sur le modèle CCA. Nous présentons certaines mesures d'évaluation concernant l'utilisation des composants dans l'environnement CCADAJ par rapport à l'utilisation des objets. Nous expliquons aussi d'où provient le surcoût de framework et de différentes structures présentées comme le super-composant et le composant distant. Ensuite, nous donnons quelques mesures concernant la comparaison d'une exécution d'une application parallèle distribuée qui traite le problème TSP. Nous comparons deux implémentations de cette application : une implémentation sans utilisation de framework de composants et une implémentation en utilisant des composants dans CCADAJ. Les mesures montrent le surcoût de l'utilisation de CCADAJ.

Chapitre 1

Les systèmes distribués : des objets aux composants

Les environnements informatiques sont de plus en plus fréquemment constitués d'un ensemble de processeurs collaborant pour fournir un service en commun. Les machines multiprocesseurs (machines parallèles) fournissent localement la puissance nécessaire aux applications demandant de gros moyens de calcul. Les réseaux de machines (systèmes distribués) permettent de faire coopérer des machines distantes pour l'exécution d'applications faisant intervenir des ressources géographiquement dispersées. L'émergence de nouvelles technologies (objets répartis, "middleware"), de nouvelles applications (calcul scientifique parallèle, travail coopératif, téléconférence, multimédia,...) et de nouveaux supports (Web, réseaux à très haut débit, mobiles) expliquent cette évolution.

1.1 Les technologies des systèmes distribués

Les systèmes distribués peuvent être définis de plusieurs façons [TS02]. Cependant, un système distribué peut être caractérisé par la mise en interaction les ressources informatiques matérielles et logicielles indépendantes reliées par un réseau de communication, afin de donner à l'utilisateur l'illusion d'un seul système. La construction des systèmes distribués est faite pour partager des ressources matérielles comme le stockage de données et les ressources de calcul ou des ressources logicielles telles que des applications Internet ou de bases de données.

Les systèmes distribués actuels sont généralement construits à partir d'une couche logicielle intermédiaire, que l'on appelle *intergiciel* ou *Middleware*, au dessus du système d'exploitation réseau pour masquer l'hétérogénéité du système et rendre plus transparent les accès aux services distants. Ces systèmes sont construits en général en se basant sur un modèle spécifique (appel de procédures distantes, notification d'événement, objets distribués, documents distribués ...etc).

La construction des systèmes distribués pose de nouveaux défis :

L'hétérogénéité : peut être vu selon plusieurs points de vue : matériel, systèmes d'exploitation, langages de programmation des logiciels réseaux, comme TCP/IP, qui permettent de masquer l'hétérogénéité des réseaux, et les middlewares gèrent les

autres aspects de cette hétérogénéité.

La transparence : un système d'exploitation doit rendre l'accès aux ressources réparties transparent, en déchargeant les applications de certains aspects (accès, représentation des données, localisation).

L'extensibilité : elle permet de faciliter l'intégration de nouvelles entités dans le système sans affecter ce dernier. Une des approches les plus utilisées est celle de la séparation entre les interfaces des services et leur implémentation.

La sécurité : la sécurité est un point important dans un système distribué. Elle permet d'assurer le bon fonctionnement des applications et de respecter les droits sur les ressources disponibles. Par exemple elle doit :
- assurer l'intégrité des données ;
- gérer les droits d'accès et la confidentialité des utilisateurs ;
- gérer l'accès aux ressources de façon sécurisée.

La tolérance aux pannes : l'une des caractéristiques des systèmes distribués est qu'il sont fortement exposés aux pannes. L'un des défis des concepteurs des systèmes distribués est de mettre en œuvre des mécanismes qui permettent selon la qualité de service requise de gérer au mieux les éventuels incidents qui risquent de se produire (indisponibilité d'une ressource de calcul, échec d'une communication, endommagement d'un fichier, etc.)

Le passage à l'échelle : l'un des principaux problèmes qui se posent lorsqu'un système distribué évolue (en nombre de ressources) est la dégradation des performances. Cette dégradation est dûe à plusieurs paramètres tels les coûts des communications (surtout si les sites sont géographiquement éloignés), ou les accès aux services du système.

La gestion des accès concurrents : ce problème déjà présent dans les applications parallèles ou multithreadées doit aussi être géré dans les systèmes distribués. Les accès concurrents ne concernent plus seulement les accès aux applications mais aussi aux ressources du système.

La construction d'un système distribué met l'accent sur les aspects architecturaux, sur les services et sur les technologies candidates pour implémenter les parties essentielles d'un système distribué. Généralement parlant, le processus de conception d'un système distribué implique trois activités principales :

- la conception d'un système de communication qui permet l'échange de l'information entre les différentes ressources du système distribué.
- la définition de la structure de système (l'architecture) et les services du système qui permettent à un ensemble d'ordinateurs de fonctionner comme un seul système plutôt qu'une collection de systèmes
- la définition de techniques de programmation distribuée pour développer des applications parallèles/distribués.

En se basant sur ces trois notions de processus de conception, la conception d'un système distribué peut être décrit en trois couches :

- La couche *architecture du réseau* : Elle comprend les différentes composantes du système de communication, le type de réseau, les protocoles de communication et les interfaces réseaux.

- la couche *architecture du système* et services : Elle représente le système distribué du point de vue du concepteur et le gérant du système. Elle définit les structure et les services du système. Elle doit être supporté par le système distribué pour fournir une image seule du système *Single System Image*.
- la couche du *paradigme de programmation distribuée*. Cette couche représente le système distribué du point de vue du programmeur (utilisateur). Dans cette couche l'accent est mis sur les méthodologies de programmation qui peuvent être utilisées pour développer des applications distribuées/parallèles.

Le modèle client/serveur est le paradigme de programmation le plus répandu dans les systèmes distribués [CDK01]. Ce modèle est né du besoin de la séparation des rôles entre les différentes composantes d'un système distribué. Les principes de base dans ce modèle sont les suivants :

- le client est l'initiateur de l'invocation au serveur ;
- le serveur implémente les services ;
- le client utilise une référence vers le serveur pour pouvoir l'invoquer.

Ce modèle possède plusieurs variations, car dans une application répartie réelle, cette séparation des reponsabilités entre les différentes entités n'est pas toujours suffisante pour définir son architecture. Un client peut jouer le rôle d'un serveur et le serveur peut devenir à son tour un client. Les systèmes pairs à pairs étendent ce mode de fonctionnement.

1.1.1 Le Middleware

On appelle middleware [Ber96], littéralement "élément du milieu", l'ensemble des couches réseaux et services logiciels qui permettent le dialogue entre les différents composants d'une application répartie. Ce dialogue se base sur un protocole applicatif commun défini par l'API (*Application Programming Interfaces*) du middleware.

On peut définir le middleware comme une interface de communication universelle entre processus. Il représente véritablement la clef de voûte de toute application client-serveur. L'objectif principal du middleware est d'unifier pour les applications, l'accès et la manipulation de l'ensemble des services disponibles sur le réseau, afin de rendre l'utilisation de ces derniers presque transparente. Le middleware fournit une image unique du système (*SSI : Single System Image*). Il vise à introduire une couche logicielle complémentaire permettant à l'application de voir la grappe des stations de travail comme une seule ressource bien qu'elle s'exécute sur différentes machines qui forment la grappe.

Un middleware est susceptible de rendre les services suivants [FIA96] :

- Conversion : Service utilisé pour la communication entre machines mettant en œuvre des formats de données différents,
- Adressage : Permet d'identifier la machine serveur sur laquelle est localisé le service demandé afin d'en déduire le chemin d'accès. Dans la mesure du possible, cette fonction doit faire appel aux services d'un annuaire.
- Sécurité : Permet de garantir la confidentialité et la sécurité des données à l'aide de mécanismes d'authentification et de cryptage des informations.
- Communication : Permet la transmission des messages entre les deux systèmes sans altération. Ce service doit gérer la connexion au serveur, la préparation de l'exécution des requêtes, la récupération des résultats et la déconnexion de l'utilisateur.

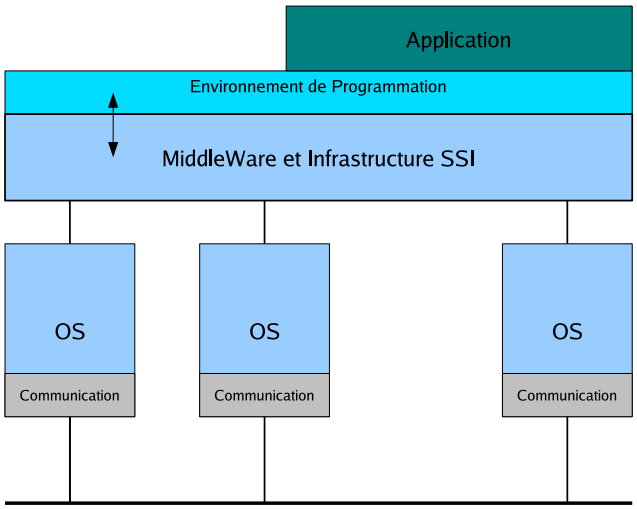

FIG. 1.1 – Positionnnement du Middleware

Le middleware masque la complexité des échanges inter-applications et permet ainsi d'élever le niveau des API utilisées par les programmes. Sans ce mécanisme, la programmation d'une application serait extrêmement complexe et rigide.

1.1.2 L'approche GRID (grille de calcul)

Le terme "grid" [FK00] est utilisé pour désigner la technologie permettant de mettre en liaison un très grand nombre (qui peut atteindre plusieurs millions) de ressources, reliées par un réseau et servant à résoudre des problèmes de calcul scientifique. Ce terme a été choisi par analogie avec le réseau électrique. Cette approche, initialement connue aussi sous le nom de métacomputing a beaucoup évolué. Elle n'est plus seulement utilisée pour le calcul haute performance, mais a aussi donné naissance à plusieurs approches plus ou moins spécialisées comme le calcul Pair à Pair ou le calcul global. Il existe aujourd'hui plusieurs système Grid. Nous pouvons citer Globus[FK97a], Harnes[MS99b, MS99a, DGK+98], XtremWeb [CDF+02] et legion[GW97, LG96, Kar96]. Chacun fournit des services de base ainsi que quelques services supplémentaires spécifiques pour une utilisation donnée.

L'architecture du Grid [FKT01] peut être décrite en identifiant les composants nécessaires pour l'exécution des applications cibles. Une description haut niveau de cette approche est la description en couches de la figure 1.2. Les cinq composantes principales du Grid décrites dans [FKT01] sont :

- Fabrique : cette couche contient toutes les ressources partagées. Ces ressources n'incluent pas seulement les ressources de calcul (PCs, SMPs, cluster), mais aussi leurs systèmes d'exploitation et de gestion, les ressources de stockage de données, les bases de données et des outils scientifiques comme des capteurs ou des télescopes.

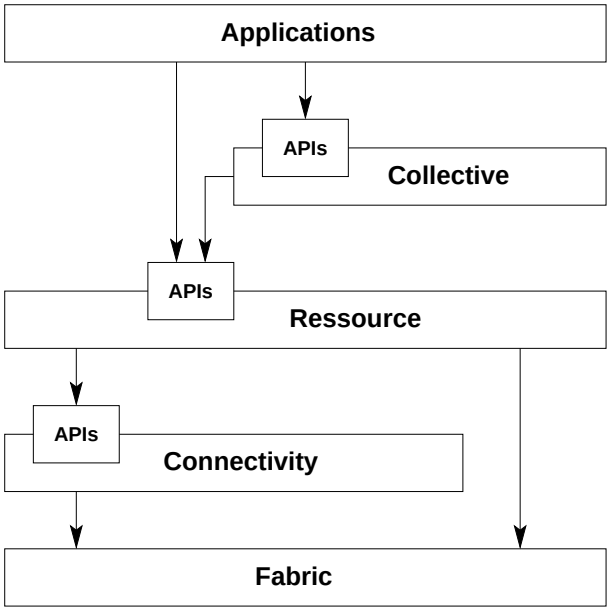

FIG. 1.2 – Architecture du Grid

- Connectivité : cette couche fournit les protocoles de communication et de sécurité. Le protocole de communication permet l'échange de données entre les différentes ressources, alors que le protocole de sécurité (basé sur les services du premier protocole) fournit les mécanismes d'identification des ressources et des utilisateurs, et la sécurisation des échanges. Le service de sécurité peut être basé sur un standard existant pour réduire la complexité du système.
- Ressource : cette couche utilise les services communication et sécurité, pour permettre de construire le protocole sous forme d'APIs de deux services sécurisés : le service d'information qui permet d'obtenir des informations sur les ressources partagées et leur état ; le service de gestion qui permet de gérer l'accès aux ressources partagées : allocation, accès et droit, qualité de service, monitoring, contrôle, ?...etc.
- Collective : cette couche est associée à l'ensemble des ressources partagées et collecte les interactions qui existent entre elles. Le modèle ne spécifie pas l'ensemble des services de cette couche. En pratique, on retrouve un grand nombre de services dont : allocation et ordonnancement des ressources ; monitoring et analyse des applications ; service de réplication de données ; service d'exploration de ressources connues généralement sous le nom de Directory Service. D'autres services existent dans cette couche, on peut citer un service comparable au Directory Service, mais qui permet d'explorer les ressources logicielles disponibles, et le service de programmation dans le système (passage de message, maître/ esclave).
- Application : cette dernière couche contient les applications utilisateurs. Ces applications font appel aux services des couches précédentes qui sont accessibles via des APIs bien définies.

Les systèmes distribués de type grid fournissent les mécanismes de base(bas niveau généralement) du métacomputing. Certains sont mis entre les middlewares et les systèmes de Grid et fournissent modèle de calcul distribué haut niveau(objet par exemple pour

Legion[LG96]), ou des environnements de programmation haut niveau. Tous les systèmes de grid doivent néanmoins fournir les services suivants : localisation et allocation de ressources impliquant des espaces de stockage pour gérer les informations sur les ressources ; création et exécution des processus, avec une politique d'ordonnancement et d'allocation de ressources ; couche de communication entre les processus. Partant de ces services, d'autres services supplémentaires sont construits pour construire un système de Grille puissant et complet. Comme de tels systèmes sont utilisés par un grand nombre de personnes et exécutent des applications à durée plus au moins longue, deux services s'imposent comme étant essentiels, ce sont les services de sécurité et de tolérance aux pannes.

1.2 les objets distribués

L'utilisation des modèles objets dans la programmation distribuée a été motivée par la modélisation et la facilité de conception des applications ainsi que la réutilisation du code. Les objets classiques sont vus comme des entités qui encapsulent des données et un ensemble d'opérations qui peuvent agir sur ces données. Les opérations supportées par un objet(appelées "méthodes") peuvent dépendre de l'état interne de cet objet. Les objets fournissent un moyen propre pour séparer les données des fonctionnalités d'autres parties du système et facilitent la construction et la maintenance d'une application.

Les objets classiques résident dans un programme unique et qu'ils n'existent pas comme entités séparées quand le programme est compilé. Les objets distribués sont des objets étendus qui peuvent résider n'importe où sur le réseau, et existent comme des entités autonomes et qui peuvent être accessibles à distance par d'autres objets. La figure 1.3 illustre le modèle d'objets distribués. Lorsqu'un client invoque un objet distant, une implémentation de l'interface de l'objet, appelée *souche* permet d'empaqueter les paramètres de la méthode invoquée et d'appeler l'objet distant en passant par les squelettes qui font l'opération inverse de dépaquetage des données, et qui transmet l'appel à l'interface de l'objet. Au retour, les données sont empaquetées par les squelettes et transmises au client par l'intermédiaire de la souche qui dépaquette les résultats fournis.

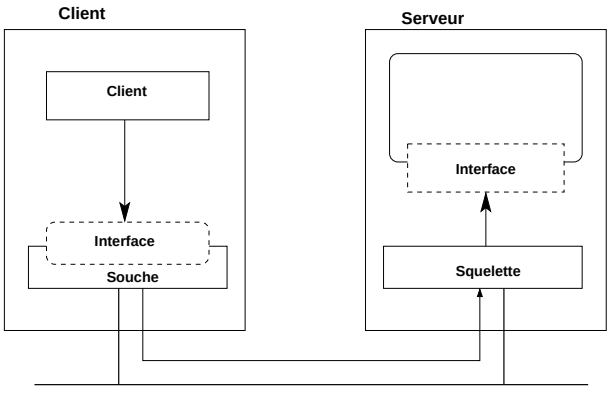

FIG. 1.3 – Le modèle d'objets distribués

Les systèmes à objets distribués permettent aux objets, écrits dans des langages différents et compilés par des compilateurs différents, de communiquer à travers des protocoles

de messages inclus dans un middleware à objets. Souvent dans la construction d'une application distribuée à base d'objets nous avons besoin d'une plateforme avec des services fournis par le middleware pour permettre d'obtenir un niveau plus élevé de transparence et d'inter-opérabilité entre les objets distribués.

1.2.1 Les avantages des objets distribués

Les objets distribués révolutionnent les constructions des systèmes client-serveur à grande échelle en fournissant un logiciel modularisé dans lequel les parties sont interchangeables. Les architectures avancées offrent aux utilisateurs finaux la capacité d'ajouter des objets, permettant une configuration personnalisée des applications.

Les objets peuvent être autogérés. L'autogestion dans un objet est sa capacité d'avoir des interfaces responsables de son autonomie. Les objets conçus pour l'autogestion sont des types d'objets simples à utiliser par le programmeur de l'application. Les objets distribués qui sont autogérés sont responsables de leur propres ressources à travers un réseau et peuvent interagir avec d'autres objets. Les plates-formes à objets distribués donnent aux objets ces capacités en mettant à disposition un middleware pour régulariser les communications inter-objets nécessaires en offrant pour chaque objet des ressources communes qui sont supprimées quand l'objet cesse d'exister.

Les objets gênèrent des événements pour notifier à d'autres objets une action à effectuer. De ce fait, les événements peuvent être vus comme des objets de synchronisation qui permettent à un thread d'exécution de notifier à un autre thread d'exécution que quelque chose s'est produit. En utilisant ce modèle, un événement peut s'adresser à un objet pour exécuter certaines actions. Les événements entre les objets red apportent plus de robustesse à la plateforme pour l'interaction entre les objets, qu'un modèle qui force les objets à attendre l'instruction suivante. Par exemple, un éditeur peut générer un événement de "fin d'impression" quand il a envoyé le document à l'imprimante. Si quelqu'un voulait rajouter une fenêtre de dialogue qui alerte l'utilisateur que l'impression est terminée, un objet écoutant l'événement "fin de l'impression" est nécessaire et doit être écrit. L'éditeur ne connaît que l'existence de l'objet de l'alerte. Dans un modèle traditionnel, l'objet de l'alerte a besoin d'être explicitement informé pour qu'il affiche le dialogue. Cela implique certaines modifications dans le code source de l'éditeur et ensuite une compilation du code qui n'est pas toujours faisable.

1.2.2 Les modèles à objets distribués

La construction des applications à base d'objets distribués dans les systèmes client-serveur conduit à l'introduction et la construction d'infrastructures à base de modèles d'objets. Les modèles d'objets distribués sont apparus comme des spécifications des middlewares qui assurent l'interaction et la connexion entre les objets distribués. Nous pouvons distinguer trois catégories principales dans le monde des objets distribués : le modèle OLE, COM et DCOM [FR97, Mic98, Cha96], le modèle CORBA [Sha01, Sun00][OMG] de l'OMG(Object Management Groupe), et modèle JAVA RMI [Sha01, Sun00] de SUN basé sur le langage JAVA.

OLE, COM et DCOM

Sous l'appellation OLE, on désigne en fait toute l'architecture logicielle de communication inter-applications de Microsoft, allant de la spécification COM aux contrôles ActiveX. L'architecture OLE présente un certain nombre de services que le programmeur peut exploiter dans ses applications Windows.

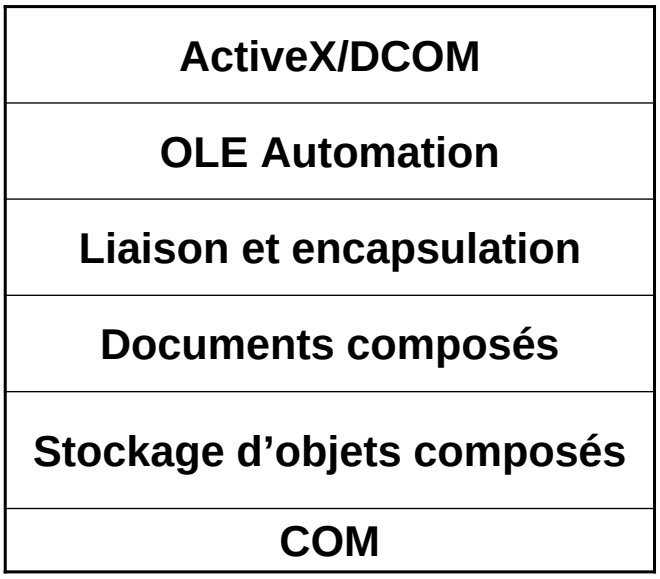

FIG. 1.4 – L'environnement OLE

Cette architecture est basée sur le modèle COM (Component Object Model). On trouve ensuite un ensemble de services concernant les documents composés (Compound Objects) permettant l'échange de données entre documents. Ces services sont représentés par trois couches dans la figure 1.4. Au dessus, on distingue le service OLE Automation qui permet à un programme d'exécuter des méthodes d'un objet serveur. Enfin, la dernière brique de cet édifice est ActiveX qui correspond en fait à des contrôles OLE renommés début 96 par Microsoft. On distingue également à ce niveau le modèle DCOM, version distribuée du modèle COM.

Le Component Object Model est, comme nous l'avons dit, le format des applications OLE, leur permettant ainsi de pouvoir communiquer entre elles. Ce format est indépendant du langage de programmation choisi, l'essentiel est que l'exécutable obtenu respecte le modèle. La figure 1.5 illustre le modèle COM. Notons que nous parlons d'exécutable mais en réalité il s'agit plutôt de bibliothèques de fonctions exécutables par plusieurs applications OLE.

Il est important de remarquer qu'OLE est un environnement basé sur des objets client/-serveur. Un objet, hébergé par un serveur (bibliothèque DLL, application etc.), peut être utilisé par plusieurs programmes clients simultanément.

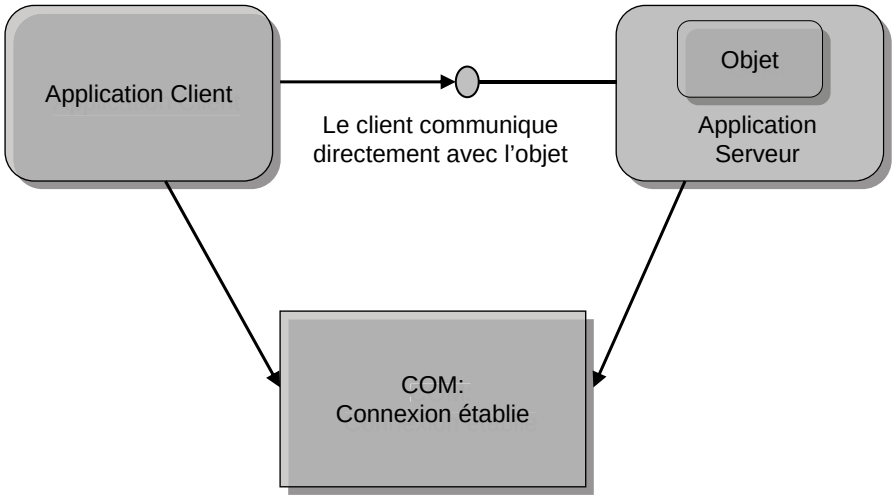

FIG. 1.5 – Le modèle COM

OLE Automation est un service très puissant qui permet de manipuler une application en appelant dynamiquement les méthodes des objets, respectant le format COM, qu'elle contient. Cette manipulation est typiquement faite via un langage de scripts tel que VBA, mais ce n'est pas une obligation.

En fait, OLE Automation permet à une application de découvrir dynamiquement les objets et méthodes disponibles dans une autre application afin de pouvoir les utiliser. OLE Automation permet également de construire des serveurs d'objets qui sont capables de savoir quelle méthode de quel objet appeler suite à une requête d'une application cliente.

On définit dans OLE Automation des objets automates (Automation Objects) qui pourront être manipulés à l'aide de langages de scripts tels que VBA ou être invoqués dynamiquement. Ces applications clientes sont appelées des contrôleurs d'automates.

Distributed COM est la version distribuée du modèle COM qui permet à des objets appartenant à différentes machines de communiquer via un réseau, alors que COM est limité à une même machine. DCOM est basé sur l'environnement DCE (Distributed Computing Environment) , standardisé par l'OSF, dont il exploite en particulier les appels de procédures distantes RPC (Remote Procedure Call), comparables dans leur principe au système RMI dont nous allons parlé pour Java.

Par conséquent, à chaque fois qu'on envoie un message DCOM à un objet distant, ce message est en réalité transporté via RPC. Nous précisons enfin que la gestion des références d'accès aux objets dans un environnement DCOM est quelque peu améliorée pour tenir compte des aléas d'une communication via un réseau. En effet, en cas de défaillance d'une machine distante, les références enregistrées par un client ou un serveur risqueraient de ne plus être les mêmes. C'est pourquoi DCOM inclut un mécanisme de

ping des clients vers les serveurs permettant ainsi d'indiquer à un serveur qu'un client est toujours opérationnel.

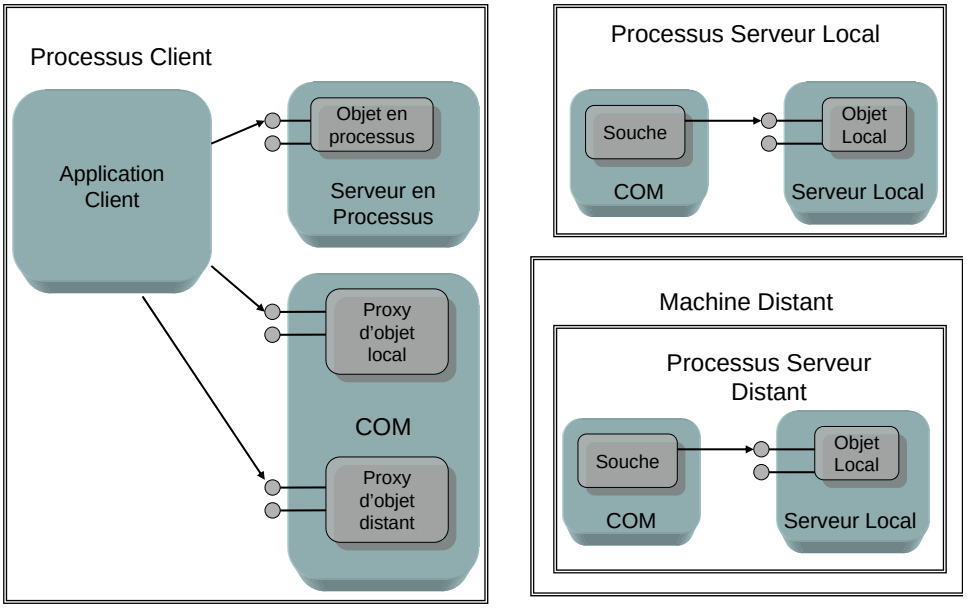

FIG. 1.6 – Le modèle DCOM

CORBA

CORBA est un middleware orienté objets qui est indépendant des langages de programmation. Il est basé sur l'OMA (Objet Management Architecture) et le COM (Core Object Model). L'architecture définie par l'OMA identifie les parties essentielles d'une application distribuée. Elle se compose principalement de cinq parties [JGP99, AMR99] :

1. Le bus objet ORB (Object Request Broker) est la partie principale de l'architecture. L'ORB prend en charge les communications entre les différents acteurs présents dans l'architecture. Il permet aussi de masquer tous les détails de communication entre le client et l'objet serveur [Vin97]
   - la localisation de l'objet serveur : le client accède à l'objet serveur de la même façon, que ce dernier se trouve dans le même espace d'adressage que le client ou sur une machine accessible à travers le réseau.
   - l'implémentation de l'objet serveur : le détail du langage d'implémentation, du système d'exploitation ou de l'architecture matérielle est transparent au client.
   - l'état de l'objet serveur : l'ORB permet d'activer automatiquement l'objet serveur lors d'une invocation du client.
   - les mécanismes de communication : l'ORB peut utiliser des protocoles de communication spécifiques de façon transparente pour le client.

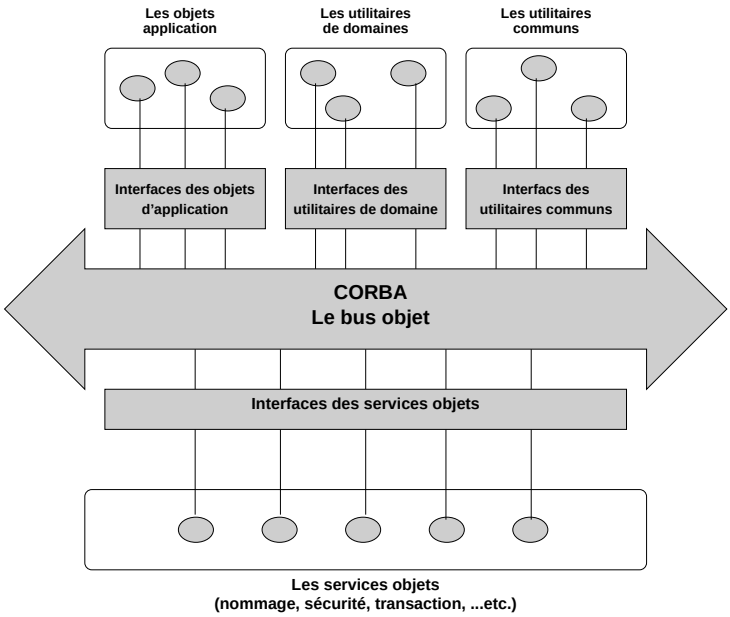

FIG. 1.7 – Architecture de l'OMA

2. Les services objets (*CORBA services*) : définissent les objets systèmes qui permettent d'élargir les fonctions de l'ORB par des services supplémentaires tels le service de nommage d'événements, de transaction, ...etc.
3. Les utilitaires communs (*CORBA facilities*) : définissent les outils utilisés par les applications. Ces outils se placent au niveau supérieur par rapport aux "service objets", et prennent en charge les besoins communs à un grand nombre d'applications. Par exemple, les gestionnaires d'impression, les outils de gestion documentaire ou les gestionnaires de messagerie sont tous des utilitaires communs.
4. Les interfaces de domaine (*Domain Interfaces*) : définissent les interfaces des "applications métier" pour différents domaines. Elles couvrent des besoins spécifiques comme par exemple les domaines de finance, de santé ou de télécommunication.
5. Les objets d'application : sont les applications développées par les utilisateurs. Contrairement aux quatre catégories précédentes, ces applications ne sont pas standardisées, car elles dépendent des besoins des développeurs.

En CORBA les objets peuvent être écrits dans des langages de programmation différents. Dans ce contexte de collaboration inter-langage, deux aspects sont introduits :

- L'interface de l'objet a été séparée de son implémentation. L'interface spécifie toutes les méthodes et attributs publiques ;
- Un langage de définition d'interface IDL (*Interface Definition Language*) a été introduit. Ce langage est utilisé pour spécifier les interfaces indépendamment des langages de programmation.

Dans ce modèle le client et le serveur (qui est un objet CORBA) communiquent par des appels de méthodes distantes. L'ORB permet de masquer la localisation de l'objet, son implémentation, son état d'exécution et les mécanismes d'exécution. L'ORB fournit deux mécanismes d'invocation de méthodes :

- les invocations statiques sont utilisées quand les objets sont connus à la compilation. Le client utilise les souches IDL et le serveur utilise les squelettes statiques. Ces souches et squelettes sont générés par la compilation des interfaces du client (si elles existent) et du serveur ;
- les invocations dynamiques sont utilisées lorsque les objets sont inconnus à la compilation. Le client et le serveur utilisent respectivement les interfaces d'invocation dynamique et les interfaces de squelette dynamique. Ces mécanismes utilisent une composante CORBA normalisée l'*Interface Repository*, qui garde une représentation des interfaces IDL. C'est grâce à ces deux mécanismes et à l'*Interface Repository* que la norme CORBA supporte la réflexivité. Par rapport à un modèle d'invocation statique, ce mode présente l'avantage de pouvoir supporter des environnements dynamiques, par contre il est moins efficace en temps de communication.

Pour les communications inter-objets, l'ORB utilise un protocole de communication haut niveau indépendant des protocoles de transport, appelé GIOP (*General Inter-ORB Protocol*). Ce protocole définit une représentation externe des données appelée CDR. L'implémentation de GIOP sur le protocole TCP/IP est appelée IIOP (*Internet Inter-ORB Protocol*).

Le modèle objet de CORBA est un modèle à objets distribués, mais le client n'est pas forcément un objet et peut être un programme quelconque qui envoie des requêtes et reçoit les réponses. Dans ce modèle, le terme objet CORBA est utilisé pour référencer les objets distants dont l'implémentation est dans le même espace d'adressage que le serveur.

Dans CORBA, la description des interfaces se fait en IDL et il existe des règles de traduction du langage IDL vers chaque langage cible. D'autres systèmes comme DCOM [Red97] ou Globe [HST99], utilisent une approche différente basée sur les interfaces binaires. Avec ce type d'interface, les règles de traduction vers les langages cibles ne sont plus nécessaires car il utilise des pointeurs vers des tables de fonctions virtuelles. On peut noter qu'un objet CORBA peut être implémenté avec un langage qui n'est pas objet (comme C ou COBOL) et la notion de classe n'existe pas dans le langage CORBA IDL.

JAVA RMI

Le modèle java étant incapable de spécifier comment initier des traitements dans un espace d'adressage distant, *SUN* a définit un mécanisme appelé RMI [SUN98a] (*Remote Method Invocation*), qui permet au programmeur d'appeler des objets distants. Schématiquement, le mécanisme RMI permet, sur une machine appelée *machine serveur*, de créer un objet et de le rendre accessible aux autres machines, pour lesquelles il devient un objet distant. Sur l'une de ces machines, dite *machine cliente*, une application Java peut alors récupérer, localement, une représentation de l'objet distant, appelée *souche (stub)*. Elle utilise cette souche comme elle utiliserait un objet local sur la machine cliente. Notamment, elle peut appeler, sur cette souche, les méthodes d'instance définies dans la classe de l'objet distant. Cet appel local de méthode sur un talon de l'objet distant est alors transmis, via le réseau, depuis la machine cliente jusqu'à la machine serveur où il engendre l'exécution, sur l'objet distant, de la méthode correspondant à celle qui a été appelée à distance.

Les objets statiques sont considérés comme uniques dans un programme centralisé,

et ainsi, les méthodes statiques sont comme des méthodes uniques. Le même principe s'applique pour les attributs statiques d'une classe. Dans le cas distribué, les classes sont dupliquées implicitement sur toutes les machines et les attributs et méthodes statiques ne sont plus uniques dans l'environnement. En conséquence, la sémantique de la partie statique interdit l'usage des attributs statiques ou l'appel des méthodes statiques en RMI.

Le principe du mécanisme RMI, similaire à celui de RPC (Remote Procedure Call) qui permet d'appeler des fonctions sur un système distant, repose sur une analogie entre le modèle client-serveur et l'appel de fonctions dans les langages de programmation. Les différentes étapes lors d'un appel de méthode, comparé à un appel de fonction, sont :

- la connexion d'un client qui correspond à l'appel de fonction,
- la requête qui correspond au passage des paramètres,
- le traitement du service qui correspond à l'exécution du corps de la fonction,
- la réponse du serveur qui correspond à la valeur de retour de la fonction.

L'architecture de RMI se compose de quatre couches :

1. la couche application : c'est cette couche qui contient l'implémentation du client et du serveur.
2. la couche proxy : c'est à travers cette couche que l'application communique. Tous les appels vers les objets distants, l'empaquetage et le dépaquetage des paramètres et le retour des résultats se font dans cette couche. Les couches et squelettes, qui représentent respectivement les côtés client et serveur sont les classes compilées par le compilateur RMI (RMIC).
3. la couche référence distante : c'est elle qui assure l'aspect fonctionnel de l'application. Elle assure notamment une référence persistante vers l'objet distant (avec re-connexion éventuelle si besoin est).
4. la couche transport : c'est elle qui assure la connexion entre le client et le serveur, la gestion et la localisation de tous les objets distants répertoriés. Elle assure les fonctions suivante :
   - Connexion entre les espaces d'adressage ;
   - Suivi des connexion en cours ;
   - Ecoute et réponse aux invocations ;
   - Constitution d'une table de tous les objets distants.

   Elle repose sur TCP, mais pourait aussi utiliser UDP ou SSL (*Secur Socket Layer*) grâce au *Socket Factory*.

1.3 L'approche Composants

L'approche composants est la continuité de l'approche orientée objets. Les composants logiciels promettent plus de réutilisabilité et d'encapsulation de morceaux logiciels. La plupart des modèles composants qui existent, sont d'une façon ou d'une autre l'extension d'un modèle objets ou basés sur les méthodes orientées objets.

Avant que le terme composant ne soit populaire, des termes comme sous-routine, procédure, fonction, module et objet ont été utilisés dès les premiers jours en développement logiciel pour identifier les fonctionnalités du logiciel. En général, les notions évoquées par

ces termes ont été introduites et utilisées avec comme objectifs la réutilisabilité, la gestion de la complexité du logiciel et le développement de systèmes plus flexibles. Le génie logiciel est un processus itératif qui comporte plusieurs étapes incluant la modélisation, la conception, l'implémentation et le déploiement. Le concept de composant a été utilisé dans toutes ces étapes sans aucun consensus pour une définition commune. D'autres termes sont apparu avec la notion de composant logiciel. La composition ou l'assemblage des composants permet d'obtenir une application à partir des composants interconnectés. Le déploiement est une notion qui détermine l'environnement (un système centralisé, distribué, parallèle ...etc) dans lequel les composants sont exécutés ainsi que leur interactions avec cet environnement. Dans les sections qui suivent nous montrerons plus en détails les notions relatives à la construction des applications à base de composants. Nous détaillerons d'abord les motivations d'utilisation de la technologie de composants logiciels dans la construction et le développement des applications.

1.3.1 Motivations

Les logiciels sont toujours plus importants et plus complexes. Cependant, les approches classiques de programmation rendent l'évolutivité et la maintenance des gros logiciels relativement difficiles. De plus, la plupart des logiciels existants ne sont pas conçus pour inter-opérer avec d'autres applications encore plus avec des nouvelles technologies. Ce manque d'inter-opérabilité d'un logiciel a des effets négatifs sur l'efficacité des systèmes d'entreprise.

Le développement basé sur les composants logiciels [Mey99] est la construction de systèmes logiciels à partir d'éléments génériques déjà encapsulés dans des paquetages(archives diffusables). La motivation du développement par composants logiciels a quatre raisons différentes [MM99] :

1. du coté scientifique : la réutilisabilité du code scientifique développé par plusieurs chercheurs.
2. du coté industriel : le grand succès de la construction à base de composants des interfaces utilisateurs *GUI*, des bases de données et d'autre part d'applications. Les VBX, OCX et ActiveX [Cha96, K.B95] de *Microsoft* et Delphi [Bor06] de *Borland* en sont de bons exemples.
3. du coté politique, la concurrence entre les technologies CORBA, COM et EJB (c.f. 1.4) pousse les acteurs principaux de l'industrie à définir leur propres standards.
4. du monde du logiciel, dans le sens large et la généralisation de la technologie objets qui fournit les bases conceptuelles et les outils pratiques pour construire et utiliser des composants.

Les facteurs qui ont obligé ces quatre acteurs à s'intégrer dans le mouvement actuel de développement à base de composants, sont la demande croissante dans l'industrie du logiciels des nouvelles technologies pour contrôler le coût de développement des produits logiciels et améliorer leurs qualités. N'importe quelle solution a besoin de l'industrialisation du processus, basé sur la réutilisation des composants standards ; ce qui évite de tout reconstruire à chaque application.

La qualité du logiciel est un objectif très important dans le développement à base de composants. La littérature du développement à base de composants est divisée en deux groupes : celle qui se concentre sur les aspects techniques et essaie de tirer avantage de la disponibilité de la construction et des mécanismes d'interopérabilité, et celle qui mise sur la garantie de la qualité de ces composants. La qualité des composants est liée à la définition de contrats [Mey00, Szy00b] sur les composants et sur la façon dont les composants prennent en comptes ces contrats. Les contrats de composants sont présentés en quatre niveaux [BJPW99] : contrats de base comme celui qui est fourni par une simple interface listant les opérations et leurs signatures (les types d'entrée et de sortie) avec des propriétés sémantiques ; des contrats de comportement qui rendent possible d'exprimer ce que font les opérations indépendamment de la manière dont elle font ; des contrats de synchronisation dans les systèmes parallèles et distribués et des contrats de qualité de service qui mettent l'accent sur les aspects systèmes.

Dans le monde du calcul parallèle et distribué, le besoin de capacité de calcul de haute performance et de parallélisme intensif ajoute une énorme complexité aux logiciels. Le respect des applications existantes est l'une des principales considérations des réalisateurs des logiciels pour le calcul performant [RC00]. Une des solutions est de réécrire les applications avec les nouvelles technologies et d'améliorer les méthodologies pour produire les mêmes capacités des logiciels. Mais tout cela conduit à un cycle de ré-implémentation à chaque fois que des nouvelles technologies apparaissent. Ce qui rend la tâche très coûteuse en temps et en efforts. Une autre solution consiste à développer un mécanisme qui supporte les deux : la réutilisation des applications héritées (legacy) et l'addition des nouvelles capacités de logiciel. Cette solution permet aux applications existantes d'être intégrées dans les systèmes logiciels avec le minimum de modifications possibles et avec un coût moindre.

Avant de présenter les composants et les architectures logicielles à base de composants nous allons présenter quelques définitions portant sur les composants, l'architecture et leurs environnements.

1.3.2 Définitions et caractérisation d'un composant

Il n'existe pas aujourd'hui une définition générale du composant logiciel. Toutefois, plusieurs définitions ou caractérisation sont bien acceptées. La plupart des définitions sont similaires aux définitions données par Szyperski et Mayer.

Szyperski [SGM02, Szy00a], directeur de recherche à Microsoft a donné la définition suivante :

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

Mayer [MM99] a défini le composant logiciel comme suit :

A software component is a program element with the following properties : the element may be used by other program elements(clients) ; the clients and their authors do not need to be known to the element's authors. The first property excludes programs meant only for

use by humans or, as with embeded software, by nonsoftware systems. The second property excludes the simple case of a module that is used by other modules but without fundamental requirement of general reusability. A true component must be usable by software developers who build new systems not forseen by the component's author and who are not personally known to that author.

A partir de ces définitions et d'autres dans la litterature [Crn01, K.W02, RPS03] nous proposons une définition du composant logiciel qui nous sera utile dans le processus de developpement du logiciel à base de composants pour le calcul parallèle distribué. Notre définition de composant est la suivante :

Un composant logiciel est une unité logicielle independante pouvant entrer dans la construction d'une application. Il satisfait un ensemble de règles de comportement et il implémente des interfaces standards qui lui permettent d'être composé avec d'autres composants et d'être déployés sur des plateformes d'exécution .

A partir de cette définition du composant logiciel, nous remarquons qu'il faut définir les outils et les règles qui sont indispensables pour la construction des composants et des applications à base de ces composants.

Définitions Nous donnerons ici quelques concepts relatifs aux composants logiciels des termes proches afin de présenter une définition à cette architecture.

- Un *élément logiciel* est une séquence d'un programme qui décrit un calcul exécuté sur une machine.
- Une *interface* est une abstraction du comportement d'un composant qui constitue un sous-ensemble des interactions de ce composant avec les autres composants et un ensemble de contraintes qui décrivent quand ces interactions peuvent se produire. L'interface décrit le comportement du composant qui est obtenu en considérant seulement les interactions de cette interface et en cachant toutes les autres interactions.
- Une *composition* (assemblage) est la combinaison de deux ou plusieurs composants qui produit un nouveau composant dans un niveau différent d'hiérarchie. Les caractéristiques du nouveau comportement de composant sont déterminées par les composants qui ont été combinés et par la façon dont ils sont combinés.
- Un *modèle de composant* définit l'interaction spécifique, entre les composants eux-même et entre eux et le monde extérieur, et les standards de la composition. Une *implémentation du modèle de composant* est l'ensemble dédié des éléments du logiciel exécutable qui sont nécessaires pour soutenir l'exécution des composants conformément au modèle.

Nous différencierons ici quatre notions de base dans l'approche orientée composants qui tendent à étendre les définitions :

– le type de composant,
– l'implémentation de composant,
– le paquetage de composant,
– l'instance de composant.

Type de composant

Un type de composant est la définition abstraite d'une entité logicielle qui représente un modèle de composant. Un composant selon un modèle est caractérisé par trois éléments : ses interfaces, les modes de coopération avec les autres composants et ses propriétés (paramètres de configuration). La mise en place des ces trois éléments définit un type de composant.

Il est important qu'une interface soit définie de manière indépendante à toute implémentation. Il est ainsi possible d'une part, de fournir plusieurs implémentations pour un même type de composant et d'autre part, une implémentation de composant peut être substituée par une autre implémentation du même type.

Les interfaces d'un composant peuvent être de deux types :

- les interfaces *fournies* par le composants sont du même ordre que les interfaces des objets : elles définissent les services fournis par le composant, en listant les signatures des opérations fournies, ainsi que les différentes données entrantes et/ou sortantes du composant.
- les interfaces *requises* par le composant représentent un progrès par rapport à l'approche objet. Dans le cas des objets, une référence sur un objet utilisé est enfouie au coeur du code. Dans le cas des composants, les interfaces des types de composants utilisés sont exprimées au niveau du composant. Il est ainsi plus aisé d'une part, de substituer un composant à un autre et d'autre part, de gérer les connexions entre des composants, i.e. de connaître les dépendances pour l'installation et le remplacement.

Pour chacune des interfaces du composant, il est nécessaire de spécifier le mode de coopération avec les autres composants. Trois modes de coopération sont souhaitables : le mode synchrone (par exemple l'invocation de méthode), le mode asynchrone (par exemple l'utilisation d'événements) et le mode diffusion en continue (par exemple les flots de données).

Le dernier élément caractérisant un composant dans un modèle est l'ensemble de ses propriétés configurables. Ces propriétés permettent d'adapter une instance de composant à l'application qui l'utilise en configurant son comportement. Le code du composant n'est pas modifié en fonction du besoin, mais paramétré, augmentant ainsi la réutilisabilité potentielle du composant. Dans le cas d'un composant potentiellement persistant, il peut se trouver en n états. Un sous-ensemble des propriétés configurables du composant constitue une possibilité pour représenter cet état.

Implémentation d'un composant

L'implémentation d'un composant regroupe deux notions : l'implémentation fonctionnelle et l'implémentation non-fonctionnelle. L'implémentation fonctionnelle d'un composant représente sa logique de traitement : *la logique métier*. C'est l'intelligence qui est mise dans l'implémentation du composant. L'implémentation non-fonctionnelle représente tout le reste. C'est à dire, d'une part, la partie de l'implémentation qui sert à la gestion des connexions entre les instances de composants, mais aussi, tout ce qui peut être décrit et ainsi non programmé.

Par exemple, dans le cas d'un porte-monnaie électronique, l'implémentation fonction-

nelle regroupe la réalisation des services tel le débit ou le crédit d'une somme. L'implémentation non-fonctionnelle regroupe quant à elle les aspects relatifs à la qualité de service comme les besoins transactionnels, de sécurité et de persistance.

Contrairement aux objets où tout est pris en charge de manière programmée, la partie d'implémentation programmée est minimale. Des propriétés comme la persistance, les besoins transactionnels ou la sécurité d'un composant peuvent être décrits et leur prise en charge automatisée.

La prise en charge automatisée des propriétés non-fonctionnelles d'un composant peut se baser pour une part sur la spécification du composant, mais doit aussi prendre en compte les besoins de l'implémentation fonctionnelle. Il est donc nécessaire de fournir un outil pour l'intégration de l'implémentation non-fonctionnelle et de l'implémentation fonctionnelle.

Paquetage de composant

Un paquetage de composant est une entité logicielle diffusable et déployable, bien souvent une archive, contenant la définition du type de composant, au moins une implémentation de ce composant et une description du contenu du paquetage : type de composant et contraintes techniques de l'implémentation, par exemple "en C++, destinée à Solaris, nécessitant la bibliothèque dynamique libToto.so.6". L'utilisation de paquetages logiciels permet de rendre diffusable un composant dans sa forme binaire.

Ici encore, un gain apparaît par rapport aux objets : un composant est diffusable de manière unitaire et sous forme binaire. Il n'est pas nécessaire de connaître l'implémentation du composant pour l'intégrer dans une application, seule la connaissance de son type est requise.

Instance de composant

Une instance de composant est, au même titre qu'une instance d'objet, une entité existante et s'exécutant dans un système. Elle est caractérisée par une référence unique, un type de composant et une implémentation particulière de ce type. De la même manière que pour une instance d'objet, une instance de composant peut recevoir des requêtes de service. Une instance de composant est aussi configurable. Cette configuration est faite lors de l'instantiation du composant. La configuration peut être statique lors de la création de l'instance ou dynamique pendant l'exécution de l'application par l'intervention du constructeur de l'application. Les paramètres de configuration sont comme des données d'entrée propres à chaque instance.

1.3.3 Avantages de l'approche composant

L'approche composant reprend et prolonge les avantages des objets. En effet, les aspects encapsulation, modularité, spécification des interfaces fournies sont conservés et d'autres points tels que le déploiement, la composition dynamique, la spécification des interfaces requises sont ajoutés. Nous présentons dans cette section, les aspects qui ont

été développés dans l'approche composant et qui la rendent d'autant plus efficace pour la construction d'applications.

Différences et progrès par rapport aux objets

La programmation par composants repose sur quelques concepts importants. Lors de la définition des modèles objet et composant, deux abstractions, boîte noire et boîte blanche, ont été définies pour démontrer l'intérêt des différents modèles. Ces abstractions reflètent la visibilité d'une implémentation à travers son interface. Dans une abstraction de type boîte noire idéale, aucun détail hormis l'interface et sa spécification n'est connu des clients. Dans une abstraction de type boîte blanche, l'interface peut imposer l'encapsulation et les limites de ce que les clients peuvent faire. Cependant, l'implémentation d'une boîte blanche est entièrement disponible.

L'approche orientée objets a abouti dans la majorité des langages à une abstraction de type boite blanche. La manipulation d'un objet se fait effectivement au travers de méthodes, mais le corps de l'objet n'est pas totalement masqué. D'autre part, la réutilisation d'un objet se fait principalement au niveau du langage et non au niveau du binaire. Les composants sont quant à eux des abstractions de type boite noire, seuls les types sont manipulés. De plus, les implémentations de composants doivent être réutilisables sous forme binaire. Par exemple, dans la plus part des bibliothèques et plateformes de classes, le développeur a besoin d'étudier l'implémentation des classes pour savoir ce qu'offrent ces classes ou leurs sous-classes. Alors que dans une réutilisation d'un composant il n'a besoin que de connaître ses interfaces et ses spécifications d'utilisation.

Selon cette définition, un programme à base de composants se décompose en un ensemble de boîtes noires autonomes dans le sens où il est difficile de connaître au lancement d'une application quels composants vont exister à un instant donné de son exécution, quelles propriétés ces composants posséderont, et dans quelles coopérations ils interviendront.

Cette approche boîte noire permet de masquer l'hétérogénéité des composants, c'est-à-dire le fait qu'ils ont été conçus et développés indépendamment les uns des autres, et qu'ils sont éventuellement écrits dans des langages différents.

La modularité permise par l'approche boîte noire, composant, permet une bonne décomposition logique et physique de l'application en composants logiciels autonomes et susceptibles d'activités indépendantes. Cette modularité assure une indépendance des différents composants permettant ainsi leur interchangeabilité. Cette propriété offre de la souplesse quant à l'évolutivité de l'application et à son adaptabilité. En effet, un ensemble de composants peut être complété par d'autres composants ou bien la configuration d'un composant peut être modifiée en fonction de son contexte d'exécution. Cette capacité de modifier la configuration d'un système offre une grande flexibilité d'évolution de l'application.

Un autre apport fondamental de l'approche composant est la capacité de déployer une application, distribuée ou non, de manière automatique à partir de règles de déploiement qui sont fournis dans un descripteur de déploiement attaché à l'environnement de construction et de déploiement de l'application.

Nous regroupons sous le terme de déploiement l'installation des implémentations de

composants requises par une application, la création des instances de composants, l'interconnexion de ces instances et la configuration de cet assemblage. L'aspect boite noire permet d'atteindre cet objectif dans le sens où un composant est réutilisé de manière binaire, et sans a priori sur son implémentation.

Un dernier point important pour l'instanciation de composants est la configuration de ces instances, c'est-à-dire les propriétés configurables (paramétrage) d'un composant qui sont données au moment de l'instanciation et qui permettent plus de réutilisabilité d'un composant. Il est nécessaire de faire une différence entre la phase d'instantiation et de configuration d'une instance de composant, et la phase d'utilisation de cette instance. Un composant peut donc être instancié avec plusieurs configurations possibles. La mise en œuvre de ces deux phases en exclusion mutuelle permet de prévenir des instances mal configurées, mais rendues disponibles à des clients. Ici encore, les objets n'offrent aucune garantie, après l'instanciation, de la bonne configuration de l'instance nouvellement créée.

Nouvelle approche de la construction d'applications

La construction d'applications selon une approche à base de composants consiste à décrire une application en termes de composants inter-connectés. Ainsi apparaît une nouvelle approche de la conception d'application qui consiste davantage dans l'expression que dans la programmation. En effet, la connaissance d'un langage de programmation n'est plus une nécessité pour construire une application.

Pour atteindre cet objectif de simplification de construction d'applications réparties, des efforts d'identification des aspects automatisables sont réalisés afin de décharger le développeur de composants. Ces aspects, non liés aux fonctionnalités de composants, sont appelés propriétés non-fonctionnelles.

Le cycle de vie d'une application à base de composants est différent de celui d'une instance et comprend d'autres phases importantes dans le processus de la production. Les objets n'offrent pas de réponse à toutes les phases du cycle de vie d'une application. Nous utilisons ici un découpage assez fin de ce cycle de vie pour souligner les nouvelles réponses offertes par les modèles de composants. Ce cycle se décompose en sept rôles :

1. Analyse des besoins en terme d'applicatif et de composants,
2. Conception des composants,
3. Implémentation des composants,
4. Diffusion des implémentations de composants,
5. Assemblage des composants créés ou dèjà existants,
6. Déploiement des implémentations de composants et des applicatifs,
7. Utilisation des instances de composants et des applicatifs.

1. **Analyse**

   La phase d'analyse, ou de définition des besoins, représente l'étape zéro de tout processus de construction de logiciel. Le résultat de cette étape est en règle générale un ensemble de diagrammes, par exemple en UML (Unified Modeling Language) [UML97], décrivant les besoins.

Cette étape d'analyse se retrouve à deux niveaux : au niveau applicatif (quels sont les besoins en termes de fonctionnalités de l'application ?), ainsi qu'au niveau composants (quelles sont les fonctionnalités d'un composant ?). Il est important de noter que cette analyse des besoins est idéalement réalisée indépendamment du modèle (sans prise en compte des contraintes techniques du futur modèle de composants).

2. **Conception**

   La conception s'attache à spécifier les types de composants dans le modèle à partir de l'expression des besoins. Pour cette phase de conception, il semble important de disposer d'un langage de déclaration. L'utilisation de ce langage doit permettre la spécification des trois aspects du composant : ses interfaces, ses propriétés, ainsi que la manière dont il coopère avec les autres types de composants.

   Pour que les futures instances soient adaptables, il est nécessaire d'exprimer les propriétés configurables du composant. Dans un but de persistance, il peut être aussi important d'exprimer de manière abstraite l'état interne du composant. Cet état abstrait peut aussi bien être représenté par un ensemble de propriétés que par un processus de sérialisation (avec les limitations de ce dernier choix).

3. **Implémentation**

   La réalisation d'une implémentation doit se faire conformément aux spécifications du modèle de composant visé. Dans un souci de qualité et de simplicité, il semble important que l'implémentation d'un type de composant se résume à l'implémentation des parties fonctionnelles. Un développeur doit pouvoir se focaliser sur la valeur ajoutée d'un composant : la *logique métier*. L'ensemble de l'implémentation non-fonctionnelle est idéalement prise en charge et automatiquement générée à partir des différentes descriptions (spécification du composant et description des besoins de l'implantation fonctionnelle).

   Cependant, l'implémentation fonctionnelle ne peut pas se faire de manière indépendante. Un cadre d'implémentation doit être fixé pour permettre l'intégration de la partie fonctionnelle avec les parties automatiquement générées. Il est nécessaire de pouvoir exprimer les contraintes techniques de l'implémentation : langage, multi-threading, bibliothèques, OS, etc. Le regroupement de ces éléments dans un descripteur doit permettre de garantir ces contraintes lors du déploiement.

4. **Diffusion**

   La diffusion d'une implémentation de composant se fait au travers de la mise à disposition du paquetage de cette implémentation de composant. Une fois un composant produit, deux types d'utilisations peuvent en être fait : sa diffusion à des tiers pour sa composition et son utilisation directe. Ces deux types d'exploitation du composant nécessitent un format de diffusion : le paquetage.

   Pour pouvoir utiliser un paquetage de composant, deux aspects sont importants. D'une part, le format du paquetage doit être standardisé. Il est ainsi utilisable via des outils multi-fournisseurs. D'autre part, le paquetage doit inclure une description de son contenu : les interfaces, les implémentations disponibles et leurs contraintes techniques.

5. **Assemblage**

Dans cette phase de production, la construction d'une application correspond à la réalisation d'un assemblage de composants. Cet assemblage reflète l'architecture de l'application. Ainsi une application est un ensemble de composants inter-connectés et configurés individuellement en fonction du comportement global souhaité.

Pour qu'un assemblage de composants soit diffusable à son tour, et utilisable par des tiers, il doit être packagé. Dans ce cas, le descripteur de paquetage contient une description de l'assemblage et de la configuration de chaque composant. Il est souhaitable qu'un assemblage de composants soit aussi un composant, et ainsi utilisable comme tel pour être assemblé.

6. **Déploiement**

   Le déploiement d'un paquetage logiciel doit être idéalement automatisé. Dans ce sens, le déploiement ne doit pas être uniquement réalisé par des personnes, mais principalement par des outils. Pour permettre cela, le déploiement doit s'appuyer sur une description du processus d'installation d'un paquetage logiciel. Le second élément indispensable à une installation automatique est la mise à disposition de structures d'accueil (environnements d'exécution) pour réceptionner les instances de composants. Ces structures d'accueil sont utilisées par les outils de déploiement pour installer les composants aux endroits désirés. Le choix d'une structure d'accueil est fonction de ses possibilités et des besoins, en terme de contraintes techniques et de ressources, requis par les implémentations de composants à installer.

   En plus de fournir l'automatisation de l'installation, les structures d'accueil offrent un environnement d'exécution. Elles doivent donc permettre d'instancier des composants et offrir un environnement d'exécution à ces instances. Il est donc souhaitable de disposer, au sein d'une structure d'accueil, de gestionnaires d'instances. Ces gestionnaires ont pour but de permettre la création de nouvelles instances et de retrouver des instances existantes.

7. **Utilisation**

   L'utilisation regroupe ici l'utilisation des services offerts par les instances de composants et par les applications bâties à partir de ces instances. Les acteurs potentiels de cette utilisation sont les administrateurs d'applications distribuées qui vont installer et maintenir les applications et les utilisateurs finaux qui vont exploiter ces services.

   Pour ces deux acteurs, l'utilisation des instances de composants peut se faire de manière statique, au travers de programmes clients prévus à cet effet, ou dynamique, en découvrant et en exploitant à l'exécution les interfaces des instances de composants. Il est donc important de prévoir sur un composant des possibilités d'introspection. Il nous semble important que l'introspection soit présente dans la plate-forme et prise en charge de manière systématique et automatique. La spécification d'un type de composant semble une base idéale pour produire la prise en charge de l'introspection des futures instances de ce type de composant.

1.3.4 Composants et Architectures Logicielles

Pour pouvoir inter-opérer à l'intérieur de l'application, les composants doivent adhérer à une structure d'accueil, appelé en général *conteneur* de composants. Cette structure

d'accueil fournit un ensemble de services nécessaire pour accueillir les composants "contenus".

Nous remarquons qu'avec le développement à base de composants, nous pouvons créer des composants qui peuvent être indépendants, en dehors d'un contexte de conteneur et d'application. Toutefois, le développement à base de composants doit respecter les directives d'une infrastructure commune pour être amené à fonctionner à l'intérieur de n'importe quel conteneur supporté par cette infrastructure. Les termes "infrastructure commune" ou "architecture de composants" sont largement acceptés dans le développement à base de composants par plusieurs standards, comme l'architecture de ActiveX/COM [Dal96] de Microsoft, JavaBeans/Enterprise JavaBeans [Sun97, Sun] de Sun et CORBA CCM [JO01] de OMG.

Définitions

Nous donnons ici quelques concepts relatifs à l'architecture logicielle et des termes proches afin de présenter une définition à cette architecture. On peut apprendre beaucoup de concepts de l'architecture et des architectes dans le monde réel. On a besoin d'une architecture dans n'importe quel système assez complexe pour demander des règles pour sa conception et son implémentation. Pour cela et avant de présenter l'architecture logicielle nous donnons des définition qui aident à construire une terminologie architecturale :

- *L'architecture d'un système* est une structure organisationnelle du système qui comprend sa décomposition en sous-système, ses interfaces et ses règles de composition (assemblage) principes de guidage qui déterminent la conception du système.
- Une *architecture d'un système de composants* consiste en un ensemble de décisions de la plateforme, un ensemble de frameworks de composants et une conception d'inter-opération pour les frameworks de composants. La *plateforme* permet l'installation des composants et les frameworks de composants tel qu'ils peuvent être instanciés et activés.
- Un *framework de composants* est une architecture dédiée et concentrée, habituellement autour de quelques mécanismes clefs, et un ensemble fixe de politiques pour les mécanismes au niveau du composant.
- Une *conception d'inter-opération* pour les frameworks de composants comprend les règles d'inter-opération au travers les frameworks de l'architecture du système. Une telle conception peut être vu comme un framework de composants de second ordre où le premier ordre est le framework de composants et ses composants.

Définition et caractérisation d'une architecture logicielle

L'architecture [Szy00a] est une vue holistique d'un système qui n'est souvent pas encore développé. Techniquement parlant, une architecture définit les variantes globales (les propriétés qui caractérisent un système construit en suivant cette architecture particulière). Les systèmes d'exploitation sont un bon exemple d'une architecture qui classe par catégorie les ressources centrales pour permettre leur indépendance en présence de la concurrence de ces ressources. Une architecture a besoin d'être basée sur des considérations concernant la fonctionnalité globale, la performance, la fiabilité, et la sécurité. On peut laisser des

décisions détaillés ouvertes mais avec des règles pour prévoir des niveaux de fonctionnalité et de performance. Nous pouvons remarquer que comme dans la construction d'un bâtiment, des composants de bases comme les portes, les fenêtres et les murs sont assemblés en suivant un certain nombre de règles architecturales. L'architecte définit un ensemble de plans décrivant cet assemblage. Lors de la construction d'une application, l'idée est d'utiliser des composants existants et de les assembler en suivant un certains nombre de règles.

L'architecture d'un logiciel se définit en termes de composants et de leurs interactions. Elle spécifie la structure du logiciel et elle montre la correspondance entre les besoins et les éléments du logiciel construit. En plus, elle peut viser les propriétés au niveau du système comme l'efficacité, la performance, l'uniformité et la compatibilité du composant avec d'autres architectures.

Les architectures basées sur un modèle de composants [MM03, OMG01b] clarifient les différences structurelles et sémantiques à travers les composants et les interactions. L'architecture logicielle fournit des règles pour un atelier de construction de cette application que l'on appelle *framework* qui est détaillé dans la suite de cette section.

On dit qu'une application est conforme à une architecture logicielle si son architecture est conçue à partir des règles décrites par cette architecture logicielle. La description de l'architecture de l'application exprime comment ses composants doivent être organisés pour fournir les traitements requis. Les travaux relatifs aux composants et aux architectures logicielles sont complémentaires. Les premiers visent à fournir des briques de base de qualité, réutilisables pour la production des applications. Les seconds visent à l'intégration de ces briques pour construire une application résolvant un problème donné. Dans la suite de cette section, nous expliquons les notions introduites avec la définition d'une architecture logicielle.

Les architectures logicielles traditionnelles sont basées sur la notion des couches. Les couches et la décomposition hiérarchique restent très utiles dans les système à composants. Chaque partie du système y compris les composants eux-même peut être intégré à une couches. Les composants peuvent être situés dans des couches particulières d'une architecture plus grande. Pour mieux gérer la complexité des systèmes à composants plus larges, l'architecture elle-même a besoin d'être structurée en forme de couche. Il est important de distinguer les couches formées par une architecture et celles formées par une méta-architecture. Les couches formées par une méta-architecture sont celles que l'on appelle des *tiers*. Les applications client-serveur multi-tiers sont un bon exemple des architectures "n-tiers". Cependant, ces tiers proposés dans une architecture de composants sont différents de ceux que l'on trouve dans les architectures client-serveur.

Une architecture d'un système à composants, comme c'est définit ci-dessus, intègre un ensemble ouvert des frameworks de composants. La notion de tiers et de couches sont orthogonales. Chaque tiers a une architecture multi-couches. La figure 1.8 illustre l'effet des couches et des tiers dans une architecture trois-tiers multi-couches. Comme c'est illustré, les tiers plus haut fournissent des couches basses partagées avec les tiers bas. C'est une architecture de seconde-tiers dans laquelle chaque framework de composant introduit une architecture du premier-tiers. Il faut noter la différence radicale entre les tiers et les couches traditionnelles. Les couches traditionnelles, comme nous l'avons vu ci-dessus, sont de nature de plus en plus abstraites et de plus en plus spécifiques à l'application.

Dans un système multi-couches bien équilibré, toutes les couches ont leurs implications de ressources et de performance. Dans les architectures multi-tiers, la performance et la pertinence des ressources sont inférieur à celle qui sont multi-couches mais la pertinence de structure est supérieur. Des tiers différents mettent l'accent sur les degrés différents de l'intégration mais ils sont tous d'une pertinence d'application similaire.

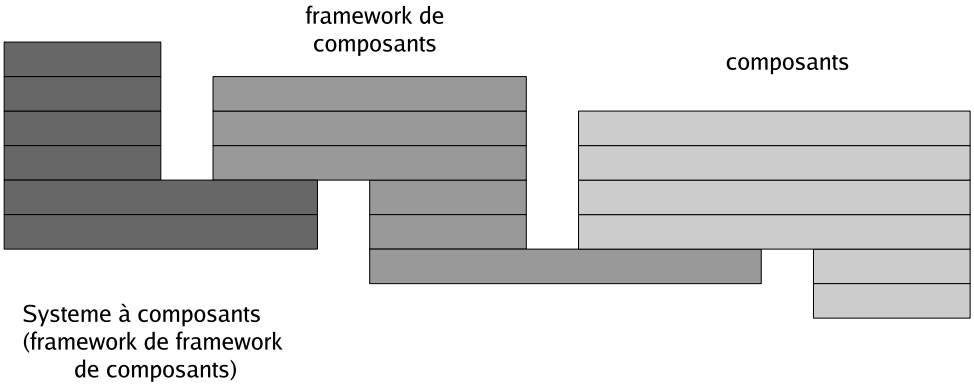

FIG. 1.8 – Une architecture multi couches avec trois-tiers - composants, framework de composants et système de composants

La réutilisation

La réutilisation de composants déjà existants est une des motivations les plus importantes de la construction des applications à base de composant. Cette réutilisation se fait en suivant certaines règles et directives d'architectures. Les architectes des logiciels combinent la théorie avec l'expérience fournit par d'autre applications qui ont été construites selon une architecture donnée. Dans une architecture logicielle à base de composants, il est important d'avoir une compréhension totale des techniques déjà établies de la réutilisation de conceptions. Il existe plusieurs approches qui couvrent tous les niveaux de granularité auxquels les architectes doivent faire face. On peut comprendre la réutilisation de la conception par le partage de certains aspects d'une solution à travers plusieurs projets. Nous pouvons citer quelques techniques déjà établies de la réutilisation selon des niveaux de partage comme :

- **Partage de la cohérence (les langages de programmation et de script)** : L'une des formes les plus anciennes de la réutilisation des méthodes prouvées est leur "casting" dans les langages de programmation ou les langage de scriptage. Un langage de programmation peut rendre facile, difficile ou impossible cette réutilisation. Cependant, un langage de programmation ne peut pas imposer la bonne conception mais il peut exclure des cas dans lesquels on peut faire des fautes de programmation. Donc, le langage de programmation établit une cohérence qui protège les efforts de programmation de certaines classes de fautes. Les langages orientés objets ont le plus de succès dans la programmation orientée-composants.
- **Partage des fragments de solutions concrètes (les bibliothèques)** : Les premiers langages de programmation ont essayé de fournir toutes les fonctions qui

pourraient être utilisées dans plusieurs programmes. En conséquence, le nombre de fonctions intégrées a été explosé comme dans PASCAL et C. Mais depuis, il y avait une tendance claire à mettre des fonctionnalités spécifiques en dehors du langage en faveur de dispositifs structurels et abstraits. Donc, il y avait la naissance des langages modulaires comme Modula. L'idée clef dans le partage des solutions prouvées est la constitution de bibliothèques modulaires et de boites à outils qui peuvent croître au fur et à mesure sans modifier le langage de programmation.

- **Partage des contrats individuels (les interfaces)** : Quand les fournisseurs d'un service et les clients de ce service existent séparément et sont combinés librement, le contrat qui lie les fournisseurs et les clients est important. Le fait d'attacher les contrats, formels et non-formels, à une interface nommée oblige les implémenteurs des fournisseurs et des clients à faire les vérifications nécessaires. Si les contrats sont soigneusement respectés, les fournisseurs et les clients des sources, totalement indépendants les uns des autres, peuvent interagir proprement. Les interfaces avec leurs contrats associés sont logiquement l'unité contractuelle la plus petite du système. Nous présentons plus en détails les interfaces et les contrats dans la section 1.3.4.
- **Partage des fragments d'interaction individuels (les messages et les protocoles)** : les interfaces résident dans un coté d'une interaction alors que les schémas de messages existent dans la ligne logique entre les parties qui sont connectées et qui interagissent entre elles. Les messages sont directionnels, c'est à dire qu'ils partent d'un expéditeur vers un destinataire. Les messages peuvent être auto-descriptifs en étiquetant le message par le nom du schéma du message. Les messages et les interfaces coexistent dans une relation subtile. D'un coté, les interfaces peuvent se restreindre en porteuse de messages auto-descriptifs et, de l'autre coté, elles ont des opérations qui prennent des arguments non auto-descriptif. C'est le cas de la programmation par API (*Application Programming Interface*).
- **Partage d'une architecture d'interaction spécifique (les patterns)** : Une tentative [EG95] a été faite pour collecter et cataloguer d'une façon systématique l'architecture la plus petite dans un système logiciel orienté objet pour que l'on puissent la reproduire. Le catalogue de *design patterns* (patrons de conception) est constitué et il définit les éléments de la réutilisation des paterns. Les quatre éléments qui définissent un pattern sont : son nom, le problème que résout le pattern, la solution qu'un pattern donne à un problème et les conséquences et les résultats de l'application de ce pattern. Dans leur catalogue, Gamma *et al* [EG95] ont définit 23 design patterns. L'idée de design patterns est que, dès que le problème est isolé, un pattern propre peut être sélectionné. Ensuite le pattern a besoin d'être adapté aux circonstances spécifiques du problème. Les design patterns sont des micro-architectures, ils décrivent l'interaction abstraite entre les objets qui collaborent pour résoudre un problème particulier. On peut citer comme exemples de design patterns l'observateur, le proxy , le subject, le factory ...etc.
- **Partage de l'architecture (les Frameworks)** : les frameworks ne sont pas nécessairement spécifiques au domaine mais ils sont habituellement spécifiques au concept. Les framework sont différents des design patterns. Les design patterns sont plus abstraits et moins spécialisés que les frameworks. Les frameworks sont une implémentation partielle d'une architecture alors que les patterns n'ont pas d'implémentation

immédiate. La plupart des frameworks utilisent plusieurs patterns. Les frameworks seront détaillés dans la section 1.3.4

- **Partage du système global (les architectures du système)** : une architecture du système logiciel a deux principes de conception : les couches et la hiérarchie. Les systèmes à couches multiples sont conçues soit à l'aide des couches strictes où l'implémentation d'une couche est basé seulement sur les opérations fournies par la couche d'en dessous, soit à à l'aide des couches non-strictes dans le cas inverse. Dans les deux cas, les couches d'un système peuvent être : l'application, les bibliothèques, le noyau OS, les drivers(pilotes) des périphériques et la couche du matériel.

Composition des Composants

Le développement de logiciels par composition (assemblage) des structures logicielles préfabriqués rencontre différents défis en utilisant des outils au niveau des langages de programmation. Ceci est vrai spécialement dans les systèmes distribués orientés objets parce que les objets ont des comportements plus complexes que les fonctions et les procédures. Les objets communiquent à travers les invocations de méthodes. L'invocation d'une méthode a besoin de deux informations : l'objet cible et la signature de la méthode. Dans les langages de programmation orientés objets fortement typés comme C++ et Java, l'objet cible (fournisseur de service) est identifié statiquement par son nom de classe ou son nom d'interface(l'appel de la méthode peut-être dynamique). Il existe un conflit fondamental dans les systèmes orientés objets entre l'encapsulation du comportement de l'objet et le besoin de connaissance sur les fournisseurs du service à l'extérieur de l'objet. Par conséquent, des dépendances fortes et implicites sont introduites non seulement entre les objets mais aussi entre les objets et le middleware qui représente la couche de communication des objets distribués.

La composition des composants préfabriqués implique l'établissement de communications collaboratives entre les composants. Pour les composants orientés objets, les communications entre les composants sont contraintes par le mécanisme fondamental de la communication objet à objet qui est appelée "invocation de méthode". Il existe plusieurs challenges pour la composition des composants [WUK99, GWK98] :

- Comment construire des composants tels qu'ils peuvent facilement se composer dans différents types d'applications ? Les mécanismes et les stratégies de composition des composants dépendent de la façon dont les composants sont définis, ce qui, à son tour, dépend de la modélisation et des paradigmes de programmation. Dans ce cas, seulement les parties du composant qui sont exposées sur leurs frontières sont utilisées pour la composition.
- Comment composer les composants qui proviennent de plusieurs fournisseurs sans modifier le code source ? La modification du code source est souvent impossible ou n'est pas économique.
- Comment identifier et lier les services requis par les composants que ce soit statiquement ou dynamiquement ? Il peut y avoir une différence substantielle de performance entre les choix d'identifications des méthodes et/ou entre les stratégies de liens sélectionnées pendant la conception des composants.
- Comment remplacer un composant par un autre qui a le même comportement ?

Les différents fournisseurs de composants peuvent utiliser des noms et des types différents pour l'étiquetage des services des composants. Avec les objets, les variables d'attributs sont irremplaçables sans modification du code source.

- pour les systèmes distribués, comment remplacer un protocole de communication ou un middleware par un autre (un autre différent, le même mais une version différente, le même mais une implémentation différente, etc.) sans modification des composants affectés par ce remplacement ? Il faut noter que différents protocoles et middleware imposent des prérequis différents (opérations, étapes, contraintes et autres détails) dans la façon le composant distribué est implémenté.
- Comment construire un système distribué tel que certaines qualités comme la fiabilité, la sécurité, la surveillance de performance et la tolérance aux fautes peuvent être ajoutées/enlevées/remplacées sans décomposer le système ? Des qualités comme celles-ci sont orthogonales à la fonctionnalité du composant.
- Comment établir le lien entre les composants dans une conception architecturale et les objets dans les implémentations distribuées orientées objet. L'architecture logicielle joue un rôle central dans le développement du logiciel à base de composants. A la différence des objets, les composants dans les logiciels architecturaux ont un ensemble plus riche de capacités et de sémantiques de description du composant.
- Comment utiliser des composants préfabriqués pour créer un nouveau composant et les déployer vu comme un seul (composant) ? A quel niveau de granularité de composition peut-on arriver sans affecter la performance, la décomposition et la réutilisation du composant ou de l'application.

Les composants sont les briques de construction des applications. Les applications doivent être conçues au début selon l'architecture appropriée. Les frontières des composants peuvent être spécifiées sur la base d'une conception architecturale. Pour cela, les défis de composition des composants doivent être traités d'abord au niveau de l'architecture logicielle. L'architecture logicielle d'un système s'intéresse à l'abstraction des composants et à leurs interactions pour réaliser les conditions du système.

Dans le contexte d'architecture logicielle, le composant a des contraintes à reparters-respecters en fournissant des services aux autres composants et naturellement quelques composants peuvent demander des services fournis par d'autres. Les mécanismes et stratégies de composition des composants dépendent de la manière dont les composants sont définis. La disponibilité des informations sur les frontières des composants va déterminer les possibilités de les composer avec d'autres. Naturellement, la frontière d'un fournisseur pour les services va consister en un seul élément : le service qu'il fournit. Dans ce cas la frontière du composant est représentée par une interface traditionnelle d'objet. La plupart des architectures utilise le terme *Port* pour ces interfaces. D'autres frontières de composant peuvent être une génération ou observation d'événement. Certaines architectures, utilisent des *Ports événementiels* ou des points de contacts dans le composant qui sont appelés *Générateur d'événements* et *Observateur d'événements.*

La composition des composants peut être faite à deux niveaux. Le premier consiste à composer les composants et à les connecter entre eux pour construire une application. Cette méthode est appelée la *composition contextuelle* qui se fait à l'aide d'un framework qui est responsable des interactions entre les composants. Le deuxième consiste à composer des composants pour avoir des composants à plus grosse granularité ou ce que nous

appelons un *super-composant* ou un *composant composite*. La construction d'un super-composant peut se faire au travers au modèle de composant et donc agir comme une propriété ou une configuration particulière d'un composant. Elle peut se faire aussi à travers le framework avec certaines fonctionnalités particulières qui transforment un ensemble de composants en un seul super-composant. Dans la suite, nous présentons les frameworks et plus particulièrement les frameworks contextuels dans lesquels on peut composer les composants.

Les interfaces, les contrats et interaction entre composants

L'indépendance est une raison majeure pour l'utilisation des composants. L'idée est simple : si le composant est construit avec des dépendances minimales et intégrées dans son environnement, il est susceptible d'être largement utilisable. Pour cela il faut avoir une conception qui prend en compte cette notion d'indépendance des composants. Nous avons besoin de savoir si nous pouvons utiliser un composant selon certains contextes. Cette information va prendre la forme de spécifications qui nous disent *ce que fait* le composant sans entrer dans les détails de la manière dont il fait. Les spécifications doivent fournir des paramètres selon lesquels les composants peuvent être vérifiés et validés et cela fournit un *contrat* entre le composant et ses utilisateurs.

Nous pouvons définir un contrat comme l'ensemble des spécifications attachées à l'interaction mutuelle entre le client et le fournisseur. Les contrats peuvent couvrir des aspects fonctionnels et extra-fonctionnels. Les aspects fonctionnels incluent la syntaxe et les sémantiques d'un interaction. Les aspects extra-fonctionnels sont relatifs à la qualité de services garanties.

Il est important de mettre l'accent sur l'interaction entre les composants en utilisant certaines interfaces. Cela peut être fait par l'écriture des spécifications qui mettent des contraintes sur le flux de données à travers un ensemble d'interfaces. Les spécifications des interfaces traditionnelles peuvent être vue comme un cas dégénéré des spécifications d'interactions en mettant l'accent sur les interactions entre deux parties à travers une seule interface.

Les interfaces sont un moyen de connexion entre les composants. Techniquement une interface est un ensemble d'opérations nommées qui peuvent être invoquées par des clients. La sémantique de chaque opération est spécifiée et les spécifications jouent un double rôle car elles servent les fournisseurs qui implémentent l'interface et les clients qui l'utilisent. Comme dans la configuration des composants, les fournisseurs et les clients s'ignorent et les spécifications de l'interface deviennent le médiateur qui permet aux deux parties de travailler ensemble.

Une méthode très utile pour voir les spécifications d'interfaces, sont les contrats entre un client d'une interface et un fournisseur d'une implémentation de cette interface. Le contrat énonce ce que doit faire un client pour utiliser une interface et ce que doit faire le fournisseur pour implémenter le service promis par l'interface. Le contrat comprend deux parties le client et le fournisseur. Donc, la spécification contractuelle d'une interface est réellement un modèle de contrat qui est instancié lors de la composition du client et du fournisseur. Théoriquement le contrat peut être raffiné et renégocié pendant la composition.

Les langages de définition d'interface IDL *Interface Definition Languages* ainsi que les langages orientés objets ou typés objets permettent au concepteur de spécifier :

- les opérations que le composant peut exécuter
- les paramètre d'entrées et de sorties dont chaque composant a besoin
- les exceptions possibles qui peuvent être levées pendant une opération.

Autrement dit, ils permettent de spécifier les contrats de base pour les composants. Les contrats sont aussi les spécifications qui permettent la composition entre les composants. Ces spécifications offrent un standard auquel les composants doivent se conformer. Les implémentations de composants doivent ensuite respecter ces spécifications. De toute façon, il n'existe pas aujourd'hui un langage qui permettent de spécifier des contrats. L'approche la plus proche est de mettre le support de l'IDL comme intermédiaire. Cette approche a été utilisée dès les premiers jours en Ada et est utilisé aujourd'hui par CORBA et COM. Ces interfaces sont le fruit d'une modélisation orientée objets.

Les langages de programmation peuvent être vus soit en perspective de temps d'exécution ou en suivant le point de vue de conception et de réutilisation. Le premier aspect vise à fournir la transparence pendant l'exécution d'un programme. Le second conduit à des concepts plus abstraits comme les classes, l'héritage et le polymorphisme ainsi qu'aux méthodes et à des notions comme UML [UML97]. La technologie à base de composants est l'extension logique du premier aspect. Le second aspect a été adopté par les ADL (*Architecture Description Languages*) . Les ADL [TC01] essaient d'établir un niveau plus élevé de conception et d'abstraction. La communauté ADL a mis l'accent sur la définition du composant, ses interfaces et ses relations avec les autres composants. Ils visent les premières phases du génie logiciel : la conception et l'analyse. Leur but est aussi d'améliorer l'indépendance entre les composants en fournissant des définitions explicites des interfaces et non simplement un protocole simple d'interconnexion.

Les Frameworks de composants

Un framework de composants est une entité logicielle qui assume la conformité des composants à certains standards et qui permet aux instances de ces composants d'être connectés dans le framework de composants. Le framework de composants établit des conditions d'environnement pour les instances de composants et régularise l'interaction entre ces instances. Les frameworks de composants peuvent être seuls ou peuvent coopérer avec d'autres composants ou avec d'autres frameworks de composants. Il est aussi important de posséder un framework d'un ordre plus élevé qui régularise l'interaction entre les frameworks de composants, ce que l'on appelle le *framework de frameworks de composants*, (voir la figure 1.8).

La contribution clef des frameworks de composants est de renforcer en partie les principes et les règles de l'architecture. Par exemple, le framework peut renforcer ces politiques de contrôle en ce qui concerne les instances de composants qui veulent réaliser certaines tâches. Les frameworks de composants implémentent souvent des protocoles pour connecter les composants et ils leur imposent l'ensemble de leurs politiques. Les politiques qui gouvernent l'utilisation des mécanismes utilisées par le framework lui même ne sont pas nécessairement fixées. Elles peuvent être plutôt mises par l'architecture d'un niveau plus élevé comme dans le cas de framework de frameworks de composants.

La plupart des efforts dans le domaine des frameworks ont été menés dans la composition contextuelle, c'est à dire dans les frameworks de composants, bien que les frameworks d'un ordre plus élevé paraissent aussi nécessaires et indispensables au long terme. Nous avons présenté la composition précédemment. Nous allons revoir la composition contextuelle avec le rôle du framework dans cette composition.

Un *contexte* est un ensemble de propriétés qui caractérisent les contraintes qui sont adoptés par tous les objets dans ce contexte. Si les objets, qui appartiennent à des contextes séparés, approuvent certaine propriété, on peut dire qu'il sont dans le même domaine (en respectant la propriété). Par exemple, un contexte peut avoir une propriété de synchronisation et une propriété transactionnelle. Il partage la propriété transactionnelle avec d'autres contexte mais pas la propriété de synchronisation. Deux objets dans ces deux contextes sont dans le même domaine transactionnel mais pas dans le même domaine de synchronisation (alors, ils peuvent être exécutés en même temps). La composition contextuelle permet la composition de plusieurs objets dans le même contexte.

Les éléments de la composition sont des instances créées par les composants. Le mécanisme qui supporte la composition contextuelle des composants peut être vue comme un framework. Il forme un framework particulier composant les instances qui ne sont pas basées sur des connections déclarées directement ou des dérivation (comme l'héritage d'une classe de framework), mais qui sont basées sur la création des contextes et sur le placement des instances dans le contexte approprié. Cette approche est utilisée pour créer des frameworks de sécurité, d'équilibrage de charge, de gestion, et d'autres propriétés.

La force de la composition contextuelle vient de trois effets :

- Les composants déclarent leurs contraintes dans un contexte acceptable mais ils ne construisent pas le contexte
- La composition suppose des contextes bien formés en minimisant de description et de traitement des conditions exceptionnelles. L'application des opérateurs de composition construit successivement les contextes.
- Les composites ont certaines propriétés par la construction comme l'utilisation d'une simple règle de composition.

Ces observations fondamentales peuvent être appliquées à tous les niveaux de l'abstraction ou de la granularité [L.L97].

Ces propriétés sont appelées parfois des *aspects* car elles ont tendance à couper le système selon des qualités qui ne sont pas établies par un composant seul dans le système. Il existe plusieurs sources d'approches académique pour l'utilisation de la composition contextuelle. Nous pouvons citer ici, les filtres de composition [AWB+93] et ses extensions en LayOM (*Layered Object Model*) [J.B96]. L'idée de base est de pouvoir co-contenir plusieurs objets de classes différentes dans un seul contexte ou un conteneur et de les rendre accessibles qu'à travers un objet passerelle. Cette idée est apparue entre autre approches de [Hog91, Sch95, Alm97, VB99]. Dans tous les cas, des objets spéciaux sont désignés comme visibles de l'extérieur de chaque ensemble d'objets alors que les autres objets dans cet ensemble ne le sont pas. C'est une forme générale de la composition contextuelle où les objets sont potentiellement accessibles de l'extérieur de leur contexte mais où les frontières du contexte peuvent intercepter tous les messages qui traversent les frontières.

Nous présentons ici les points importants des frameworks contextuels vis à vis d'autres

approches de compositions comme les connecteurs, les méta-programmation et la programmation orientée aspect, pour comprendre en quoi sont ils sont les plus répandus dans les architectures logicielles à base de composants. La relation entre les frameworks de composants et les autres approches est traité selon trois catégories d'approches :

- **Les connecteurs** : Les ADL (*Architecture Description Languages*) ou langages de description d'architecture [DK76, Mor94] distinguent les composants et les connecteurs. Les composants fournissent les fonctionnalités tandis que les connecteurs mettent l'accent sur les connections entre les composants. Comme partie de la description de l'architecture, des règles sont formulées pour dire quels connecteurs doivent supporter l'interaction et entre quels composants. Les connecteurs peuvent ensuite effectuer la synchronisation, le cryptage, l'authentification. Un connecteur peut facilement avoir une complexité substantielle et peut être lui même partitionné en composants. La connaissance du comportement fonctionnel peut être aussi nécessaire pour implémenter le comportement de connexion d'un connecteur. Donc il apparaît important de classer certains composants comme les connecteurs. Dans les approches orientées connexion, les composants sont contraints d'interagir seulement quand ils sont connectés proprement et donc ne laissent pas d'espace pour des actions spécifiques lors de la connexion. Par contre, avec l'utilisation des connecteurs, ils peuvent être des composants normaux et donc réaliser certaines actions lors de la connexion. Les premiers ADL ont été restreints à la connectivité statique. Plus tard, ils ont supportés les connectivités et la configuration dynamique. Dans [AT03] nous implémentons une approche qui utilisent les deux approches de connexion directe et à travers des composants spécifiques de connections.
- **La méta-programmation** : L'idée de la méta-programmation est de permettre à un programme (appelé un méta-programme) d'inspecter et de traiter un autre programme et son exécution. Le traitement systématique des capacités de la méta-programmation a conduit à l'introduction de la notion du protocole méta-objet MOP (*MetaObject Protocol*) [GdRB91]. La méta-programmation permet la notion d'hiérarchie dans l'application. Cela implique la modification des méta-programmes par des méta-méta-programmes. Smaltalk était un des premier exemple de systèmes avec une hiérarchie de méta-programmation. Le projet de l'OMG, MOF (*Meta Object Facility*) [OMG00] est une approche qui supporte quatre niveaux de modélisation. Il existe des propositions répétitives [VD99] pour utiliser la méta-programmation afin de séparer le code orienté aspect du code fonctionnel. Cela est réalisé par des méthodes d'interception qui utilisent les manipulateurs des pré- et post-invocations de méthodes (ou invocation handlers). Le langage Java est un langage qui utilise des propriétés de la méta-programmation grâce à la notion de la réflexivité.
  L'idée d'utiliser les interceptions systématiques des invocations pour séparer le code orienté aspect peut être généralisé à un ensemble d'objets. Il est important de souligner que cela n'est qu'une autre façon de décrire la composition contextuelle. Les infrastructures contextuelles actuelles ne supportent pas encore la méta-programmation hiérarchisée. Le problème de l'ordonnance et de la composabilité d'un contexte d'un unique niveau reste un problème important et rajouter encore des structures hiérarchique pourrait plus compliquer les choses et parait aujourd'hui non-praticable.
- **La programmation orientée aspects** : La programmation orientée aspects AOP

(*Aspect Oriented Programming*) [Kic92] est une tentative de factoriser le code spécifique à l'aspect et le code fonctionnel et de tisser (incorporer) les fragments du code résultant, dans le code actuel pour être exécuté. L'aspect est une partie d'un programme qui est indispensable mais qui n'est pas spécifique au domaine pour lequel le programme est écrit. Séparer les aspects (comme le logging, la persistance) de la logique de métier (fonctionnel) est le rôle de la programmation orientée aspect. Comme le tissage est un processus de pré-déploiement, l'AOP peut être vue comme un type de méta-programmation statique. Les capacités de réflexion et d'interception de la méta-programmation sont effectivement disponibles lors de tissage main et non pas lors de l'exécution. Les prédécesseurs de l'idée de l'AOP ont mis l'accent sur des aspects spécifiques : Par exemple, les couches de middlewares ou tels que DCOM et CORBA utilisent une IDL pour capturer séparément les aspects "distants" . Les compilateurs IDL génèrent les proxies nécessaires et ils sont effectivement les "tisseurs" des aspects distants. L'AOP peut être vue comme appartenant à la catégorie des approches génératives et il existe des interactions subtiles entre les approches orientées composants et les approches génératives. L'AOP, comme la plupart des approches génératives (programmation générative qui utilise des codes sources générés automatiquement à travers des classes génériques, templates ou aspect), marche mieux, si il est appliqué dans un composant. Donc, la manipulation des aspects dans un système à base de composants est laissée aux frameworks de composants (incluant les frameworks contextuels) qui comble le manque d'aspects entre les composants. Avec un tel scénario, les frameworks de composants (comme des artefacts pré-existants) interagissent avec le code aspect. Nous pouvons citer, comme exemples sur les projets AOP, le Xerox PARC qui a intégrér les AOP en java pour avoir le AspectJ [Asp05]. Un autre projet de recherche est celui d'IBM, appelé Hyper/J sur les hyperspaces [IBM01] et la réalisation liée au Java.

Dans cette section, nous avons discuté de l'impact des frameworks pour la composition contextuelle et des raisons pour lesquelles ils sont les plus répandus. Cela nous mène à explorer les technologies qui reposent sur ce type de composition et la commercialisation de ce type de framework. Les modèles industriels actuels de composants logiciels pour une architecture à base de composants sont basés sur la notion de composition contextuelle, autrement dit les modèles à base de conteneurs comme les modèle EJB, COM et CORBA qui sont décrits en détail dans la section suivante.

1.4 Les modèles de composants logiciels

Comme nous l'avons vu dans la section précédente, la construction des applications à base de composants logiciels, a besoin de spécifications et de règles standards pour la connexion des composants et leurs interactions. La standardisation à ce niveau est très importante pour que les composants soient connectables. Depuis longtemps, l'interopérabilité des logiciels était limité à des conventions d'appels binaires au niveau procédural. Chaque système d'exploitation définit des convention d'appels et tous les langages d'implémentation respectent les conventions d'appel de leur plateforme. Les appels de procédure avec leurs conventions d'appels binaires fournissent un standard de connexion entre les

morceaux logiciels.

Les invocations d'objets sont différentes des invocations procédurales surtout dans leur sélection du code à appeler, conduite par les données. L'appel d'une méthode inspecte la classe de l'objet qui reçoit l'appel et sélectionne l'implémentation fournie par cette classe. D'autres aspects ont besoin d'être standardisés pour réaliser l'interopérabilité : comment les interfaces sont-elles spécifiées ? Comment les références d'objets sont traitées quand elles quittent leurs processus locaux ? Comment les services sont-ils placés et fournis ? Comment l'évolution du composant est-elle traitée ? Toutes ces questions ont amené à définir des modèles de composants pour d'interopérabilité, la connexion et la construction des applications à base de composants.

Les modèles de composants logiciels peuvent être répartis selon deux catégories principales : les modèles industriels et les modèles de recherche (académiques).

Les modèles industriels sont des standards qui font l'objet d'une utilisation par la plupart des constructeurs des applications distribuées dans le monde des affaires, les banques, les systèmes d'informations distribués et les application Web. Nous pouvons citer sur cette catégorie les modèles des trois grands standards : DCOM de Microsoft [EE99], JavaBeans et Enterprise JavaBeans de SUN [Sun97, Rom99, Sun00] et CORBA CCM de l'OMG [OMG01a, JGP99].

Les modèles de recherche sont des modèles qui sont définis comme but de recherche dans les laboratoires ou comme modèles d'études pour des applications spécifiques de simulation ou de calcul scientifique à haute performance. Les modèles de composants pour des architectures logicielles sur grille(GRID) et le calcul scientifique à haute performance sont inclus aussi dans cette catégorie. Nous pouvons citer quelques modèles de composants académiques les plus répandus comme Darwin[BFT04], Durra[BWD+93] et JavaPod [BR00] ainsi qu'un modèle de composant qui nous intéresse dans la suite de cette thèse, le modèle CCA [AGG+99]. Dans la suite, nous détaillons les trois modèles standards proposé par Microsoft, SUN et l'OMG. Dans chaque modèle traité nous allons parler du modèle de composant, de l'implémentation et de l'assemblage et du déploiement des composants.

1.4.1 Java Beans et Enterprise Java Beans de SUN

Les premières spécifications Java étaient centrées sur les *Applets Java* tandis que la plate-forme Java 2 marginalise les applets et introduit la notion des éditions de plate-formes qui sont de sélections des spécifications Java qui concernent des catégorie d'utilisateurs de Java. L'édition de la plate-forme J2EE (*Java 2 platform Enterprise Edition*) en 1999 avait le plus du succès. Avec la spécification des EJBs (*Enterprise Java Beans*), J2EE devient la spécification qui est au dessus d'un grand nombre de serveurs d'applications. Beaucoup de vendeurs de serveurs d'application ont intégré J2EE dans leur serveur comme IBM avec ses produits de *WebSphere*, BEA avec ses produits de *WebLogic* et d'autres jusqu'à 40 vendeurs sur la liste en 2001.

Avec l'environnement java, SUN a sorti des modèles de composants logiciels pour le développement des logiciels Java à base de composants. Nous citons dans la suite les successions des modèles proposés par SUN sous l'environnement Java.

Java Beans

Le modèle Les JavaBeans [Sun97] se présentent plus comme un mode d'utilisation de l'environnement de développement Java que comme un modèle en soit. La spécification des JavaBeans définit clairement les Beans comme des composants graphiques. La définition donnée par SUN sur les JavaBeans est :

A JavaBean is a reusable software component that can be manipulated visually in a builder tool.

Comme tout objet Java, un Bean possède une interface de classe. Il peut disposer de méthodes pour offrir des traitements et d'attributs pour représenter ses propriétés (voir Figure). Du fait de l'orientation graphique des Beans, la coopération entre instances se base sur l'utilisation d'événements. Tout comme en Java standard, les événements sont mis en œuvre de manière synchrone.

Au niveau de l'interface, rien ne différencie un Bean de tout autre objet Java. Toute personne intéressée par les fonctionnalités d'un Bean, et ne connaissant pas les JavaBeans, pourra sans difficulté utiliser ce Bean selon le modèle objet traditionnel de Java.

L'utilisation de l'introspection sur un Bean permet de mieux souligner ses capacités : mise en évidence des propriétés, expression des ports, etc.

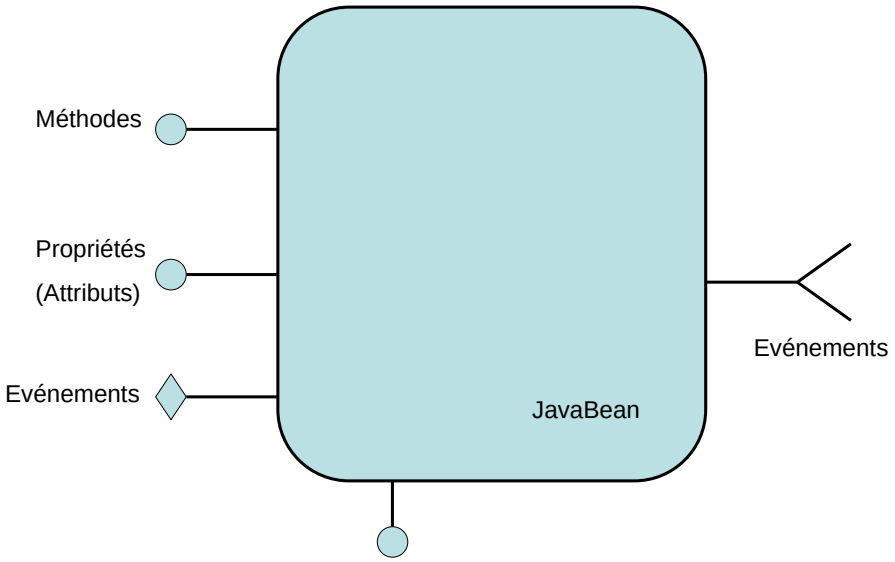

FIG. 1.9 – Modèle de composant JavaBean

L'implementation, la composition et le déploiment La définition d'un JavaBean se fait de la même manière que la définition d'un objet Java. Les règles supplémentaires sont principalement syntaxiques. Les attributs sont définis par la présence d'une ou deux

méthodes pour les lire et éventuellement les modifier. De la même manière une source d'événements est définie par la présence de deux opérations pour s'abonner et se désabonner aux événements. Un Bean consommateur d'événements devra implémenter une interface de consommateur d'événements (*java.util.EventListener* ou une sous-classe).

La spécification n'impose pas de définir des interfaces pour les Beans : il est possible de fournir uniquement l'implémentation. Le code suivant représente cependant une interface pour illustrer la définition d'un Bean.

```
interface MonBean {
// méthode pour lire l'attribut
     String getCouleur ( ) ;

     // gestion des événements
      void addMonBeanListener ( MonBeanListener l ) ;
      void removeMonBeanListener ( MonBeanListener l ) ;
}
```

La prise en charge de l'aspect distribué des Beans se fait au travers de RMI (*Remote Method Invocation*) dont nous avons parlé dans la section 1.2.2. RMI offre un moyen assez simple pour faire coopérer des objets Java répartis dans différentes JVM. Il permet d'invoquer des méthodes sur un objet distant en masquant la répartition.

Pour faciliter la recherche d'instances de composants logiciels dans un système, SUN propose l'utilisation de la technologie Jini [KA99]. Jini permet d'enregistrer des instances ou des implémentations de composants au sein d'un référentiel et offre une opération de recherche. Cette opération de recherche permet de retrouver une implémentation via son interface ou via une caractérisation de cette interface (utilisation de mots-clés).

L'environnement d'exécution des JavaBeans est simplifié au maximum : un conteneur graphique. Pour qu'un JavaBean puisse s'exécuter, il lui faut une fenêtre graphique (Frame, Browser Web, ...).

Dans le cadre des JavaBeans, la fourniture d'outils système se résume à la mise à disposition d'un environnement graphique complet pour le développeur (conception par composition, drag-and-drop, écriture de code, ...). Cet environnement doit permettre de positionner/lire les méta-informations sur le Bean, contenues dans BeanInfo. Ces méta-informations offrent au développeur une liste des services fournis par le Bean, guidant ainsi son intégration dans une application.

La fourniture d'un Bean se fait sous la forme d'un paquetage regroupant l'implémentation du Bean, son interface si elle existe et des informations sur le Bean contenues dans une classe de type BeanInfo. Ce paquetage s'exprime sous la forme d'une archive Java (.jar) qui peut être distribuée ou vendue.

Le déploiement des JavaBeans se fait en deux étapes. La première, manuelle, revient à rendre disponibles des archives de beans sur un serveur Web ou un système de fichiers. La seconde, automatique, correspond au chargement des beans dans un browser lors de l'accès à la page Web correspondante ou dans une JVM. La répartition des applications

tient essentiellement dans le fait qu'une partie cliente de l'application composée de Beans a besoin de dialoguer avec la partie serveur.

Enterprise Java Beans

De par leur définition, les JavaBeans représentent un modèle de composant client. Les JavaBeans n'étant pas en adéquation avec la réalisation de composants serveurs, SUN a mis au point les Enterprise Java Beans [Ble01, Rom99] (EJB) dont la spécification 1.0 est sortie en novembre 1997. Ce modèle représente un canevas de composants serveurs encore appelés composants *métier* pour concevoir des applications distribuées orientées transactions.

Le modèle Contrairement aux JavaBeans, un EJB est nécessairement représenté par une interface Java : l'interface distante du Bean qui définit la vue cliente de l'EJB. Cette interface hérite de plusieurs interfaces soit prédéfinies, comme *EJBObject*, soit définies par l'utilisateur, interface regroupant les opérations métier d'un EJB. La fourniture d'opérations *métier* est l'objectif de la conception d'un EJB. L'implémentation de ces opérations représente l'implémentation du composant.

En plus de cette interface métier, un EJB offre une interface pour accéder aux méta-données de l'instance de composant et une interface qui gère le cycle de vie d'un composant (*maison* ou *home interface*). L'interface *maison* permet de créer (ou rechercher dans le cas d'un composant persistant) et de détruire une instance de composant. Elle offre aussi une opération retournant une référence sur l'interface de méta-données. Cette dernière interface permettra par la suite de construire dynamiquement des requêtes sur le composant.

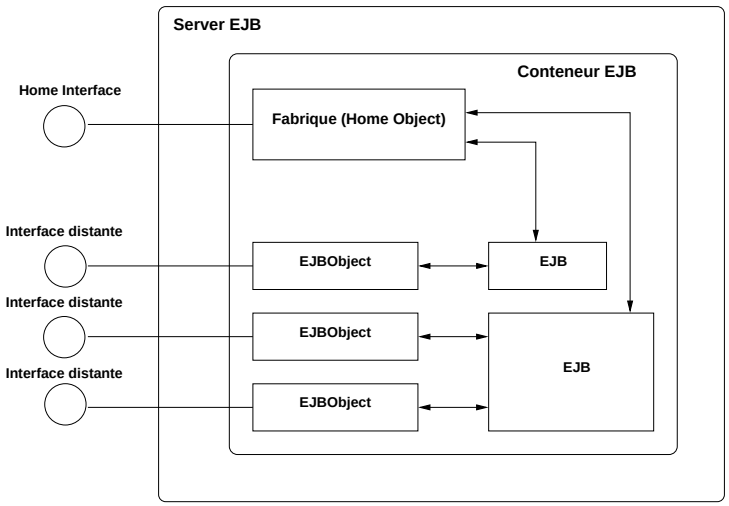

FIG. 1.10 – Modèle de composant Enterprise JavaBean

Les composants sont de deux types : composants de session et composants entités. Les instances des premiers sont créées à chaque connexion d'un client. Elles peuvent être avec ou sans état (composant de traitement). Lorsque le client termine sa session, l'instance

de composant est détruite. Les instances du second type disposent d'un identifiant unique qui permet au client de retrouver une instance particulière.

Implémentation, composition et déploiement La déclaration de l'interface distante d'un EJB se fait au travers d'une interface Java qui étend l'interface `javax.ejb.EJBObject`. La déclaration d'une maison de composant se fait de la même manière au travers d'une interface Java qui étend `javax.ejb.EJBHome`. Ces déclarations sont présentées dans le code suivant :

```
// Interface distante de l'EJB RepertoireBean
public interface RepertoireBean
                 extends javax.ejb.EJBObject {

   public void ajouterNom ( String nom )
                      throws RemoteException
   public String[] listeNoms ( )
                       throws RemoteException ;
}

// Interface de maison de l'EJB RepertoireBean
public interface RepertoireBeanHome
                extends javax.ejb.EJBHome {

   RepertoireBean create ( )
                      throws RemoteException, CreateException ;
}
```

L'environnement d'exécution se divise en deux parties : un ou plusieurs conteneurs (structure d'accueil) et un serveur (type serveur d'application). Les différents conteneurs sont présents dans le serveur. Plusieurs serveurs peuvent être mis en œuvre pour une application, mais en général il semble qu'un seul serveur par hôte soit le bon choix (un serveur peut être multi-threades).

Un utilisateur ne dialogue pas directement avec une instance d'EJB. Le client utilise une interface distante (remote interface). Cette interface délègue les requêtes à l'instance d'EJB. Cette délégation est prise en charge par le conteneur qui a un rôle de médiateur. Ce rôle correspond à la prise en charge des propriétés non fonctionnelles d'un EJB comme la persistance.

Un conteneur peut recevoir plusieurs types d'EJB. Comme le montre la figure 1.10, l'architecture de l'environnement d'exécution est en couches : les EJB, les conteneurs et les serveurs. Les conteneurs interviennent dans les échanges entre client et instance d'EJB alors que les serveurs sont uniquement mis en œuvre pour héberger les conteneurs. Les conteneurs sont localisés par l'application via l'API JNDI (*Java Naming and Directory Interface*) pour pouvoir être par la suite utilisés. Un EJB présente deux interfaces à ses clients : une interface de maison et une interface distante offrant l'accès à ses traitements.

Dans le but de faciliter leur déploiement, les Beans sont décrits dans un descripteur qui regroupe d'une part des informations générales sur le fournisseur du Bean, mais aussi une description des besoins du Bean en termes de ressources : gestion des transactions, politique de sécurité, ressources systèmes. La spécification précise qu'un Bean ne doit jamais accéder directement aux ressources système, mais à des objets encapsulant ces ressources. Pour utiliser ces ressources, les instances de Beans s'adressent à des fabriques de ressources pendant la phase d'exécution.

La recherche des ressources de l'environnement par un Bean est programmée au sein de son implantation. Dans le but de pouvoir fournir ces dépendances au Bean, chacun de ses points d'entrées vers l'environnement est décrit au sein du descripteur de déploiement. La fourniture de ces dépendances est à la charge de l'installateur de Bean au moment du déploiement.

L'implémentation au sens programmation d'un EJB doit se résumer à l'implémentation de la partie fonctionnelle (la logique métier). Il est important qu'un développeur se concentre sur cette partie qui représente toute la valeur de l'implémentation du composant. Dans ce sens, les aspects comme la sécurité et la gestion des transactions sont fixés dans le descripteur de déploiement et seront pris en charge par le conteneur lors de l'exécution. La persistance est un autre aspect qui peut être pris en charge par le conteneur.

Ces différentes interfaces et implémentations sont reprises par le fournisseur de composants. Il va packager ces différents éléments au sein d'une archive (.jar). Pour permettre l'utilisation d'un EJB, le fournisseur produit aussi un descripteur de Bean.

La composition d'EJB se fait à partir des packages qui comme pour les JavaBeans se présentent sous forme d'archives (.jar). Cette tâche débouche sur la production d'une ou de plusieurs archives de Beans. Pour chaque nouvelle archive, un descripteur de l'archive est produit, ainsi que les instructions de déploiement contenues dans les descripteurs de déploiement.

Le déploiement d'applications à base d'EJB paraît assez statique. Une fois que les environnements d'exécution sont opérationnels, les packages sont installés ainsi que leurs dépendances qui sont exprimées dans les descripteurs de déploiement. La phase de déploiement comprend aussi une part de génération de code. Les implémentations des interfaces maisons et des interfaces distantes sont produites à partir d'outils associés aux conteneurs et fournis avec.

L'administrateur système se charge de rendre opérationnel les serveurs et les conteneurs d'EJB. Son rôle est de configurer et de maintenir ces éléments pour que l'installateur déploie les applications et pour que les clients puissent les utiliser. Il doit aussi maintenir et monitorer l'exécution des applications déployées.

Les EJB restent une solution propriétaire de Sun, uniquement destinés à une utilisation en Java. Le faible nombre de propriétés non-fonctionnelles reflètent certainement une demande du marché, mais qu'en est-il de l'ajout de nouvelles dans le modèle ? Un dernier regret quant à la mise en œuvre est l'enfouissement de la connectique au sein même des implémentations. L'expression des besoins par description résout en partie le problème de la disponibilité des besoins à l'exécution, mais est-il réellement souhaitable qu'un composant gère sa connectique, et donc que cette gestion soit figée ?

1.4.2 ActiveX, DCOM, COM+ et .NET de MICROSOFT

COM (*Component Object Modèle*) [Dal96] est une extension du modèle OLE (*Object Linking and Embedding*). Ce dernier est un outil pour la manipulation de "documents composés" comme des composants. Il a résulté de l'évolution progressive de l'environnement MS-Windows vers les composants. COM a été créé pour permettre la communication entre les applications Windows dans un environnement à base de composants. DCOM (*Distributed COM*) [EE99] est une extension du modèle COM. Il a étendu les communications interprocessus permis par le modèle COM pour que ce soit au travers d'un réseau. Il se base sur le protocole DCE RPC pour implémenter un système d'appel de procédures vers des objets distants.

Les contrôles OLE, renommés ActiveX, sont des composants autonomes permettant de réaliser des applications. Ces composants s'appuient sur COM et OLE Automation que nous venons de voir. Les contrôles OLE de première génération, appelés contrôles OCX (OLE Control eXtension), étaient des composants pouvant réagir à des événements extérieurs. La deuxième version de ces contrôles, désormais appelés contrôles ActiveX étend les possibilités d'OCX en permettant d'utiliser des composants distribués sur plusieurs systèmes interconnectés par un réseau tel qu'Internet, exploitant le modèle DCOM, extension de COM, que nous décrirons plus loin. Une application incorporant des contrôles ActiveX est appelée un document. Notons qu'un même document peut contenir des contrôles de types très différents comme des tableaux Excel ou des documents HTML.

Le Modèle Dans le cadre de COM, un composant est essentiellement une entité binaire pour laquelle sont définis une interface et le mode d'interaction. Ces deux éléments se résument en fait en un simple pointeur, offrant accès aux fonctions offertes par la bibliothèque, implantation du composant. COM n'impose pas de contrainte sur l'implantation du composant : le composant peut être implanté sous la forme d'une ou plusieurs classes, sous la forme d'une bibliothèque de procédures ou de fonctions, etc. Un composant peut, par exemple, avoir plusieurs interfaces. Une interface est nécessairement disponible sur un composant, `IUnknown`. Cette interface offre une opération `QueryInterface` qui permet de découvrir les interfaces fournies, et ainsi, de naviguer entre elles.

Dans le but d'une utilisation générique, un composant peut fournir l'interface `IDispatch`. Cette interface regroupe l'utilisation de toutes les opérations d'un composant sous une unique opération : `invoke`. Cette opération permet l'utilisation systématique et dynamique d'un composant à partir d'un langage interprété comme Visual Basic.

De manière optionnelle, un composant peut déclarer des *outgoing interfaces*. Ces interfaces, sont des interfaces utilisables par le composant. Ces interfaces sont exploitées par le composant s'il est connecté à des instances de composants offrant une de ces interfaces (ou une interface dérivée). Ces interfaces sont principalement mises en œuvre par une instance de composant pour fournir de l'information soit au travers d'événements, soit par invocation directe. L'utilisation de ces interfaces peut être faite de manière dynamique grâce à l'utilisation de l'introspection sur les points de connexion, permettant la connexion et la déconnexion dynamique des composants consommateurs.

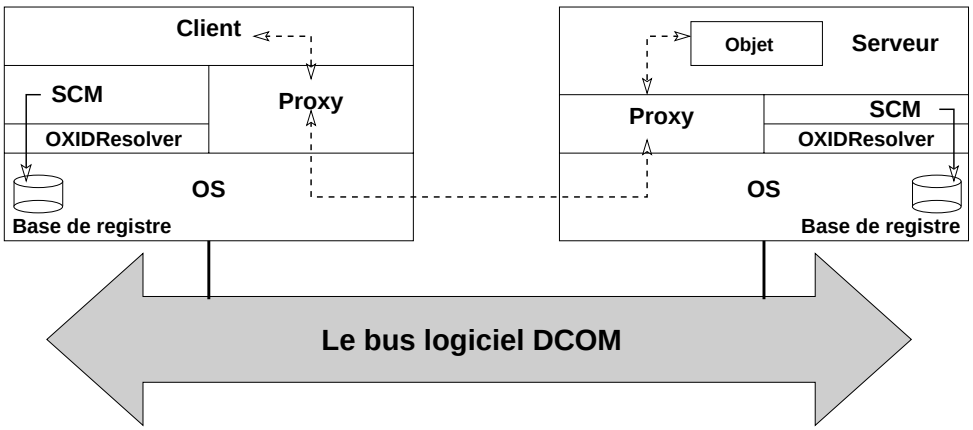

FIG. 1.11 – L'architecture DCOM

Implémentation et composition L'implémentation d'un composant COM se résume à la fourniture d'une bibliothèque offrant une ou plusieurs interfaces et implantant le comportement de ces interfaces. Ces bibliothèques se traduisent en général sous forme de DLL (Dynamically Linked Libraries). La notion de container pour héberger des instances de composants n'existe pas en COM. C'est en fait l'environnement, essentiellement MS-Windows, qui tient le rôle de container. La création d'instances de composant se fait au travers du patron de conception [EG95] *Fabrique*. Un fabrique est offerte par un serveur de composant. Lors de l'insertion d'un serveur de composant dans un système, ce serveur est enregistré auprès d'un registre système défini par COM, qui maintient une liste des serveurs de composants disponibles.

La réutilisation en COM ne se traduit pas par l'héritage d'implémentation (qui est possible en COM+, l'évolution de COM). Ceci s'explique par le fondement même de COM qui est de ne faire aucune spéculation sur l'implémentation des composants, qui ne sont pas nécessairement des objets. Le seul héritage possible en COM est l'héritage d'interface. Une interface A hérite d'une interface B si l'interface A contient toutes les opérations définies dans B et en propose des nouvelles. En cela, le polymorphisme au sens de COM est le fait de proposer un ensemble d'interfaces.

Les deux modes de réutilisation d'implémentations en COM sont la composition (appelée containement) et l'agrégation. Dans le cas de la composition, un composant dispose d'une référence exclusive sur un autre composant. Il s'agit de la réification de la notion de contenance. Le composant contenu (composant interne) est invisible des usagers qui ne sont conscient que du composant contenant (composant externe). Cette première possibilité est la méthode la plus simple et la plus courante pour faire de la réutilisation en COM.

Dans le cas de l'agrégation, il n'y a plus de notion de contenance entre les deux composants. Les deux composants doivent en contrepartie collaborer. Un composant peut "choisir" s'il accepte de faire partie d'un agrégat ou pas. Pour ce qui est de la vision cliente de l'agrégat, toutes les interfaces de composants sont visibles au même titre. En effet, un agrégat se comporte comme un composant multi-interfaces. Les clients utilisent donc directement les composants constituant l'agrégat. L'utilisation de l'agrégation est

un bon choix pour produire des *wrappers* de manière assez simple et automatique.

DCOM est une extension de COM pour prendre en compte l'aspect distribuée, principalement multi-processus, d'une application. Dans ce sens, DCOM définit la génération de *stubs* pour chaque interface d'un composant prenant en charge la partie communication avec un *proxy*. Ce proxy est utilisé par tout client du composant, qui pense utiliser une instance locale de composant. Pour la partie cliente, l'utilisation d'une instance de composant distante est transparente. La génération des stubs et proxies repose sur l'utilisation du langage de définition d'interfaces COM IDL.

COM+ est quant à lui une extension incluant un modèle d'objets légers (*lightweight object model*) à COM dans le but de prendre en compte des spécificités du langage Java. COM+ est un modèle uniquement utilisable dans un contexte intra-processus. Une utilisation inter-processus implique l'utilisation de DCOM. L'utilisation de COM+ étant fortement liée au langage Java, l'interface *IDispatch* disparaît de ce modèle (pour réapparaître sous la forme de l'API *reflect* de Java).

Le déploiement Les outils système de COM pour la production d'applications se résume essentiellement à un environnement graphique de développement comme MS Visual Studio. Cet environnement adresse l'ensemble des besoins pour produire une application : production de librairies, d'exécutables, assistants pour la génération automatique de code, etc.

La diffusion d'implémentations de composants se fait principalement au travers de bibliothèques (DLL MS Windows). Ces bibliothèques sont à déployer manuellement sur les machines cibles, car aucun processus de déploiement n'est prévu dans COM, si ce n'est à l'aide de solutions ad-hoc. D'autre part, la gestion des versions des bibliothèques n'est pas proposée, et chaque fournisseur doit mettre en œuvre sa propre politique.

A partir de sa disponibilité sur un système, une bibliothèque est utilisable par tout programme. Il est cependant difficile d'utiliser une bibliothèque sans sa documentation associée (non auto-contenue). La composition de composants est principalement basée sur la possession de références sur la bibliothèque contenant son implémentation. Cette opération se doit d'être simplifiée par l'utilisation d'un environnement comme MS Visual Studio.

Le framework .NET Les composants .NEt sont la dernière évolution que microsoft a fait sur son modèle de composant. .NET est plutôt un espace qui comprend le CLR (*Common Language Runtime*), des frameworks partiellement interfacés et partiellement basés sur les classes et empaquetés dans des *assemblées* et un ensemble d'outils. Le CLR est une implémentation des spécifications de la CLI (*Common Language Interface*) en rajoutant l'interoperation de COM+ et les services d'accès aux plates-formes Windows. CLR offre en particulier des possibilités de chargement et déchargement dynamique, de garbage collection , d'interception de contexte, de réflexion des méta-données, de l'accès distants, de la persistance et d'autres services temps réel et qui sont totalement indépendant du langage. Maintenant, Microsoft supporte quatre langages sur CLR : C#, JScript, Managed C++ et Visual Basic .NET. Les *assemblées* sont les unités de déploiement, de versionnement et de gestion dans .NET, autrement dit, ils sont les composants logiciels

.NET.

D'un point de vue technique .NET vise trois niveaux :

- les services Web ;
- la plate-forme de déploiement (serveurs et clients) ;
- la plate-forme de développement.

Les services Web visent la capacité de programmation de l'internet. Cette programmation d'Internet est différente du Web traditionnel. Pour cela Microsoft a introduit des services comme .NET passport et .NET alerts et d'autres initiatives. Les plate-formes Microsoft sont plutôt des produits de serveur de *Windows .NET server*. Ces serveurs sont transformés pour qu'il soit possible d'intégrer et supporter les services Web et de traiter XML. La plate-forme de développement comprend les CLR, les frameworks et les outils. Les CLR contribuent à fournir une nouvelle infrastructure de composants qui protège le composant de la plate-forme matérielle. Comme JVM, CLR définit un ensemble d'instructions virtuelles pour s'isoler des processeurs particuliers. A l'inverse de JVM, CLR permet une construction des composants qui peuvent être liés fortement à la plate-forme matérielle.

Avec les dizaines de millions de stations de travail tournant sous le système d'exploitation de Microsoft et sachant que chacune de ces stations est susceptible d'abriter des application DCOM et .NET, ces derniers peuvent être considérés comme le système le plus répandu dans le monde et sera sûrement utilisé longtemps encore.

1.4.3 CORBA CCM de l'OMG

L'utilisation d'objets répartis avec la technologie CORBA [Dan02] de l'Object Management Group (OMG) n'a pas permis d'atteindre la simplicité escomptée pour concevoir des applications distribuées à base d'entités logicielles hétérogènes et multi-fournisseurs. Pour faciliter et augmenter la qualité du processus de production de telles applications, l'OMG a défini le CORBA Component Model (CCM)[OMG99] , un modèle de composants logiciels serveurs répartis basé sur une technologie de conteneurs similaire à ceux des EJB.

Le modèle Le modèle abstrait des composants CORBA vise essentiellement deux choses. Premièrement, un type de composant décrit l'extérieur d'une boîte noire dont on ne sait rien de l'implantation. Deuxièmement, seule l'implantation fonctionnelle devrait être à programmer et tout le reste doit être décrit. Le premier point met en avant la volonté de rendre explicite toute interaction avec ou à partir du type de composant. Dans ce sens, un type de composant ne se limite pas à définir les services qu'il fournit, mais aussi les services qu'il utilise. Le second point tend à faciliter la production et à accroître la réutilisabilité des implantations de composants. La description sert de base à la génération automatique de la prise en charge des aspects non-fonctionnels.

Dans le cadre du modèle abstrait de composant tel qu'illustré dans la figure 1.12, un composant est potentiellement multi-interfaces. La partie gauche de la figure présente les différentes interfaces fournies par un type de composant. Ces interfaces peuvent être de deux types : des facettes, qui offrent un mode de coopération synchrone, et des puits d'événements, qui offrent un mode de coopération asynchrone. Chaque facette regroupe

une partie des opérations disponibles sur le type de composant et sont sémantiquement équivalentes à des vues sur le type de composant. A partir de la référence de base d'une instance de composant, il est possible de naviguer entre les différentes facettes au cours de l'exécution.

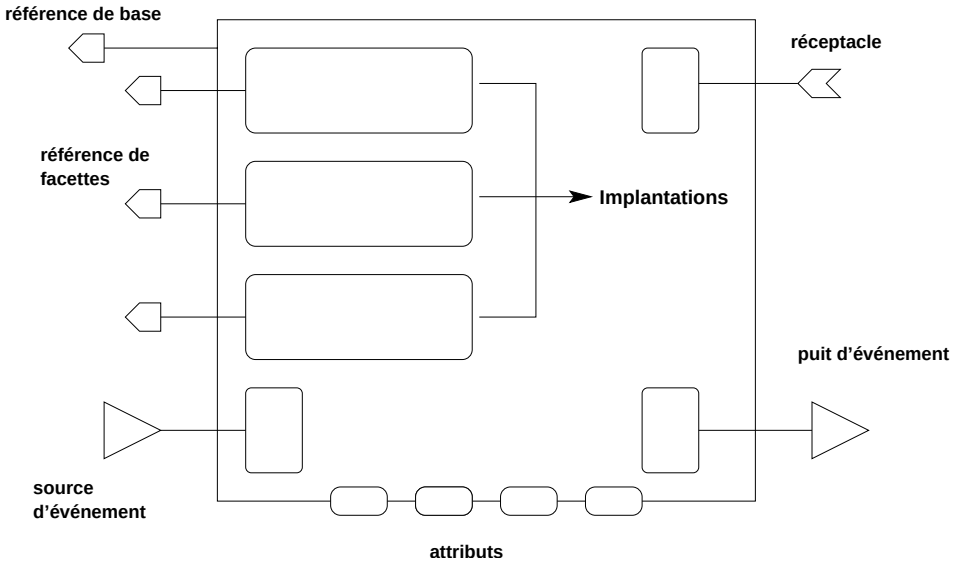

FIG. 1.12 – Modèle de composant CCM

La partie droite de la figure 1.12 présente les interfaces utilisées par un type de composant. De même que pour les interfaces fournies, deux types de coopération sont disponibles : l'utilisation synchrone de types de composants tiers au travers des réceptacles, et l'utilisation asynchrone au travers des sources d'événements. L'utilisation d'événements se fait selon le modèle subscribe/publish de CORBA. Les communications asynchrones peuvent être de deux types : un vers un, coopération privée entre deux types de composants, ou un vers n, coopération mettant en jeu plus de deux instances de composants.

Les implémentations de facettes et de puits d'événements de la figure 1.12 représentent l'implémentation fonctionnelle d'un composant. Cette partie de l'implantation est la seule à programmer. Le reste de l'implantation comme la prise en charge des points de connexions (aussi appelés ports), les aspects persistance, transactionnels et sécurité d'un composant sont décrits, à l'aide du langage CIDL (*Component Implementation Definition Language*), et générés automatiquement.

Pour chaque type de composant est défini un (ou plusieurs) type de maison de composant. Une maison de composant est un gestionnaire d'instance construit autour des deux patterns [EG95] : Fabrique et Recherche. Ce gestionnaire prend en charge le cycle de vie des instances de composants.

Pour permettre la définition des types de composants, le langage OMG CIDL est l'extension de l'IDL pour prendre en compte les nouveaux concepts comme les ports. Cette extension permet de définir, comme illustré par le code suivant, les différentes facettes (mot-clé provides), les puits d'événements (mot-clé `consumes`), les réceptacles (mot-clé `uses`) ainsi que les sources d'événements (mot-clé `emits` pour une coopération un-vers-un et `publishes` pour une coopération un-vers-n).

```
component Distributeur {
    provides FacetteClient      client;
    provides FacetteFournisseur fournisseur;
    consumes TemperatureEvt     temperature;
    uses     PriseCourant       courant;
    emits    VideEvt            vide;
}
home DistributeurHome manages Distributeur { };
```

Le type de maison `DistributeurHome` est déclaré comme prenant en charge le type de composant Distributeur. L'opération de base `create()` sera automatiquement générée pour permettre la création d'instances de composants de ce type. Cette description en IDL étendu n'est utilisée que pour exprimer le résultat de la conception. Pour être utilisé, cette déclaration est projetée en IDL telle que défini dans la spécification CORBA 2. Les habitudes des développeurs ne sont ainsi pas modifiées.

Pour permettre une bonne intégration des parties fonctionnelles et non-fonctionnelles de l'implémentation d'un composant, CCM fournit un canevas, le *Component Implementation Framework* (CIF), qui spécifie comment les deux parties de l'implémentation doivent coopérer. Ce canevas fixe d'autre part comment est généré la partie non-fonctionnelle. Pour permettre cette génération, le CIF s'appuie sur un langage déclaratif, le *Component Implementation Definition Language* (CIDL), qui permet de décrire les aspects non fonctionnels d'un composant.

Comme dans le cadre des EJB, les composants CORBA s'exécutent dans un conteneur qui leur offre d'une part un environnement d'exécution (espace mémoire, flot d'exécution), et d'autre part les services systèmes nécessaires au composant. La complexité de ces derniers est masquée au développeur d'implantations de composants, qui ne se préoccupe plus de maîtriser ces aspects. Les conteneurs se déclinent selon différents types de base, mais sont génériques par rapport à une même famille de composant (service, session, processus, entité).

Les échanges entre instances de composant et conteneur sont définis par des API standardisées. Le conteneur offre les interfaces des aspects système, comme la persistance et les aspects transactionnels, aux instances de composant et les instances de composants disposent d'une interface de callback permettant au conteneur principalement de gérer la persistance de ces instances de composant.

Le déploiement, l'implémentation et la composition Dans le but d'automatiser au maximum la phase de déploiement des applications à base de composants CORBA, la spécification décrit le langage *Open Software Description* (OSD) pour fournir les informations nécessaires au déploiement. Ce langage spécifié par le *W3C* et étendu par l'OMG est défini comme un vocabulaire XML (*eXtensible Markup Language*). Il permet de décrire

d'une part les besoins des implémentation de composants (persistance, sécurité, transactionnel) et d'autre part les compositions de composants. L'architecture d'une application est ainsi décrite de manière abstraite et pourra être projetée sur un ensemble de ressources physiques lors du déploiement.

L'expression de la conception d'un type de composant se fait à l'aide du langage CIDL. Ce langage de description va permettre d'exprimer les services fournis par un composant ainsi que les services requis par un composant. La connectique entre instances de composants est exprimée sémantiquement (par des mots clés réservés) et non syntaxiquement (par des règles de nommage des méthodes comme les JavaBeans).

L'implémentation d'un type de composant tel que défini par la phase de conception se découpe en deux étapes : la projection de la définition accompagnée de la description en CIDL vers le langage de programmation pour fournir les squelettes prenant en charge les aspects système, puis la programmation de la partie fonctionnelle. A l'implantation d'un composant s'ajoute sa description. En partie générée à partir de l'IDL et du CIDL, elle est complétée par le développeur et contient les contraintes techniques de l'implantation (OS, JVM, librairies requises, etc.).

La diffusion d'implémentations de composant se fait au travers de paquetages. Ces paquetages sont définis comme des archives "ZIP" qui regroupent les interfaces du type de composants, une ou plusieurs implémentations (par exemple dans différents langages de programmation), le descripteur de l'implémentation de composant et le descripteur du paquetage (décrivant son contenu). Ces paquetages pourront alors être diffusés à des architectes ou bien déployés dans un système et utilisés par des applications.

La réalisation de la composition se base sur l'utilisation des paquetages de composants offrant les types requis par l'application. Ces paquetages de composants sont assemblés entre eux grâce à l'exploitation des descripteurs. Une fois les compositions mis enœuvre, l'outil de la composition doit permettre de générer un paquetage de cette composition regroupant un descripteur de la composition (écrit en OSD) ainsi que les différents paquetages de composants mis en œuvre ou un lien vers leur localisation et un descripteur de la configuration globale par défaut. Ce paquetage de composition devient alors une application ou une nouvelle unité de composition réutilisable.

Une fois les applications conçues, réalisées et packagées, elles sont diffusables (commercialisables). Pour permettre leur déploiement et leur utilisation, il est nécessaire de mettre en place un ou plusieurs serveurs de conteneurs sur les sites potentiels. Une fois disponibles, ces conteneurs servent de base au déploiement des applications. Cette opération est réalisée à partir d'une console de pilotage du déploiement qui permet de déployer une application définie de manière logique dans un descripteur sur un ensemble de sites cibles choisis par la personne responsable du déploiement. L'ensemble de ce processus peut être automatisé (aucune intervention du responsable du déploiement si ce n'est le choix des sites).

Finalement, un des apports essentiels de CCM est l'ensemble des moyens fournis pour décrire les types, les implantations et les compositions de composants. Cette proposition fait un grand pas vers la non programmation des applications : décrire au mieux pour programmer au minimum. Cette puissance d'expression prend tout son sens sur les aspects non-fonctionnels des composants qui peuvent, pour une même implantation, varier en fonction du contexte d'exécution. Le second atout majeur des composants CORBA est le

fait d'expliciter les dépendances des composants, en terme de composants et de ressources systèmes, pour permettre une composition dynamique des instances. Ensuite, le modèle de déploiement est une première réponse satisfaisante, dans le sens automatisation suffisante, au problème du déploiement des applications distribuées.

1.4.4 Comparaison entre Java, CORBA et COM

Les modèles présentés ci-dessus qui appartient au trois environnements JAVA, CORBA et DCOM, sont actuellement les plus largement utilisés dans l'industrie et chacun possède des avantages et des inconvénients. Ils ont été développés avec des philosophies différentes. Il existe plusieurs critères pour comparer les trois technologies sur lesquelles les modèles présentés ci-dessus s'appuient. Voici une liste non-exhaustive des différences significatives entre les approches :

- **Le standard d'interfaçage binaire par plateforme** :
  Le cœur de COM est le standard de l'interfaçage binaire. Les interfaces dans COM/DCOM sont spécifiées au niveau binaire et vues par le client comme un pointeur vers une table de fonctions virtuelles(comme en C++). Il faut remarquer que COM n'a jamais quitté l'environnement Windows malgré toutes les tentatives pour le faire. Java évite le standard binaire par la standardisation du bytecode. Pour l'interfaçage binaire, Java définit le JNI (*Java Native Interface*) dont la conception est basé sur COM mais qui est spécifique à Java. Il est particulièrement conçu pour créer un espace pour les ramasses-miettes *garbage collecteur* modernes. CORBA n'a pas encore défini un standard binaire. Les standards binaires sont demandés par les compilateurs "Direct-to-*" qui projettent les structures d'un langage spécifique directement en interfaces binaires.
- **Les standards pour la compatibilité et la portabilité de source** :
  CORBA est particulièrement performance pour la standardisation de liaisons des langages de programmation qui assure la compatibilité à travers les implémentations de l'ORB. Le grand nombre d'interfaces de services standardisés consolide cette position. Les pratiques courantes de l'accès aux fonctions spécifiques à l'ORB au coté du serveur d'objets réduisent la portabilité des serveurs basés sur CORBA. Pour Java, l'accord sur les spécifications de Java résout le problème tant que l'on n'utilise pas d'autres langages qui visent la plateforme Java. L'espace Java couvre un ensemble large de standard *de facto* de SUN. Par exemple, le standard de J2EE a été implémenté par une douzaine de composants. COM n'a aucun concept de standard de niveau source ou de standard de liaisons de langages. L'interfaçage COM n'est pas standardisé au delà des standards *de facto* de Microsoft.
- **Gestion de mémoire, cycle de vie et ramasse-miettes (garbage collection)** :
  CORBA ne fournit pas aujourd'hui de solution à la gestion de la mémoire globale dans les systèmes à objets distribués. COM/DCOM dépendent du comptage de références en partant du bas vers le haut, mais cela implique un problème dans les systèmes distribués à grande échelle. Java dépend totalement sur le ramasse-miettes. Avec Java RMI, Java définit un modèle objets distribués et supporte un ramasse-miettes distribué où le concept est basé sur le cycle de vie de références distantes. Le CLR de COM fait aussi du ramasse-miettes combiné avec une invalidation basée

sur les taux des références à distance. En plus, CLR supporte un ensemble ouvert d'autres protocoles de communication comme SOAP sur HTTP.

- **Gestion par les conteneurs de la persistance et des relations** :
  EJB a introduit l'espace de la persistance et les relations gérées par les conteneurs. CCM a suivi cette évolution puisque c'est un super ensemble de EJB. Ni COM ni CLR ne fournissent un tel support. Ces mécanismes ont tendance à avoir toujours des tâches qui alourdissent le système, comme le chargement de toutes les entités en relation avec les serveurs J2EE. Ceci mène à basses de performances pour les applications. COM+ et CLR supportent des liaisons de persistance permettant la persistance de données en stockage externe. EJB par contre n'inclut pas encore des liaisons de persistance. Ce qui rendre les mécanismes de persistance et les relations basées sur les conteneurs plus faibles en dehors des applications pures de bases de données.
- **Les services** :
  CORBA a un ensemble complet de services standardisés mais à la plupart d'entre eux il manque une implémentation. COM+ a complété COM avec un ensemble riche de services clés incluant les transactions et les messages. Un large ensemble comparable de service fait partie de J2EE qui inclut EJB. CLR offre, avec COM+, un support d'interopérabilité incluant tous les services COM+ (appelés maintenant les *Enterprise Services*). Cependant ceux-ci ne sont pas couverts par les spécifications CLI. Le support de coordination de transactions distribuées est disponible en CORBA et COM+(et donc en CLR) main il ne l'est pas dans la portée du standard EJB.
- **Le Déploiement** :
  Tous les J2EE, COM+, CCM et CLR suivent le concept MTS (*Microsoft Transaction Server*) de la programmation basé sur l'attribut. EJB a factorisé les attributs et les a placés dans des descripteurs de déploiement basé sur XML permettant une étape de déploiement spécialisée. J2EE a élargit le concept de descriptuer de déploiement pour plusieurs modèles de composants. CLR combine la configuration basé sur XML avec les attributs personnalisés basés sur CLI. Les attributs personnalisés simplifient l'alignement du code et de méta-données car l'attribut est placé directement dans le code source approprié. Cela factorise le rôle du développeur (qui place les attributs personnalisés) et le déployeur (qui manipule le fichier de configuration).
- **Les communications** :
  CORBA supporte IIOP comme standard comme protocole pour les interoperations inter-ORB au niveau des communications. En plus, l'OMG a adopté XML et son mode graphique format de description des communications au niveau de l'application. Java supporte les liaisons IIOP, mais à la base il supporte le protocole RMI. Le support Java pour XML est en train de s'améliorer. COM utilise DCOM comme un protocole de communication natif et COM+ ajoute le support de plusieurs formats de messageries. CLR continue le support de tous les formats supportés par COM et COM+ et ajoute le support des définitions de schémas XML et des protocoles d'invocation SOAP.

1.5 Conclusion

Dans ce chapitre, nous avons présenté l'évolution du développement des systèmes distribués et parallèles. Ce développement a permis la création de nouveaux types d'applications qu'il n'était pas possible de concevoir en utilisant un seul système informatique. Nous avons introduit quelques architectures de systèmes distribués. L'approche GRID(grille de calcul) introduit une autres vue des machines distribuées comme calculateur unique. Nous nous sommes intéressés aux systèmes à image unique (SSI) dans lesquels le système distribué est vu comme une seule machine qui a ses ressources propres. La plupart des systèmes de ce genre sont généralement construits à partir d'une couche logicielle intermédiaire au dessus du système d'exploitation réseau ou au dessus de système du GRID. Cette couche logicielle a pour rôle de masquer l'hétérogénéité et de traiter les aspects comme la transparence du système, l'extensibilité, la sécurité, la tolérance aux pannes et le passage à l'échelle. Dans certains cas nous appelons cette couche un *middleware*.

Les middlewares actuels sont basés sur des technologie de programmation avancées comme les objets repartis ou les composants repartis. La programmation orientée-objets a fourni un ensemble de caractéristiques comme l'encapsulation, la séparation entre les données et les opérations, l'extensibilité, la réutilisation et la modularité.

La programmation orientée-objets n'a pas vraiment répondu aux espérances au niveau de la réutilisation du code existant. Pour cela il était évident de chercher d'autres technologies qui améliorent spécialement cette caractéristique de réutilisation. La programmation par composants semble la solution trouvée pour ce genre de problème. Le développement par composants est inspiré par la construction des bâtiments où nous composons le bâtiment par l'ajout de briques et de motifs. Pour cela il faut suivre certaines règles d'architecture et de spécifications. Cette démarche a introduit la notion d'architecture logicielle. Dans ce chapitre, nous avons donné quelques définitions importantes sur l'architecture logicielle et les termes utilisés dans la conception et le développement des applications a base de composants comme le composant, le framework, le design pattern, la composition ...etc.

La programmation orientée-composants a donc apporté une évolution importante et elle a été considérée dans la plupart des cas une évolution de la méthodologie objets. Donc, il était évident d'introduire et de standardiser des modèles de composants. Les modèles standards de l'industrie sont les modèles basé sur les anciens modèle à objets. Les grandes écoles dans ce domaine sont SUN, MICROSOFT et l'OMG. Il était donc nécessaire de présenter ces trois modèles ainsi qu'une comparaison entre eux selon certains aspects.

Les modèles de l'industrie ne fournissent pas certains aspects considérés dans le monde de calcul distribué parallèle. Pour cela, nous nous sommes intéressé, dans le chapitre suivant, à des modèles de composants qui sont issues du monde de la recherche. Nous présentons également des frameworks qui utilisent soit des modèles de l'industrie étendus pour des aspects de calcul distribué/parallèle, soit des modèles de recherches.

Chapitre 2

Environnements de développement et d'exécution des applications distribuées et parallèles à base de composants

2.1 Introduction

La conception d'une application parallèle et distribuée exige de s'appuyer sur une méthodologie et des outils qui facilitent ce travail. Ces mécanismes peuvent être basés sur des modèles de conception, des bibliothèques ou des frameworks. Le choix de cet environnement de programmation n'est pas simple et, de plus, deux types de problèmes se posent : comment exprimer le parallélisme, sous quelle forme, et comment réaliser la distribution des applications sur un ensemble de stations.

L'utilisation d'un modèle de programmation peut être maîtrisée par des outils génériques et flexibles permettant au développeur d'écrire son programme parallèle et distribué de manière simple et intuitive. Ces outils appartiennent à des modèles de conception, frameworks ou bibliothèques, approches différentes ayant pour objectif l'aide à la conception des applications parallèles.

Nous avons présenté dans le chapitre 1, les architectures des systèmes distribués et les avantages que la conception et la programmation par composants peuvent apporter à ces systèmes. Cela nécessite la construction de frameworks qui fournissent des outils de conception et apportent plus de facilité dans leur utilisation.

Les technologies industrielles décrites dans le chapitre 1, ne s'adressent pas aux applications parallèles, dans le sens de paralléliser le calcul. Certes, elles fournissent un environnement distribué qui peut être utilisé comme un middleware pour distribuer certains traitements, mais ces environnements doivent être modifiés, étendus et enrichis pour prendre en compte la programmation parallèle. Leurs spécifications et leurs implémentations visent les applications industrielles, les systèmes d'informations, le Web...etc. Il n'était donc pas dans leur but de fournir des frameworks pour les applications parallèles et prendre en compte les aspects haute performance.

Tout cela nous conduit à nous intéresser à une catégorie d'architecture logicielle plutôt dédiée aux applications distribuées parallèles. Ces architectures ont été conçues dans des laboratoires de recherche afin de créer des frameworks d'exécution de programmes parallèles et distribués. Pour le calcul parallèle et distribué, l'enjeu est différent.

L'objectif du calcul parallèle et distribué est de distribuer et paralléliser les tâches de calcul sur le plus grand nombre (optimal) d'unités centrales possibles pour atteindre une meilleure efficacité d'exécution. D'autre part, il faut minimiser le coût de la conception de ce type d'application et donc fournir certains outils d'aide au parallélisme et à la distribution. Cette conception devient de plus en plus difficile, en raison de la complexité croissante de ce type de système. La technologie composants tente de trouver des solutions aux problèmes de la conception, de la gestion de l'évolution, de la réutilisation des applications.

Pour construire une architecture ou un framework à base de composants pour la programmation distribuée parallèle, il existe certains éléments à prendre en considération :

- **Caractéristiques du composants** : l'architecture est utilisée pour des composants à haute performance de différentes granularités. Les interfaces seront implémentées selon différents paradigmes comme par exemple dans le style SPMD (*Single Program Multiple Data*) ou celui des modèles multithreadés à mémoire partagée. Il est important aussi de résoudre des problèmes comme l'interaction des composants avec des processus multiples, la présence de systèmes d'exécution sophistiqués, de bibliothèques de passage de message, de threads et du transfert efficace d'ensembles de données.
- **Hétérogénéité** : l'architecture doit être capable de gérer une application multi-composants qui peut s'exécuter dans des architectures variées, implémentée en différents langages de programmation et utilisant différents systèmes d'exécution. En outre, les priorités de la conception doivent être tournées vers la satisfaction des besoins communs des applications dans les environnements de calcul haute performance. Par exemple, l'interopérabilité, entre les langages usuels dans la programmation du calcul scientifique comme Fortran, C et C++, peut avoir la priorité.
- **Composants locaux et distants** : les composants locaux sont des composants qui résident dans l'espace d'adressage de l'application contrairement aux composants distants. L'interaction entre les composants locaux ne doit coûter qu'un appel de fonction et les composants distants doivent prendre avantage des protocoles à zéro copie et exploiter les autres avantages offerts par la gestion des réseaux. Il faut distinguer les besoins des composants distants qui s'exécutent dans un réseau local et ceux qui s'exécutent dans un réseau étendu. Les applications à base de composants qui s'exécutent sur une grille de calcul à haute performance doivent être capables de satisfaire des contraintes de temps réel et interagir avec les différents ordonanceurs du calcul à grande échelle.
- **Intégration** : l'intégration d'un composant dans un framework doit être souple. En général, il n'est pas nécessaire de développer un composant spécialement pour l'intégrer avec le framework ou de réécrire substantiellement un composant existant .
- **Haute performance** : il est très important qu'un ensemble de standards soit défini pour supporter les aspects interactions à haute performance. Il est préférable

d'éviter le plus possible les copies et les communications ou la synchronisation et d'encourager les implémentations efficaces comme le transfert parallèle de données.
- **Ouverture** : il est important que les spécifications d'une architecture soient ouvertes et qu'elles soient utilisées dans un logiciel ouvert (libre). Dans le monde du calcul à haute performance, cette flexibilité est nécessaire pour suivre les demandes de changement dans le monde de programmation scientifique.

La définition d'une architecture spécifique pour les environnements distribués et/ou parallèles est faite soit par l'extension d'une architecture logicielle distribuée industrielle soit par la définition d'une nouvelle architecture dédiée à ce type d'environnement. Dans le premier cas, les environnements sont basés sur les modèles industriels, dans le deuxième cas ils sont basés sur les modèles de recherche.

Nous présentons, dans la suite de ce chapitre, quelques environnements de développement et d'exécution pour les applications parallèles et distribuées à base de composants.

2.2 Le projet GridCCM

Le projet "*GridCCM*" [CPR03, DPP03] réalise une extension du modèle CORBA CCM pour définir et construire des composants et applications parallèles. Il s'agit notamment de pouvoir coupler plusieurs codes de calcul scientifique pour réaliser des simulations. Comme dans la plupart des solutions à base de composants cette approche est apparue dans ce contexte afin de maîtriser la complexité de réalisation de telles applications.

Dans ce projet, les auteurs décrivent une étude sur *GridCCM* qui est une extension du modèle de composant de CORBA CCM supportant les composants parallèles (que nous définissons plus tard). Un prototype réalisé avec deux implémentations de CCM a permis de valider la faisabilité de l'approche.

2.2.1 Vue d'ensemble du modèle

L'objectif de *GridCCM* est d'encapsuler du code parallèle dans des composants CORBA avec une modification minimale des codes parallèles et d'apporter un minimum de modifications au modèle CCM. Pour cela, CORBA IDL n'a pas à être modifié. Les auteurs ont restreint l'introduction de codes parallèles, au seul type de code parallèle qu'est le SPMD (*Single Program Multiple Data*) dans un composant parallèle. Dans le code SPMD, chaque processus exécute le même programme mais sur des données différentes. Ce choix est fait selon deux considérations : la première est que de nombreux codes parallèles sont en effets des codes SPMD. La seconde, est que les codes SPMD apportent un modèle d'exécution gérable. La figure 2.1 illustre le composant parallèle dans un framework CORBA. Le code SPMD continue à être capable d'utiliser MPI pour ses communications inter-processus mais, il utilise CORBA pour communiquer avec d'autres composants. Pour éviter un goulot d'étranglement, tous les processus d'un composant parallèle participent aux communications inter-composants [RP00]. Quand les données doivent être redistribuées pendant les communications, le modèle *GridCCM* supporte cette redistribution de manière la plus transparente possible. Le client doit seulement décrire comment les données sont distribuées localement et comment les données doivent être automatiquement redistri-

buées selon la performance des serveurs. Un composant *GridCCM* doit apparaître comme le plus proche possible d'un composant *séquentiel*. Par exemple, un composant séquentiel doit pouvoir se connecter avec un composant parallèle. Pour atteindre ces objectifs, *GridCCM* introduit la notion de composant parallèle. Sa définition est : *Un composant parallèle est une collection de composants séquentiels identiques. Il exécute en parallèle tout ou partie de ses services.*

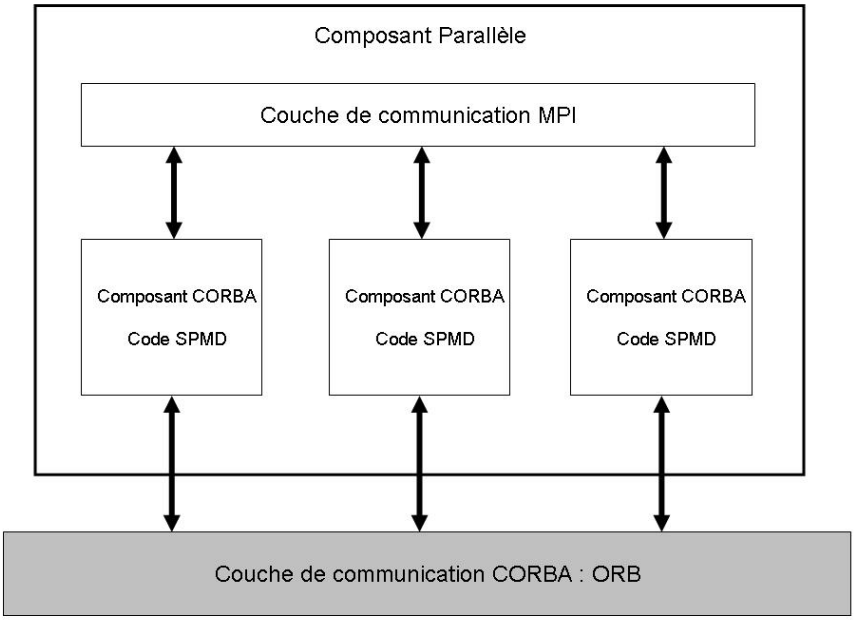

FIG. 2.1 – Le concept de composant parallèle dans *GridCCM*

Le concepteur du composant parallèle exprime le parallélisme du composant au travers d'un fichier XML auxiliaire. Le fichier contient la description des méthodes parallèles du composant, les arguments distribués de ces méthodes et la distribution prévue de ces arguments.

2.2.2 Introduction du parallélisme à CCM

Pour introduire le support du parallélisme, comme la redistribution de données, sans modifier l'ORB, les auteurs ont choisi d'introduire une nouvelle couche logicielle entre le code client et le code de talon. Ce schéma a été utilisé avec succès avec PaCO++ [PPR04] pour un problème similaire de la gestion des objets CORBA parallèles. Le rôle de la couche *GridCCM* est de permettre une gestion transparente du parallélisme. Un appel à une méthode d'un composant parallèle est intercepté par cette nouvelle couche. La couche envoie les données distribuées des nœuds clients aux nœuds serveurs. La redistribution des données peut être faite du côté client, du côté serveur ou pendant la communication entre le client et le serveur. La décision dépend de plusieurs contraintes comme la faisabilité (spécialement les besoins en mémoire) et l'efficacité (la performance du réseau client et la performance du réseau serveur). Un autre objectif est de gérer les exceptions parallèles.

La couche de gestion parallèle est générée par un compilateur spécifique à *GridCCM* comme illustré dans la figure 2.2. Ce compilateur utilise deux fichiers : une description IDL du composant et une description XML du parallélisme du composant. Pour avoir une couche de transparence, une nouvelle description IDL est générée pendant la génération du composant. La couche GridCCM utilise intérieurement une interface dérivée de l'interface originale. La nouvelle interface IDL est l'interface qui est invoquée à distance du côté serveur. L'interface IDL originale est utilisée entre le code client et la couche *GridCCM* côté client et côté serveur. Dans la nouvelle interface, les arguments de l'utilisateur qui sont décrits comme distribués ont été remplacés par leurs types de données équivalents.

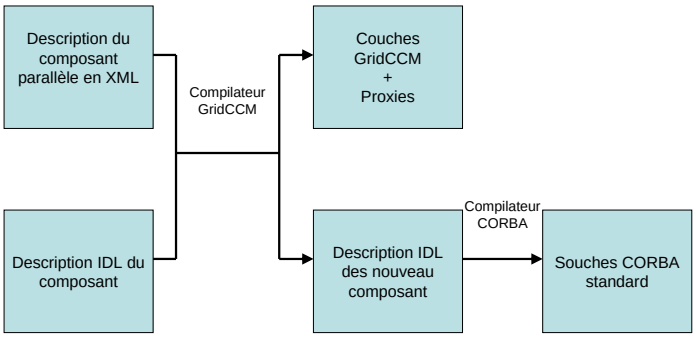

FIG. 2.2 – Compilation dans *GridCCM*

A cause de cette transformation, il existe quelques contraintes sur les types de données qui peuvent être distribuées. L'implémentation actuelle exige que le type utilisateur soit un type de "IDL sequence", c'est-à-dire un tableau d'une dimension. Une distribution d'une dimension peut être appliquée automatiquement. Cet arrangement peut être facilement étendu pour des tableaux multi-dimensionnels : un tableau de deux dimensions peut être transformé en une séquence de séquences. Dans ce contexte, il est important de remarquer que les types IDL ne permettent pas un mapping direct de types scientifiques comme les tableaux multi-dimensionnels ou les types de données de nombres complexes.

Transparence du parallélisme

Un autre objectifs de *GridCCM* est de permettre de voir un composant parallèle comme un composant séquentiel. Pour cela les nœuds d'un composant parallèle et les nœuds des composants *maison* ne sont pas directement exposés aux autres composants. Deux entités sont introduit : le `HomeManager` et le `ComponentManager`. Ceux sont respectivement des proxys pour les nœuds maisons et les nœuds du composant parallèle.

Une application qui a besoin de créer un composant parallèle interagit à la façon standard de CCM, avec le *HomeManager* , au lieu d'utiliser chaque nœud maison. Le

`HomeManager` est configuré pendant la phase de déploiement. Les références à tous les nœuds maisons sont données à travers une interface spécifique de `HomeManager`. Ensuite, le `HomeManager` appelle les maisons sur tous les nœuds.

D'une façon similaire, quand un client récupère une référence à un composant parallèle, il a effectivement une référence au *ComponentManager*. Quand deux composants parallèles sont connectés, les `ComponentManagers` s'échangent d'une façon transparente les informations sur les composants parallèles pour configurer les couches *GridCCM*.

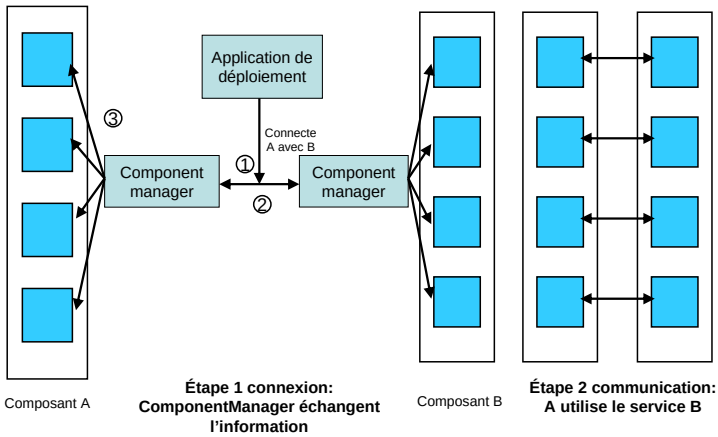

FIG. 2.3 – Connexion entre deux composants A et B

La figure 2.3 montre un exemple de connexion entre deux composants parallèles : A et B. L'outil de déploiement connecte d'abord, A avec B en utilisant le protocole standard de CCM. Ensuite, le `ComponentManager` de A demande au `ComponentManager` de B l'information. Par exemple, le composant B donne au composant A toutes les références des nœuds de B. Troisièmement, le `ComponentManager` de A configure les couche de tous les nœuds de A. Quatrièmement, quand le composant A achève un appel à un service de B, tous les nœuds de A peuvent participer à la communication avec les nœuds de B.

2.3 Le projet Proactive

ProActive [Ant02, BCH+02] est une bibliothèque Java pour la programmation d'applications parallèles et distribuées. *ProActive* simplifie la programmation distribuée sur une grappe de stations de travail ou sur internet. *ProActive* est facile à mettre en œuvre et facile à porter puisqu'il est 100% Java et ne demande aucune modification ni dans la JVM ni dans le compilateur de Java. *ProActive* est basé sur l'utilisation du protocole meta-objet (meta-object protocol) [KdRB91] et utilise RMI [Sun98c] et la réflexion [Sun98b] afin de réaliser la distribution des objets des applications.

ProActive propose le concept d'objet actif [CKV98] et de groupe de communication d'objets [BBC02a].

2.3.1 Les objets actifs

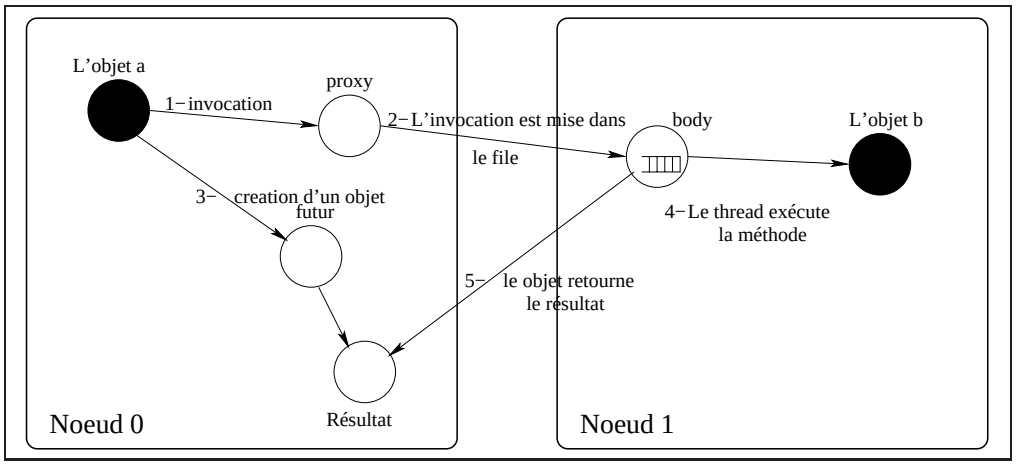

FIG. 2.4 – L'invocation d'une méthode d'un objet actif.

Une application distribuée qui utilise *ProActive* est composée d'objets standards Java et d'objets particuliers appelés objets actifs. Chaque objet actif a son propre thread d'exécution qui se charge d'exécuter les invocations de méthodes provenant d'autres objets. Ces invocations sont des invocations asynchrones (appel non bloquant). Lorsqu'une invocation de méthode est effectuée sur un objet actif, un objet spécial appelé "objet *futur*" [1] est immédiatement renvoyé. Le thread courant peut alors continuer son exécution normalement et l'exécution de l'invocation de la méthode est faite par le thread lié à l'objet actif. Lorsque le thread qui a fait l'invocation veut utiliser le résultat, il utilise l'objet futur jusqu'à la disponibilité du résultat. C'est la notion d'attente par nécessité [Car89] (cf. la figure 2.4).

Les invocations entrantes d'un objet actif sont gérées par défaut par une strategie FIFO (First In First Out) qui définit l'ordre d'exécution des invocations par le thread lié à l'objet actif. Il y a possibilité de redéfinir une autre stratégie de gestion que celle donnée par défaut FIFO.

ProActive fournit un mécanisme permettant la création d'objets actifs à distance. Il donne donc la possibilité de créer un objet dans une autre JVM que celle qui a fait la demande de création. *ProActive* modifie le code correspondant à la création d'un objet actif afin de transformer un objet standard en un objet actif. Le reste du code reste sans modification. La création d'un objet actif demande en plus des paramètres passés au constructeur de l'objet, un paramètre supplémentaire qui représente le nœud où l'objet actif doit être créé.

Le code suivant est un exemple de la création d'un objet actif de la classe A sur le nœud *monNœud*. Le constructeur de la classe A, par exemple, prend deux paramètres. Le premier est une chaîne de caractères et le second est un entier.

```
Object[] params = {''chaine'', new Integer(1)};
A a = (A) ProActive.newActive(''A'', params, monNoeud);
```

[1] un objet *futur* est un objet de la sous-classe de la classe du résultat de la méthode invoquée.

L'objet actif *a* cré son propre thread d'exécution qui exécute les invocations de méthodes sur cet objet actif dans l'ordre par défaut FIFO.

La création d'objets actifs demande d'indiquer la JVM hébergeant ces objets. Afin de réaliser cela, *ProActive* définit une classe *Node*. Un objet de la classe *Node* regroupe plusieurs objets actifs dans une seule entité logique. Une JVM peut héberger plusieurs objets de la classe *Node*. Elle héberge donc les objets actifs regroupés dans ces nœuds. *ProActive* utilise des fichiers XML pour décrire les nœuds d'une application, les ressources du calcul disponibles et le déploiement des ces nœuds sur les ressources.

La migration des objets actifs est possible dans *ProActive*. Cette migration est faite par invocation de la méthode *migrateTO()* qui fait migrer un objet actif d'une JVM vers une autre.

2.3.2 Les groupes de communication d'objets

ProActive fournit un mécanisme pour réaliser des communications dans un groupe d'objets [BBC02a]. La communication entre un groupe d'objets est basée sur les mêmes mécanismes d'invocation asynchrone et d'attente par nécessité du résultat dans les objets actifs. Une communication (une invocation de méthode) dans un groupe de n objets se traduit par n invocations de la méthode sur les objets du groupe.

Le mécanisme de communication de groupe simplifie la réalisation d'activations parallèles de tâches semblables. L'utilisateur utilise un groupe d'objets de la même manière qu'il utilise un objet actif.

Les objets appartenant à un groupe d'objets doivent être du même type (avoir une super-classe commune entre eux) et ne sont pas forcément des objets actifs. Il y a la possibilité de créer un groupe d'objet vide et d'y ajouter plus tard des objets déjà existants ou nouveaux. On peut également créer les objets et le groupe en même temps. Dans ce dernier cas, il faut fournir deux listes. La première est une liste de paramètres passés aux constructeurs d'objets et la deuxième est une liste d'objets de la classe *Node* qui hébergent les objets créés.

La gestion de groupe d'objets est faite en utilisant des méthodes prédéfinies comme *add()*, *remove()*, *size()* etc...

L'invocation d'une méthode d'un groupe d'objets se fait de la même façon qu'une invocation d'une méthode sur un objet normal. Cependant, sa sémantique est différente puisque cette invocation est réalisée par une série d'invocations asynchrones de la méthode sur les objets actifs du groupe.

Il y aura création implicite des objets futurs afin d'accueillir les résultats. Par défaut, les paramètres de la méthode sont diffusés à tous les objets du groupe. On peut aussi distribuer (opération scatter) les paramètres aux objets actifs.

Le résultat d'une invocation d'une méthode sur un groupe est aussi un groupe d'objets (groupe de résultats). Ce groupe de résultats est créé implicitement. Il se charge de collecter les résultats des invocations de la méthode sur les objets du groupe. Le mécanisme d'attente par nécessité est utilisé pour accéder aux résultats.

ProActive propose la possibilité de créer des hiérarchies de groupes, c'est-à-dire des groupes de groupes d'objets. Cela permet de structurer une application.

En conclusion, *ProActive* est un outil qui utilise la surcharge de méthodes et la réflexion pour réaliser la distribution d'applications. Il utilise la notion d'objet actif. Ce dernier est un objet dont les méthodes peuvent être invoquées de manière asynchrone. Les résultats d'invocation de méthodes sont gérés par un objet spécial appelé "objet futur". *ProActive* est 100 % Java, donc il est facile à porter.

2.3.3 Les composants de *ProActive* avec *Fractal*

ProActive a été étendu [BCM03] pour permettre de construire des composants pour le parallellisme et la distribution de code. Pour cela, il implémente le modèle de composants *Fractal* [BCL+04].

Le modèle de composants *Fractal*

Fractal définit un modèle conceptuel général avec des API en Java. Le modèle de composants fractal [BCS04, BCL+04] est "*un modèle de composant extensible et modulaire qui peut être utilisé avec différents langages de programmation pour concevoir, implémenter, déployer et reconfigurer différents systèmes et applications, des système d'exploitation aux middlewares et aux interfaces d'utilisateurs graphiques*". Le modèle de composant *Fractal* répond aux limitations des modèles standards de l'industrie (c.f. 1.4) sur l'extensibilité et l'adaptabilité. Il introduit la notion du composant entouré par un ensemble de capacités de contrôle. Autrement dit, les composants Fractal sont réflectifs, et leurs capacités de réflexivité ne sont pas fixées dans le modèle mais elles peuvent être étendues et adaptées pour atteindre les objectifs et les contraintes du programmeur.

Le but essentiel de *Fractal* [BCS04]est d'implémenter, déployer et gérer (surveiller et reconfigurer dynamiquement) les systèmes logiciels complexes. Ce but motive le dispositif principal du modèle Fractal : les *composants composites* (pour avoir une vue uniforme de l'application sur des niveaux d'abstraction divers), les *composants partagés* (pour modéliser les ressources), les *capacités d'introspection* (pour surveiller un système en marche), et les *capacités de configuration et reconfiguration* (pour déployer et reconfigurer l'application dynamiquement). Un autre but de Fractal est d'être applicable à plusieurs logiciels, des systèmes embarqués aux applications de serveurs et aux systèmes d'information. Les dispositifs avancés dans Fractal ont un coût qui n'est pas toujours compatible avec les ressources limités des environnements disponibles. Cela affecte l'efficacité du système. Le modèle Fractal donne une solution à ce problème par le fait d'être défini comme une spécification fixe et minimale que tous les composants Fractal doivent suivre. Cette spécification est aussi un système extensible de relations entre tous les concepts définis et les API correspondantes que les composants Fractal peuvent implémenter ou non, selon ce qu'ils veulent offrir aux autres composants.

L'ensemble des spécifications Fractal est organisé selon des niveaux de contrôle croissants, c'est-à-dire dans un ordre croissant des capacités de refléxivité (introspection et intersession). Ainsi le modèle Fractal est représenté par les caractéristiques suivantes :

- Au niveau le plus bas, un composant Fractal est une entité d'exécution qui ne fournit pas de capacité de contrôle à d'autres composants. Il agit donc, comme un objet (un tel composant peut être utilisé d'une seule manière, par l'appel des méthodes).

En effet, un objet est un composant Fractal sans aucune capacité de contrôle. Ce dispositif est utile pour manipuler les cas où les composants doivent se connecter à des logiciels de legacy.

- ***L'introspection*** : Dans ce niveau de legacy, un composant Fractal peut fournir une interface standard semblable à l'interface `IUknown` du modèle COM(c.f. 1.4.2). Cela permet de découvrir toutes ses interfaces externes, ou autrement dit, toutes ses frontières (comme un objet, un composant Fractal peut fournir plusieurs interfaces). Selon le niveau de l'observation ou de la granularité, un composant Fractal peut être une boîte noire ou une boîte blanche (c.f. 1.3.4). Quand il est considéré comme une boîte noire, les seuls détails visibles du composant sont les points de connexion qui sont appelés *interfaces externes*. Il existe deux types d'interfaces : clients(requis) et serveurs(fournies). Un composant Fractal peut avoir un seul type de ces interface, les deux ou aucun type de ces interfaces. L'introspection de composant est de deux types : introspection du composant et introspection d'interface. Dans l'introspection du composant, le composant fournit l'implémentation de l'interface `Component` pour découvrir les interfaces (points de connexion) du composant. Dans l'introspection d'interface, des opérations permettent d'obtenir le nom, le type et d'autres information concernant l'interface externe du composant. Il est possible aussi de récupérer les noms d'autres interfaces du composant à travers l'introspection d'une interface données.

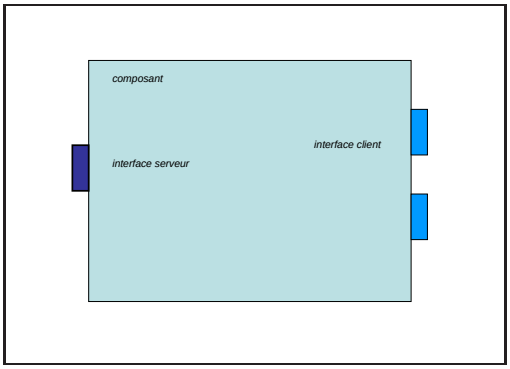

FIG. 2.5 – Vue externe du composant Fractal

- ***La configuration*** : Un composant Fractal peut fournir des interfaces de contrôle pour introspecter et modifier son contenu, c'est-à-dire ce qui est à l'intérieur des frontières. Dans le modèle de composant Fractal, ce contenu est constitué d'autres composants Fractal qui sont liés ensemble, et appelés ses sous-composants. Un composant Fractal donc, peut ou ne pas fournir une interface pour contrôler l'ensemble de ses sous composants, l'ensemble des liens entre ses sous-composants. La figure 2.6 représente la structure interne d'un composant Fractal.
  Le contrôleur du composant peut avoir des interfaces internes et externes. Les interfaces externes sont accessibles par l'extérieur du composant tandis que les interfaces internes sont accessibles seulement par les sous-composants du composant. Le contrôleur d'un composant peut fournir des représentations de connection de sous-composants, intercepter les invocations entrantes et sortantes des sous-composants

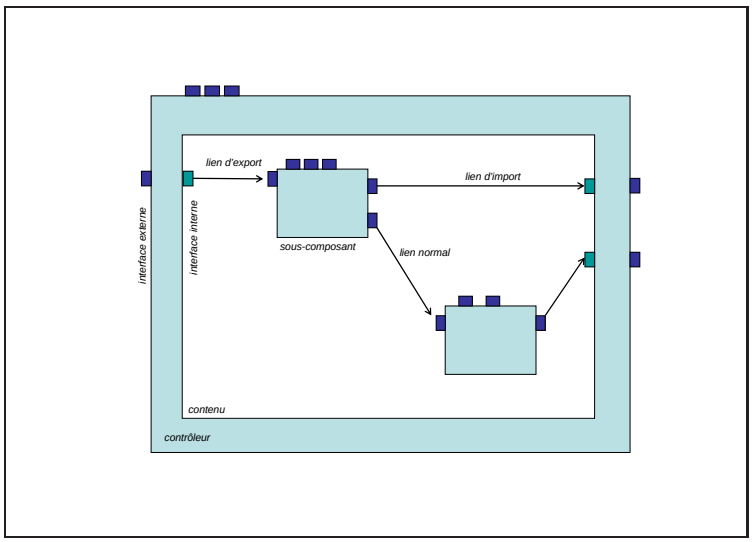

FIG. 2.6 – Vue interne du composant Fractal

et/ou superposer un contrôle au comportement d'un sous composant.
Les liens représentent les communications entre les composants. Dans Fractal, il existe deux types de liens : *primitifs* et *composite*. Les liens primitifs sont des liens entre une interface client et une interface serveur. Les liens composites sont les communications entre un nombre arbitraire d'interfaces de composants de types arbitraires de langages. Ils sont représentés comme un ensemble de liens primitifs et des composants de liens (souches, squelettes , adaptateurs ...). Un composant de liens est un composant Fractal dont le rôle est dédié à la communication. Les composants de liens sont appelés aussi des *connecteurs*.
Un composant Fractal peut disposer d'un *attribut* qui est une propriété configurable du composant. Les attributs en général sont de type primitif et ils sont utilisés pour configurer l'état du composant sans avoir besoin d'utiliser les liens. Fractal dispose de certaines interfaces pour contrôler le composant comme des interfaces de contrôle pour les attributs (changer, mettre, enlever un attribut), les liens (mettre, enlever des liens entre les interfaces), le contenu (ajout ou suppression de sous-composants) et le cycle de vie (pour commencer et arrêter l'exécution d'un composant).

- ***L' instanciation*** : En plus de ces capacités de contrôle, le modèle Fractal spécifie un framework pour l'instantiation des composants. Le rôle essentiel de ce framework est la création de nouveaux composants. Il est basé sur les *fabriques* (factories). Il existe des fabriques standards (pour créer certains types de composant) et des fabriques génériques (pour la création des composants de plusieurs types). Fractal spécifie un type spécial de fabriques standards que l'on appelle un *template* qui crée des composants qui sont quasi isomorphes à eux même. Autrement dit, les composants créés par le composant template ont les mêmes interfaces fonctionnelles client et serveur comme dans le composant template, mais ils peuvent avoir des interfaces de contrôle différentes.
- ***Le typage*** : Fractal dispose d'un système simple de typage pour les composants et les interfaces. Ce système de typage reflète les caractéristiques essentielles des

interfaces du composant, c'est-à-dire leurs noms, leurs types de langage et leur rôle (client ou serveur). Dans ce système de typage, un type de composant est un ensemble de types de ses interfaces. Le type d'une interface est fait d'un nom, d'une signature et d'un rôle. Un composant peut avoir "une cardinalité" et l'indication que le fonctionnement de cette interface est disponible ou non (contingence).

Le framework de composants ProActive

Le framework de composants en *ProActive* a été conçu et implémenté en utilisant *Fractal* et *Proactive*. Cela permet de coupler les codes parallèles et distribués directement dans une bibliothèque Java *ProActive*. La notion de composant grille (*Grid Component*) [BCM03]a été introduit afin de supporter les besoins en haute performance et de déploiement parallèle et distribué. La figure 2.7 récapitule les trois cas différents pour la structure d'un composant de grille. Dans le cas d'un composite construit à partir d'une collection de composants qui fournissent un service commun, les *communications collectives* sont essentielles pour la facilité et l'efficacité de la programmation.

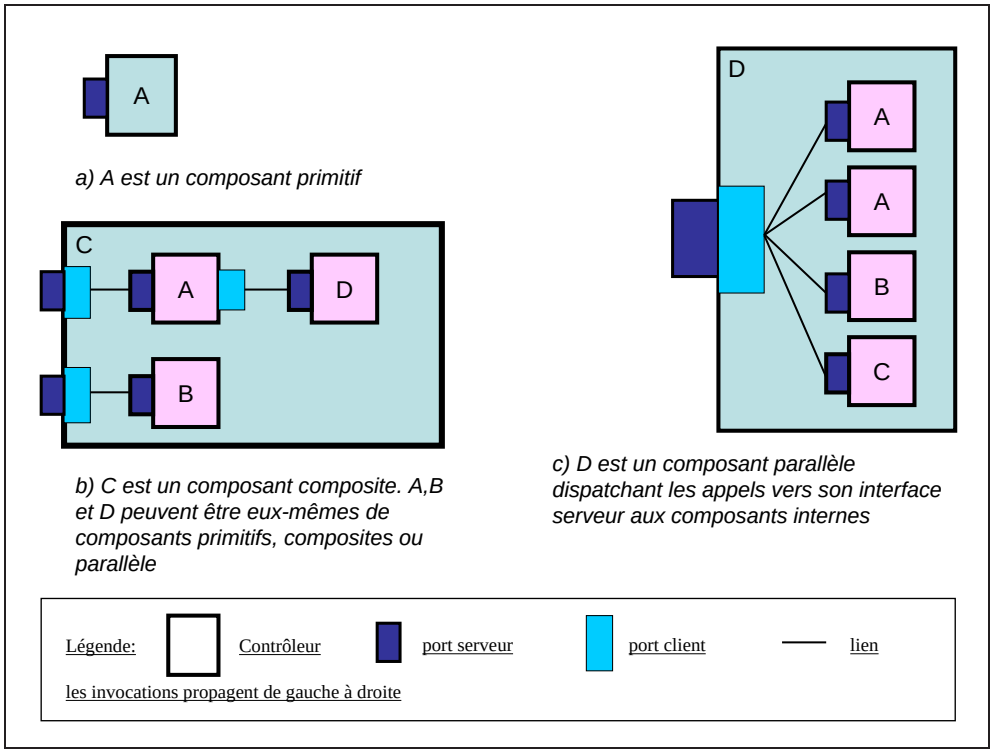

FIG. 2.7 – Différentes architectures de composant de grille ProActive : *GridComponent*

Une définition synthétique d'un composant Proactive est le suivant :
Un composant *ProActive* :

- est formé d'un (ou plusieurs) objets actifs s'exécutant sur une (ou plusieurs) JVM.
- fournit un ensemble de ports serveurs (Interfaces Java)
- définit un ensemble de ports clients (attributs Java si le composant est primitif)

- peut être de trois types différents :
  1. primitif : définit avec un code Java qui implémente les interfaces serveurs fournies et qui spécifie le mécanisme de liaison avec le client.
  2. composite : qui contient d'autres composants
  3. parallèle : qui est un composite mais qui réexpédie les appels à ses interfaces externes vers ses composants internes.
- communique avec les autres composants à travers des communications 1-à-1 ou de groupe.

Les composants *ProActive* peuvent être configurés en utilisant

- un descripteur XML qui définit les interfaces client/serveur, le contenu et les liaisons dans un style de langage de description d'architecture
- la notion de *nœud virtuel* en attrapant les capacités et les besoins de déploiement.

Les connections entre les composants sont de types seules ou collectives. Pour le déploiement des composants *ProActive* la notion de nœud virtuel est introduite. Ce concept est très important dans l'abstraction pour les composants qui ont ainsi un moyen pour manipuler les activités distribuées. Il est important aussi de pouvoir *composer* les nœuds virtuels. Un composite est un composant qui définit à travers un nombre de sous-composants qui définissent déjà leur propre usage et liaisons des nœuds virtuels. La composition des nœuds virtuels est un moyen pour contrôler la distribution des composants composites.

ProActive est basé sur le protocole Meta-Object (MOP) qui permet d'ajouter plusieurs aspects aux standards des objets Java comme l'asynchronisme et la mobilité. Les objets actifs sont référencés à travers les souches et la communication avec eux, est faite de la même manière qu'ils soient locaux ou distants. La même idée est utilisée pour gérer les composants en ajoutant seulement des méta objets pour se charger des aspects des composants. Dans cette implémentation et à cause de l'utilisation des facilités de MOP tous les composants sont constitués d'au moins un objet actif qu'il soit primitif ou composite.

Pour intégrer les opérations de gestion des composants dans la bibliothèque *ProActive*, une extension de l'architecture de la bibliothèque est utilisée. De cette manière, les composants restent pleinement compatibles avec les objets actifs standards et donc héritent des dispositifs tels que la mobilité, la sécurité, le déploiement. Un point particulier, concernant l'intégration de Fractal et de ProActive, est la gestion des requêtes du composant à côté des requêtes fonctionnelles. Quand les appels de méthodes arrivent au corps, ils sont dirigés vers la file d'attente des requêtes. Une politique de file d'attente FIFO est utilisée pour cette gestion. La manipulation de la requête dans la file d'attente dépend de la nature de cette requête et correspond à l'algorithme suivant :

```
loop
  if componentLifeCycle.isStarted()
      get next request
      // all request are served
  else if componentLifeCycle.isStopped()
          get next component controller request
          // only component requests are served
  ;
```

```
  if gotten request is a comp. life cycle request
     if startFc --> set started = true ;
     if stopFc --> set started = false ;
  ;
;
```

Il faut remarquer que dans l'état d'arrêt seules les requêtes du contrôleur sont des services. Ceci veut dire qu'un appel ProActive standard, provenant d'un stub ProActive standard, ne sera pas traité dans l'état d'arrêt (mais il restera dans la queue).

L'implémentation des interfaces collectives est basée sur les API de groupe de ProActive. Ce type d'interface a seulement un sens sur les interfaces clients qui peuvent être liés avec plusieurs interfaces serveurs. A l'opposé, une interface serveur peut être accédée par plusieurs interfaces client, les appels sont traités séquentiellement. Le fait de spécifier l'interface serveur en tant que "collective" ne changerait pas son comportement.

Les groupes API de ProActive permettent de faire des communications de groupes d'une manière transparente. L'implémentation des interfaces collectives diffère légèrement des spécifications de Fractal : au lieu de créer une nouvelle interface avec un nom étendu pour chaque membre de la collection, on utilise seulement une interface (qui est effectivement un groupe).

Les liaisons collectives sont ensuite faites d'un manière transparente comme des liaisons séquentielles multiples sur la même interface. L'utilisation d'une interface collective serveur implique l'utilisation du formalisme API de ProActive y compris la possibilité de choisir entre "*à la diffusion* " et *la collecte* des appels[BBC02b].

Avec cette implémentation, un type composant parallèle est introduit. Les composants parallèles sont des composants, composites puisqu'ils encapsulent d'autres composants. Leurs spécificités reposent sur le comportement de leurs interfaces serveurs externes. Ces interfaces sont connectables à travers un groupe proxy vers les interfaces des composants internes du même type. Un appel vers un composant parallèle va être expédié et suivi vers un ensemble de composants internes, qui vont traiter la requête de façon parallèle.

2.4 Common Component Architecture (CCA)

Common Component Architecture (CCA) est un projet monté par des laboratoires de recherche nationaux et des institutions académiques aux Etats Unis afin de définir des standards pour une architecture logicielle à base de composants pour le calcul scientifique à haute performance. Le forum CCA [AGG+99, BEKE02] a pour objectif la définition d'un ensemble minimal d'interfaces standards que doit fournir aux composants un framework de composants efficace et qu'il doit pouvoir composer les composants pour obtenir une application qui fonctionne dans un environnement performant. Ces standards assurent l'interopérabilité entre les composants développés par différentes équipes dans différentes institutions. La motivation de CCA est la collaboration entre ces équipes de recherche de calcul scientifique [LRN03] qui ont des champs d'investigation divers tels que la combus-

tion, la microtomographie [GvLM99], la simulation de plasma [WPS99] et la visualisation scientifique.

En relation avec les recherches théoriques et expérimentales, ces simulations jouent un rôle important dans le progrès scientifique particulièrement dans les domaines où les expérimentations sont coûteuses en temps et en matériel et parfois même impossibles à réaliser. Même si chacune de ces simulations a besoin de différents modèles mathématiques, de différents modèles numériques et de différentes techniques d'analyse de données, elles peuvent toutes bénéficier d'une infrastructure qui est plus flexible et extensible et meilleure pour la gestion de la complexité et le changement. Pour cela, il faut concevoir une architecture qui peut être intégrée dans la plupart des plateformes de calcul, y compris les réseaux de stations de travail, les multiprocesseurs à mémoire distribuée, les grappes (clusters) de SMP et des ressources à distance.

Dans cette section, nous présentons l'architecture CCA ainsi que quelques frameworks de composants basés sur cette architecture.

2.4.1 L'architecture CCA

Le but de CCA est de simplifier l'introduction des nouvelles technologies dans le cycle de vie des applications existantes et de faciliter la construction de nouveaux modèles. Il vise à faciliter le processus d'intégration d'outils et la traduction entre les interfaces et les structures de données qui est un processus qui demande un travail intensif. Il permet aussi des interactions dynamiques (pour ajouter des composants lors d'une simulation).

CCA est défini comme un ensemble de spécifications et de leurs relations. Comme illustré dans la figure 2.8, les composants interagissent avec d'autres composants et avec une implémentation spécifique du framework au travers des API (*Application Programming Interfaces*) standards. Chaque composant peut définir ses entrées et sorties en utilisant un langage de définition d'interfaces : le SIDL (*Scientific Interface Definition Language*). Ces définitions peuvent être déposées et récupérées dans un référentiel. L'API référence définit les fonctionnalités nécessaires pour chercher les composants dans le framework, ainsi que pour manipuler les composants dans le référentiel. De plus, ces définitions peuvent servir comme entrées pour un générateur de proxy qui génère les souches de composants qui forment les parties spécifiques des composants des ports CCA. Les composants peuvent utiliser des services de framework directement à travers l'interface `CCAServices`. L'API de configuration fournit l'interaction entre les composants et les constructeurs des applications (*Builders*).

Un framework est dit *conforme CCA*, s'il est conforme à ces standards, ce qui signifie qu'il fournit les services CCA nécessaires et implémente les interfaces CCA nécessaires. Les composants différents demandent des ensembles différents de services pour inter-opérer. Par exemple, les composants ont besoin d'une communication à distance (remote) tandis que d'autres communiquent dans le même espace d'adressage. Pour cela le standard CCA permet différents niveaux de conformité ; chaque composant suit un minimum de la conformité demandé du framework dans lequel il interagit.

Nous présentons dans la suite, les trois éléments essentiels du standard CCA qui sont les plus critiques pour le calcul scientifique à haute performance :

- Le *SIDL* (Scientific Interface Definition Language).

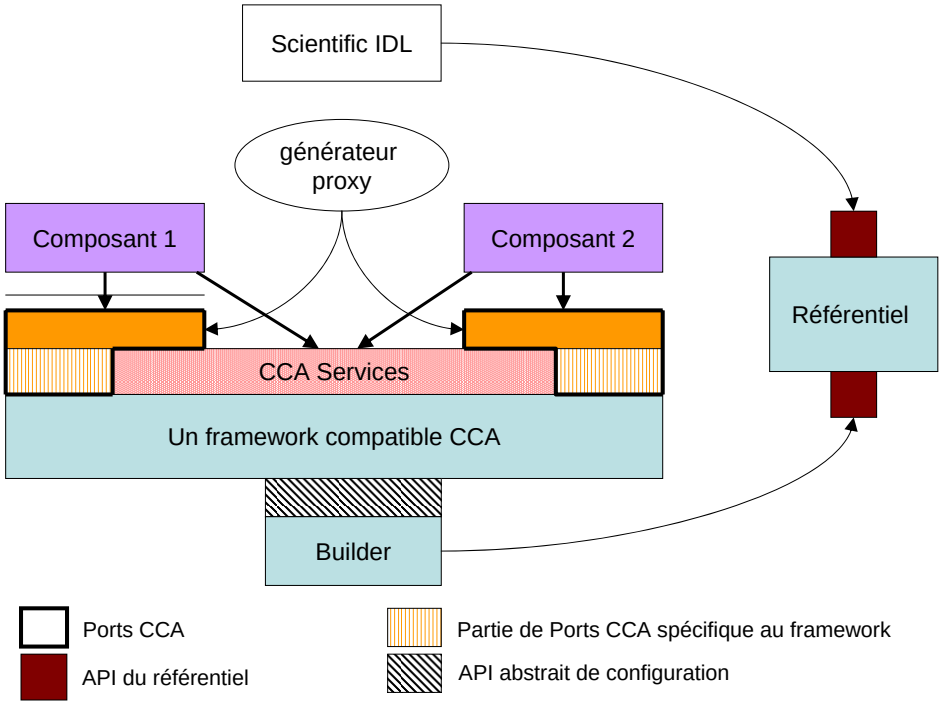

FIG. 2.8 – L'architecture CCA

- Les *Ports CCA* définissent le modèle de communication pour les interactions de composants.
- Les *Services CCA* représentent l'abstraction du framework qui est utilisée dans l'implémentation de la souche du composant.

Le SIDL

Le SIDL (Scientific Interface Definition Language) [BEKE02] est un langage de définition d'interface indépendant des langages de programmation utilisés pour définir les interfaces de composants. SIDL fournit l'interopérabilité entre langages et cache les dépendances de langage pour simplifier l'interopérabilité des composants écrits en différents langages de programmation. Avec la prolifération par le passé, des langages utilisés pour la simulation numérique comme C, C++, Fortran 77, Fortran 90, Java, Python, le manque d'interopérabilité de tels langages peut être une barrière significative pour le développement des composants scientifiques réutilisables.

Avec comme cible les architectures de composants scientifiques, SIDL doit être suffisamment expressif pour représenter les abstractions et les types de données communs, dans le calcul scientifique, tels que les tableaux multidimensionnels dynamiquement dimensionnés et les nombres complexes. Il n'existe pas un tel IDL car la plupart des IDL sont conçus pour des systèmes d'exploitation ou pour les systèmes d'information client serveur.

La conception principale de SIDL emprunte beaucoup de concepts à des standards existants comme CORBA IDL et le langage de programmation Java. Cette approche per-

met d'accroître la projection des langages et la technologie IDL existante. SIDL fournit de nouvelles capacités qui sont nécessaires pour le calcul scientifique. Il supporte les sémantiques orientée-objets avec un modèle d'héritage semblable à celui de Java, avec l'héritage multiple des interfaces et un héritage unique de l'implémentation. De nouvelles primitives ont été ajouté à IDL pour le cas des types de données complexes pour obtenir plus d'expression et plus d'efficacité lors de projection vers les langages d'implémentation. SIDL supporte la réflexion et l'invocation dynamique de méthodes qui sont des capacités importantes pour les architectures de composants. Ces mécanismes sont basés sur la conception de la bibliothèque de classes de java, dans java .lang et java.lang.reflect. Les informations de réflexion pour chaque interface et chaque classe, sont générées automatiquement par le compilateur SIDL qui est basé sur des descriptions IDL.

Les Ports CCA

Chaque architecture à base de composants est caractérisée par la manière dont les composants sont composés ensemble pour obtenir une application. Les Ports CCA définissent le modèle de communication pour les interactions de composants. Chaque composant définit un ou plusieurs ports pour décrire les interfaces du composant. Les liens de communication entre les composants sont implémentés par la connexion de ports compatibles. La compatibilité entre les ports est définie par la compatibilité du type de l'interface du port. Chaque port a deux parties : la première est une fonctionnalité spécifique au framework, indépendante du composant et qui a le même API dans chaque composant. La deuxième implémente la fonctionnalité spécifique au composant, mais est aussi indépendante du framework ; cette partie est désignée comme la souche du composant et peut être générée par un générateur de proxy à partir de la définition SIDL du composant.

CCA adopte un mécanisme d'échange d'interface provides/uses semblable à CORBA. Cette approche permet les connections qui n'affectent pas la performance inter-composants. Il permet ainsi au framework de créer des connections distribuées, si nécessaire. Dans le cas idéal, un composant attaché peut réagir aussi rapidement qu'un appel de fonction interne. Cette situation est vue comme une connexion directe. Ce type de connexion a plus de sens, quand les instances du composant se trouvent dans le même espace d'adressage. Les connections distribuées de faible couplage doivent être disponibles à travers la même interface que les connections directes de couplage fort. Ce besoin apparaît parce que les composants à haute performance sont souvent eux-mêmes des programmes parallèles. Un composant parallèle peut résider dans un unique multiprocesseur ou être distribué sur plusieurs hôtes. Les modèles de composants traditionnels de l'industrie n'ont pas de concept pour attacher deux composants parallèles. Des systèmes de recherche existant, comme PAWS [SMF02], aborde ce problème de manières différentes. Pour cela le CCA forum a introduit le modèle de port collectif pour permettre l'interopérabilité entre les composants parallèles.

Le concept de ports CCA vient du monde *data flow* (dirigé par les données) où les interactions de composants sont limités par l'acheminement de données d'un composant à un autre. Les ports CCA généralisent cette idée pour admettre les appels de méthodes et les retours de valeurs à travers cet acheminement. Les liens entre les composants sont implémentés par l'interface du patron de conception (design pattern) *provides/uses* qui

est assez flexible pour permettre des connections directes d'interfaces de composants ou des connections à travers un proxy intermédiaire pour permettre les interactions des composants distribués. La connexion par un port, dans le modèle CCA, est rélative de la responsabilité du framework ; ainsi, un composant particulier peut se trouver connecté de plusieurs manières en fonction de son environnement et de son mode d'utilisation.

Dans l'architecture CCA, les composants sont liés par la connexion de l'interface port d'un composant à l'interface port d'un autre. Il existe deux type de ports :

- port *Provides* : est une interface qu'un composant offre aux autres
- port *Uses* : est une interface qui a des méthodes qu'un composant (l'appelant) veut appeler sur un autre (l'appelé). Le composant appelant récupère l'interface Uses de Services CCA .

Les Services CCA

Les Services CCA représentent l'abstraction du framework qui est utilisée dans l'implication de la souche du composant. Cet élément de CCA fournit une définition claire des services minimaux qu'un framework doit implémenter pour qu'il soit conforme à CCA. Les services clefs de CCA sont la création des ports CCA et l'accès au port CCA qui, à son tour, permet les connections entre les composants.

Toutes les interactions entre le composant et son framework ont lieu à travers l'objet `CCAServices` du composant qui est disponible dans le framework. Le composant crée et ajoute les ports *provides* aux `CCAServices` et enregistre et récupère les ports *uses* de `CCAServices`. Le CCAServices permet l'accès à la liste des ports *provides* et *uses* et à un composant individuel par le nom de son instance. Il implémente aussi une méthode pour obtenir les ports divers et les enregistrer dans le framework.

2.5 Les frameworks du projet CCA

Les participants au projet CCA ont conçu chacun un framework [RC00]adapté à leurs besoins. Nous citons ici, des frameworks qui sont destinés aux applications sur des machines parallèles pour le calcul parallèle intensif ou pour le le calcul parallèle/distribué. Le framework XCAT [GKC+02][KBG+01] est un framework pour les applications parallèles/-distribués qui est implémenté avec les services GRID. CCAFFEINE [AAW+02][BAA04] est un framework pour les applications parallèles SPMD (*Single Program Multiple Data*). Un autre framework qui destiné aux applications parallèles est le framework du projet PAWS (*Parallel Application Work Space*[SMF02][HKM+03]. Ce framework traite le problème de redistribution de données parallèles entre les composants. Il existe aussi le framework DIVA (*Distributed VIsualization Framework*) qui est un framework pour rendre les outils de visualisation scientifique, qui existent dans les laboratoires concernés, sous forme de bibliothèque de composants.

Nous avons parlé dans l'introduction du modèle CCA, du langage de description SIDL. Le projet BABEL [Lab04] traite du problème de l'interopérabilité des langages des applications parallèles afin de construire un interpréteur du langage SIDL pour l'implémentation vers des langages de programmation tels que C++, JAVA, ou Fortran. Les frameworks de

CCA doivent avoir les fonctionnalités BABEL pour qu'il soit possible de faire coopérer les composants écrits dans plusieurs langages de programmation. Dans la suite de cette section, nous nous intéressons plus particulièrement, aux frameworks XCAT, CCAFFEINE et PAW.

2.5.1 Le projet CCAT/XCAT

Le projet XCAT [GKC+02, KBG+01, KBG+01] est un projet de l'université d'Indianna qui est un membre collaborateur du projet CCA [AGG+99]. Le projet XCAT est l'extension des projets faits par le même groupe de recherche. Les premiers projets étaient le projet CAT (*Component Architecture Toolkit)* et le projet CCAT (*Common Component Architecture Toolkit*) [VGS+99, BGVW99] dont l'objectif est d'avoir une implémentation des spécifications CCA dans un framework pour des composants adaptés au calcul sur la grille (GRID) [RRA+02].

Au départ le projet CAT était un projet indépendant CCA. L'intégration du framework CAT, pour prendre en compte les spécifications CCA, a donné naissance au projet CCAT. Ensuite, l'amélioration de framework et l'intégration des service web (*Web Services*) [GKC+02] a donné un nouveau framework de composant CCA qui est l'actuel projet XCAT. L'architecture XCAT est une architecture qui repose sur le GRID, mais avec l'intégration du modèle CCA comme modèle de composant. La composition des composants se fait à travers la connexion des "ports" de chacun des composants participants à l'application. La définition des ports *provides* et *uses* est faits de la même manière que la définition des ports CCA. La différence ici est l'implémentation et les services de framework. Dans la première version CCAT [BCD+00] le système a été construit avec HPC++ [GBJ+01] et NexusRMI [BDV+98] comme une couche de communication. Le système utilisé sur la grille est *Globus* [FK97b]. Cette version a été encore modifiée pour intégrer maintenant des services web.

XCAT, Web Services et OGSA

Les Services Web sont des interfaces aux applications qui sont accessibles en utilisant les standards Internet. Elles sont indépendantes du système d'exploitation ou du langage de programmation. XML est utilisé dans ces standards. Les Services Web utilisent des protocoles qui sont divisés en cinq couches :

1. découverte : c'est la couche d'enregistrement qui permet aux Services Web d'être découverts et publiés.
2. description : la description d'un Service Web inclut l'interface disponible, le réseau, le transport et le protocole de paquetage utilisé. Le langage *WSDL*, *Web Services Description Language* [CCMW01] est le plus répandu.
3. message : c'est a travers cette couche que les données voyagent sur le réseau. XML et le protocole SOAP [BEK+00] sont très utilisés pour ce type d'échange de données
4. transport : c'est la technologie utilisée pour le transfert de messages entre les applications. Elle inclut HTTP, SMTP et FTP.

5. framework : cette couche fournit la possibilité aux autres Services Web tels que les applications puissent construire des systèmes distribués comme par exemple le .NET et ONE.

Comme il est présenté dans la figure 2.9,le framework XCAT dispose d'une couche correspondante à chacune des couches services web :

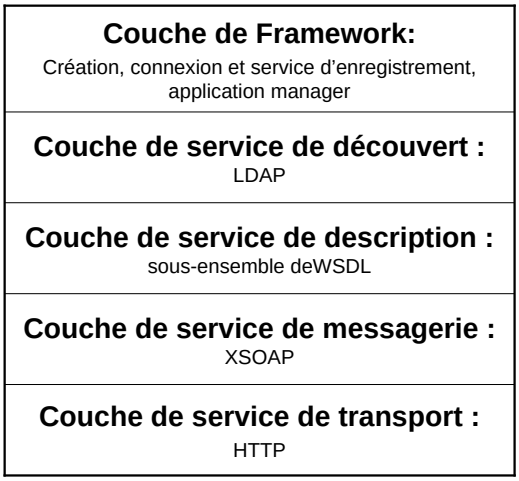

FIG. 2.9 – Les différentes couches web services dans XCAT

- framework : XCAT et CCA fournissent la réalisation de la couche framework de services web.
- découverte : les services web ont besoin d'un mécanisme de découverte pour l'introspection dans les langages de programmation. XCAT utilise un service d'enregistrement basé sur LDAP
- description : les interfaces aux composants XCAT, qui sont les ports CCA, utilisent des schémas de documents XML. Ces documents sont utilisés pour générer le code nécessaire aux utilisateurs pour ne pas entrer dans les couches les plus basses. Le code généré peut aussi manipuler la conversion nécessaire pour l'interopérabilité entre les composants basés sur C++ et Java.
- messages : XCAT utilise le système de communication XSOAP [SGGB01] pour l'échange de messages. XSOAP est une implémentation du modèle Java RMI qui utilise le protocole SOAP pour les communications.
- transport : même si SOAP n'exige pas l'utilisation d'un protocole de transport spécifique, HTTP est largement utilisé, et XSOAP utilise aussi HTTP.

L'intégration de ce framework dans un environnement GRID demande aussi d'enrichir le framework avec d'autres services qui sont spécifiques aux GRID. Les spécifications OGSA (*Open Grid Services Architecture*) [IF05] présentent les efforts pour intégrer les services à travers un environnement GRID. Elles construisent au dessus des services web un ensemble de conventions (interfaces et comportements) qui définit l'interaction des clients avec les services OGSA. Ces conventions incluent un ensemble d'interfaces standards utiles pour découvrir les méta-données des services, contrôler la durée de vie des services et créer

une nouvelle instance de service. Des spécifications de OGSA ont été implémentées dans le framework XCAT.

Dans la partie suivante, nous présentons le framework XCAT, en détaillant les services de framework.

Le Framework et les services XCAT

Le framework XCAT contient les parties suivantes :

- L'implémentation : chaque appel de méthode à distance est intercepté par le framework XCAT-Java avant d'invoquer une méthode sur le port *provides*. Cela permet au service de sécurité de s'interposer entre le port *provides* et le framework XCAT.
- Le `ComponentID` : il représente le point d'identification du composant. XCAT utilise le mécanisme de référence à distance fournit par XSOAP pour représenter un `identifiant de composant`.
- Les exceptions : XCAT fournit un modèle d'exception pour les communications entre les composants. Les exceptions sont transformées en erreur SOAP sur les lignes et en exceptions spécifiques au langage avant traitement.
- Les services XCAT : les spécifications CCA ne disent pas explicitement comment les composants sont découverts, créés ou connectés. XCAT est une approche basée sur les services pour résoudre les problèmes cité ci-dessus. Ces services sont modifiés pour être conformes aux spécifications de OGSA.

La figure 2.10 illustre un processus typique de XCAT avec ses services. XCAT fournit les services suivants :

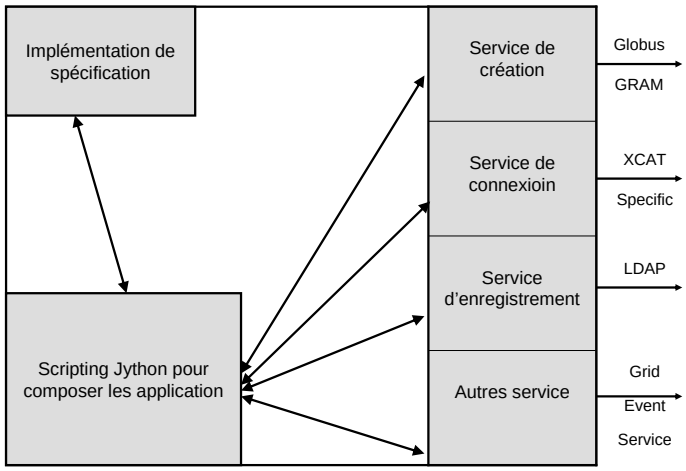

FIG. 2.10 – Un processus typique de XCAT

- **Le service de création** : c'est un composant spécifique à une implémentation qui permet à un composant d'instancier d'autres composants. Ce service exporte un port *provides* qui a comme fonctionalité de créer des instances des composants et de supprimer les instances. Un composant peut être instancié dans le même

espace d'adressage du composant créateur ou dans un espace différent. Pour cela les mécanismes d'instanciation disponibles sont :
- le protocole GRAM (*Grid Resource Allocation Management*) qui est fourni comme une partie de l'outil GLOBUS.
- (R/S)SH : les composants peuvent être créés en utilisant les protocoles RSH et SSH.
- exec : les composants sont créés par le mécanisme de lancement d'un processus `exec`.
- proc : les composants sont créés dans le même processus que le composant créateur.

- **Le service de connexion** : le service de connexion est un mécanisme spécifique au framework par lequel les composants instanciés établissent une communication à travers leurs ports. Toute les méthodes de ce service utilisent le `ComponentID` pour identifier un composant pour la connexion. Le service fournit les fonctionnalité suivantes :
  - Connect/Disconnect : le composant peut utiliser ce service pour connecter et déconnecter ses ports aux autres composants. Il peut aussi connecter d'autres composants dont il dispose des références.
  - Export : le service de connexion peut être aussi utilisé pour exporter (déclarer) les ports d'un composant dont le `ComponentID` est connu.
  - Provides Uses : la méthode `provideTo()` fournit un port *provides* qui appartient au même composant en tant qu'un port *uses* appartenant à un autres composant.
- **Le service de nommage/registre** : le service de nommage ou registre permet aux composants d'enregistrer pour eux-même des références qui peuvent être récupérées ultérieurement. Le service de nommage est basé sur le *Java RMI Registry API*. Ce service fournit les dispositifs : `bind` et `rebind` pour lier les références distantes d'un composant avec le registre ; `list` et `lookup` pour permettre aux utilisateurs d'explorer le registre et les liaisons courantes ; et `unbind` pour enlever une liaison existante.
- **Le service de gestionnaire de l'application** : c'est un composant XCAT générique et scriptable qui peut agir comme un contrôleur des applications. Sa fonctionalité est décrite par le script qu'il charge et exécute dans un interpréteur Jython (bibliothèque de liaison Java pour le langage de script Python).

2.5.2 L'approche SPMD avec CCAFFEINE

Le framework CCAFFEINE est une extension des frameworks orientés-objets pour le calcul parallèle comme POOMA [Rey97] et OVERTURE [BBD+98]. Ces derniers sont des frameworks orientés-objets mais ne sont pas basés sur une approche composant. Dans l'approche composant, les composants sont vus comme des participants égaux dans la construction d'une application plutôt que des éléments appartenants à une hiérarchie d'héritage comme dans l'approche objets.

Ce framework est basé sur le modèle CCA et donc sur l'implémentation des interfaces CCA ainsi que sur les services minimums pour assurer la connexion entre les composants et les aider à être instanciés et composés dans l'environnement.

Le framework met l'accent sur le patron (pattern) SPMD (*Single Program Multiple Data*) du calcul parallèle pour le choix du calcul haute performance. Un traitement SPMD peut être défini comme un programme identique pour chaque processus participant où les données qui sont traitées par ce programme, diffèrent dans chaque processus. Dans ce framework, le programme est divisé en composants CCA. Chaque instance du composant CCA est représentable par un pointeur unique dans une mémoire locale. Le composant communique à travers les ports (le mécanisme *provides/uses*) avec les autres composants qui sont dans le même espace d'adressage et communique au travers d'un protocole processus-à-processus (comme MPI) avec l'ensemble SCMD (*Single Component Multiple Data*) des composants correspondants. Il est supposé ici qu'il n'existe pas de connexion entre les ports ou de messages passés entre des composants qui sont instanciés dans des processeurs différents.

Le but principal de la création du framework CCAFFEINE est de :

- créer un framework et un modèle de programmation qui supportent l'architecture de calcul à mémoire distribuée et à passage de messages de type SPMD.
- fournir la plupart des services du framework à travers d'autres composants.

Des suggestions ont été pris en compte pour le modèle SCMD :

- une mémoire distribuée avec passage de messages
- dans une instance SPMD du framework, tous les composants et ports existent sur tous les nœuds et n'importe quel port utilisé par le framework sur un nœud sera utilisé sur tous les nœuds.
- le framework s'exécute et les composants interagissent dans un seul thread d'exécution sur chaque nœud.
- le support de liaison temps réel (et chargement dynamique) des composants est nécessaire pour simplifier la maintenance du framework. Les composant chargés dynamiquement sont chargés dans des tableaux de symboles privés pour réduire les erreurs de liaison.
- une personne (ou, un script) de contrôle.

Le standard SCMD (Single Component Multiple Data) pour le calcul haute performance

Pour la portabilité inter-framework, les composants SCMD ont besoin de standards pour réaliser des tâches communes. CCAFFEINE présente une liste de facilités pour permettre une meilleure utilisation du modèle de programmation de l'architecture SCMD.

- **un ensemble de services pour le contexte parallèle :** Dépend des méthodes de communications parallèles que le composant utilise et donc les besoins sont diffèrent pour PVM et MPI. En PVM, l'instance du composant sur un nœud est capable de trouver les autres composants de l'application SPMD. Dans les composants basés sur MPI, et parce que MPI n'est pas fourni comme une bibliothèque de composants, tous les composants supposent premièrement que le MPI_{Init} (initiation de l'environnement MPI) a été déjà appelé, parce que les composants n'ont pas de corps principaux `main()` et que le framework a besoin d'appeler MPI_{Init}, s'il veut l'utiliser. Deuxièmement, les composants supposent qu'un connecteur peut être obtenu à travers un port. Le port peut être fourni par un autre composant ou par le

framework lui même.

- **une convention d'un port d'exécution :** une fois qu'un ensemble de composants est assemblé, il faut un mécanisme de lancement de l'exécution. Ceci est fait fourni avec l'aide d'un port spécial, le port *GoPort*, qui est invoqué par le framework. Le composant implémente donc l'interface de ce port. Cette implémentation fonctionne quand le composant est lancé par le framework.
- **une convention de paramètres de configuration de composants :** pour configurer un composant afin de le contrôler, CCAFFEINE a le fait en introduisant des ports de paramètres. Le composant peut implémenter et fournir les interfaces d'écoute (listener) nécessaires pour être averti par le framework lors du changement des paramètres.
- **un service d'événements de connexion des ports :** plusieurs composants parallèles ont besoin d'exécuter des processus pendant la phase de connexion. Cela est réalisé par le service d'événements de connexion qui émet un événement lors de la connexion. Il a été implémenté dans CCAFFEINE par les ports.
- **un composant gestionnaire de données parallèles :** ce type de composant est spécifique au domaine comme le domaine du *Data Warehousing* dans lequel il existe un porteur de données qui gère les données structurées de l'application. Pour cela il faut développer des ports et des composants de données afin de fournir les service requis.
- **l'échange de données parallèles de M-processus à P-processus entre frameworks :** quelques applications peuvent coupler des applications SPMD disjointes à travers un champs commun de données. Une portion de l'application s'exécute sur N nœuds et une autre portion s'exécute sur P nœuds. Souvent, un programme unique SPMD enjambe tous les nœuds M+P en utilisant simplement des sous ensembles de nœuds dans l'abstraction SPMD fournit par la bibliothèque de choix de passage de messages. La méthode alternative pour créer et coupler ces applications distribuées n'a pas besoin d'un seul framework qui enjambe les nœuds M+P. C'est un composant qui enjambe (par son mécanisme interne) deux applications qui s'exécutent dans des frameworks différents.

2.5.3 Interactions collectives et le transfert de données dans PAWS

PAWS (*Parallel Application Work Space*) est une bibliothèque logicielle d'API et une application de contrôle qui peut être utilisée pour lier des applications parallèles pour qu'elles partagent des structures de données parallèles d'une manière simple et efficace. Il est constitué deux éléments : l'interface de programmation d'applications (API) PAWS et le contrôleur PAWS.

Les applications qui utilisent les API PAWS peuvent partager leurs structures de donneés avec les autres programmes parallèles. Les applications peuvent avoir un nombre différent de processeurs et utiliser différentes stratégies de placement de données parallèles, et peuvent même être écrites dans des langage différents en utilisant des systèmes temps-réel ou des bibliothèques de passage de messages séparées. Le contrôleur de PAWS est

utilisé pour coordonner les connections initiales entre les composants et les structures de données dans un composant et pour maintenir une base de données de connections et des composants actifs

Les invocations collectives et les transferts de données

Les composants collectifs (*Collective components*) sont une collaboration de processus multiples qui représentent logiquement un seul calcul. Pour être efficace et utile, une abstraction, qui représente les composants collectifs, doit être capable d'accroître la création des processus multiples mais, en même temps, de laisser l'utilisateur manipuler le composant collectif comme une seule unité logique. Cette fonctionalité est fourni par les ports collectifs (*Collective Ports*) à travers lesquels les composants collectifs interagissent. Les ports collectifs sont une extension des ports CCA. Un cas commun très spécial de l'interaction collective est le transfert de données distribuées d'un composant collectif à un autre. Cette opération implique une sorte de traduction de données, par exemple des opérations de redistribution de données, ou de formatage. Un composant de traduction (*Translation Component*) est un composant qui est dédié à la traduction de données distribuées entre deux composants collectifs. Le composant *MxN* implémente un schéma très populaire pour traduire des données distribuées sur M processus, en données distribuées sur N processus.

Les auteurs ont préféré construire un composant pour jouer ce rôle, au lieu de le faire sous form d'un service de framework accessible facilement. La raison pour laquelle. Ce choix a été fait„ est que les programmes parallèles travaillent typiquement sur plusieurs et différents formats de données et des nouveaux formats de données sont conçus chaque jour. Il est donc impératif que les composants de traduction soit modifiables facilement, ce qui est fait en leur donnant l'aspect d'un composant ordinaire. En même temps, il est important qu'ils puissent être invoqués aussi efficacement par le *framework* que par d'autres composants. De plus, le processus de traduction est souvent une composition de différentes opérations de traduction. Parce que plusieurs combinaisons peuvent être demandées et qu'il n'est pas pratique de les fournir toutes, il est nécessaire d'incorporer un mécanisme de combinaison efficace des composants de traduction dans des méta-composants. Pour cela la conception de PAWS est basée sur des composants de traduction non modifiables qui sont fixés dans le framework.

Les invocations collectives et les retours de résultats

Les ports collectifs sont associés à un ensemble de processus dont chacun peut se trouver dans un état différent en respectant le cas général des interactions de composants. Etendre le mécanisme de l'invocation aux port collectifs, demande l'identification des états dans lesquels chaque groupe de processus est capable de répondre aux interactions. Ces états précisent aux programmeurs quelle est la garantie du framework qu'ils peuvent avoir pour implémenter les composants et précisent aussi au framework quelles fonctionalités il doit fournir pour qu'il soit capable d'interagir avec des composants spécifiques. De plus, il est demandé au framework, qui implémente l'abstraction des ports collectifs, d'être capable de gérer ces états là, qui peut impliquer des opération de synchronisation, de résolution

de conflits entre les groupes de processus chevauchés et de fourniture des invocations non-bloquantes.

La représentation de données et le transfert MxN

PAWS définit une représentation flexible de données, le `DataField` , qui peut être utilisée pour représenter n'importe quelle structure de données rectiligne dense et distribuée, composée d'éléments arbitraires définis par l'utilisateur et non distribués. `DataField` n'est pas conçu pour l'utilisation par le programmeur, son but est de couvrir les structure de données utilisées par des paquetages indépendants et d'en extraire les informations nécessaires pour faire la traduction MxN. L'interface de `DataField` est décrit comme suit :

```
template <calss DataType>
class DataField{
 public:
   DataField();
   DataField(const std::string& dataFieldName,
             Paws::Domain& gDom,
             Paws::order ord=PAWS_ROW_MAJOR);
   DataField(const std::string& dataFieldName,
              Paws::Domain& gDom, Paws::Domain &lDom,
              DataField* userDataPtr,
              Paws::order ord=PAWS_ROW_MAJOR);
   layoutInfo& layoutInfo();
   actualDataBase* actualData();
   void addDataBlock(DataField* d, const Paws::Domain lDom);
   void update(Paws::Domain& gDom,
               std::vector<Paws::Domain> lDom,
               std::vector<void*> dptr);
};
```

Les informations sur l'emplacement de données réelles sont portées par `actualData` qui est une liste de pointeurs mémoire contigus correspondant aux informations de domaine qui sont portées dans `layoutInfo`. En se basant sur cette représentation de l'accès aux données, PAWS implémente une gamme de composants de traduction. Ils comprennent le composant `MxN` qui fournit le transfert efficace de données distribuées sur M processus en données distribuées sur N processus, transforme les données en données plus grande ou plus petite en taille par l'ajustement de la granularité, change le template (calibre) de distribution, change l'ordre de données d'une représentation par la ligne en représentation par colonnes, et bascule la structure de données sur des dimensions sélectionnées. L'implémentation d'un composant de traduction qui opère dans un environnement distribué implique le calcul d'un programme qui détermine quelle partie de données associées à un processus donné est envoyée à un processus spécifique au destinataire.

La séparation des informations de la présentation des données réelles donne à PAWS une flexibilité dans la manipulation de données. Les composants MxN sont un exemple de l'utilisation des composants de traduction. L'utilisation d'un composant MxN implique les étapes suivantes : instancier le composant de traduction approprié, le connecter à une structure de données telle qu'il est possible d'en extraire les informations appropriées pour le calcul du programme, et invoquer ce composant par l'agent(un composant ou un framework) qui veut partager les données. Le composant MxN de PAWS est représenté par l'interface suivante :

```
class MxN {
 public :
  int translate(GlobalId<DataField> &local,
                GlobalId<DataField> &remote);
}
```

Les composants de traduction peuvent être invoqués par un composant client qui souhaite envoyer des données vers un autre composant collectif et vice versa. Cependant, les sémantiques de telles invocations sont limitées. Pour les étendre, PAWS considère l'utilisation d'une abstraction semblable au canal push/pull de CORBA combiné avec la fonctionalité de traduction des composants. Cela étend la fonctionalité de transfert de données pour inclure des fonctions utiles comme le buffering, le backup et la réponse en traitant plusieurs clients et fournisseurs.

2.6 Comparatif

Dans les sections précédentes nous avons présenté quelques environnements de développement et d'exécution à base de composants pour les applications parallèles/distribuées. Les objectifs et le type d'applications ainsi que les systèmes cibles sont différents selon chaque environnement. Il est important donc de positionner chaque environnement par rapport à des critères généraux pour pouvoir comparer ces environnement. Pour cela, nous présentons les points les plus importants dans chaque environnement selon les critère suivants :

2.6.1 L'efficacité

l'un des point essentiel dans la construction d'un environnement pour des applications parallèles/distribuées est de pouvoir obtenir une meilleur efficacité malgré l'utilisation de plus en plus d'outils pour rendre la tache de développement et de réutilisation plus aisée.

GridCCM GridCCM repose sur le modèle de composant CORBA. L'efficacité des composants GridCCM est donc liée au modèle CORBA. Pour cela GridCCM a essayé d'améliorer cette efficacité en s'appuyant sur l'amélioration de communication entre les composants. Nous remarquons qu'il existe au moins deux middleware à utiliser CORBA et MPI.

MPI pour les communication entre les codes parallèle en mode SPMD dans un composant, et CORBA pour la communication entre les composants. L'efficacité se mesure donc dans la connexion entre les composants et le passage entre ces deux couches de communication. A priori, selon [DPP03], rien ne garantie qu'une implémentation de CORBA peut coexister avec une implémentation MPI. Pour cela les auteurs ont conçu PadicoTM [DPP03] qui est un framework d'intégration et un middleware de communication et d'exécution. Il permet à plusieurs middleware et systèmes, comme CORBA, MPI ou SOAP d'être utilisé au même temps. l'implémentation CORBA utilisé est le MICO CCM [AF05]. L'expérimentation montre des mesures de latence qui est la somme de l'invocation MICO CORBA et le barrière MPI. Les variation de latence est due aux variations de temps que prend la barrière MPI en fonction de nombre de nœuds. Pour cela on estime que GridCCM n'a pas de surcoût signifiant dans la redistribution de données. L'implémentation de CORBA utilisée, MICO CORBA fait toujours une copie de données pendant une communication CORBA même si la source et la destination sont dans le même espace de mémoire.

ProActive ProActive repose sur le modèle de composant Fractal. Un composant est formé d'un ou plusieurs objets actifs. Les connections entre les composants sont de types seuls ou collectifs. Dans le cas de composant primitif, l'objet actif peut être invoqué par une simple invocation de méthode. Dans le cas d'un composant collectif ou un composite il existe une traduction de cet appel par le contrôleur. L'efficacité de l'exécution dépend de la manière dont le contrôleur traduit l'appel à l'objet actif interne. ProActive utilise le langage Java donc pas de traduction d'appel au niveau d'inter-opération entre des objets écrits en plusieurs langage de programmation. Les coût des appels entre les objets sont de l'ordre d'un coût d'appel d'une méthode Java. Dans cet environnement, il n'y a pas besoin d'utiliser plusieurs couche de middleware pour les communication entre les composant et ou les groupes d'objets. On est dans un cadre de communication entre JVM a travers le protocole IIOP de RMI.

CCA Le CCA est un standard qui est destiné aux application parallèle et distribué pour le calcul scientifique. Le mode d'interaction entre les composants CCA repose sur le patron de conception de flux de contrôle *uses/provides*. Pour CCA, la connexion entre les composants prend la forme d'un appel de fonction. Ceci est idéal dans le cas où les deux composant sont dans le même espace de mémoire. Dans le cas de port collectifs comme dans le framework CAFFEINE ou PAWS pour la redistribution de données il existe encore de traduction de l'appel et la couche de communication choisi. L'utilisation des composants qui sont écrits dans plusieurs langages de programmation sont encore plus coûteux. Dans ce dernier cas, il existe une surcoût de l'utilisation du système intermédiaire pour la traduction entre les différents langages de programmation. Le framework CAFFEINE est une implémentation de CCA avec MPI comme système de communication. Cela signifie qu'il n'existe pas un autre middleware de communication entre les composants. Etant donné que le coût de l'appel entre deux composants est de l'ordre d'un appel de fonction, l'efficacité de l'exécution dépend directement de l'efficacité de la couche MPI.

2.6.2 La composition et la hiérarchie de composition

GridCCM GridCCM étend le modèle CORBA CCM pour définir un modèle de composant pour les composants parallèles. La composition à partir des composants n'est pas traité. Cependant le composant parallèle dans sa définition est une composition des composants séquentiels. Il n'est pas indiqué s'il peut y avoir une composition hiérarchique. Le modèle de base CCM est un modèle de composants à plat.

ProActive En utilisant le modèle de composants *Fractal*, *ProActive* permet la hiérarchie de composants avec le contrôleur(c.f. 2.3.3. Il est possible de composer des composants de base pour obtenir ce qu'on appelle un composant composite qui est à son tour un composant compatible avec les autres composants, que ce soit des composants de base ou des composants composites.

CCA CCA fournit un modèle de composant qui est plutôt pour des composants de grosse granularité. Un composant CCA peut être une application entière. Il n'existe pas de hiérarchie de la composition. Comme dans le modèle de composants CCM, les composants ne peuvent pas être composé à partir d'autres composants. Si un composant utilise d'autres composants il faut passer par les ports.

2.6.3 Dynamicité et configuration

GridCCM CORBA CCM n'offre pas dans ses spécifications un moyen d'instanciation ou de connexion de composants à la volé. Certains environnements basé sur CCM tente d'ajouter au modèle des spécifications lié à la dynamicité des opérations sur les composants. GridCCM n'offre pas cette possibilité et pour ajouter un composant à l'ensemble des composants participants à l'application il faut arrêter l'application, ajouter le composant et ensuite démarrer l'application. Dans un travail relatif au projet GridCCM , il existe une possibilité d'adaptation de composants au changement de l'environnement dans lequel ils sont exécute. Cet adaptation est faite en définissant un composant dit "composant parallèle auto-adaptable" qui peut changer son comportement selon le changement de l'environnement [ABP04]. La configuration des composants est fait selon le modèle CCM.

ProActive Le framework de composants de ProActive est basé sur le modèle de composant Fractal qui offre une possibilité de réconfiguration dynamique de l'applications et des composants. Dans les spécifications de Fractal, on suppose pouvoir changer dynamiquement un composant serveur qui est connecté à un composant client. Pour faire cela il faut déconnecter le client et supprimer le serveur et puis remplacer le serveur et ensuite reconnecter le client à ce nouveau serveur. Cette méthode est considérée récursive et il est supposé qu'elle ne change pas les états des composants concernés. Tous les appels de méthode sont suspendus jusqu'à ce que les composants redémarrent.

CCA La configuration et la dynamicité de connexion et de composition des composants est l'un des point le plus importants dans cette architecture. CCA propose un modèle d'événements de connexion et de configuration simple et utile. Un composant dans sa

conception peut intervenir selon ses besoin et demander au framework un ajout, une suppression ou une connexion à certains composants. Cela est fait sans arrêter l'application. Les connections entre les ports des composants CCA sont considérées comme des activités d'exécution. Les ports de CCA sont semblable à ceux de CCM mais en CCM les connections sont des activités de composition.

2.6.4 Inter-opérabilité de frameworks

Les modèle de composants CCM ou Fractal ne propose pas de spécifications sur l'interopérabilité entre plusieurs framework. CORBA en tant que middleware propose une interoperabilité entre les application utilisant ce middleware. Le modèle de composant par contre n'offre pas un moyen de faire inter-opérer deux implémentations de ce modèle. Il n'existe pas de règles précises pour faire cet interopérabilité. GridCCM n'a pas de politique d'interopérabilité avec d'autres framework de composant basé sur CCM.

ProActive avec l'implémentation du modèle Fractal n'offre pas non plus la possibilité de l'interopérabilité avec d'autres framework Fractal.

CCA propose des spécifications concernant l'interopérabilité entre plusieurs frameworks basés sur le modèle CCA. Dans la conception d'un framework CCA , il est considéré comme un composant qui propose des services au travers des ports. L'utilisation d'un framework de framework de composants (c.f. 1.3.4) est possible avec cette considération. Les travaux de CCA [LGC05, LGK06] montre qu'il est possible de faire inter-opérer des différents frameworks CCA ensemble.

2.7 Conclusion

Dans ce chapitre, nous avons présenté plusieurs environnements de développement et d'exécution pour des applications parallèles/distribuées à base de composants. Nous remarquons qu'il existe deux tendances dans la construction de ce type d'environnements : la première est de se baser sur un standard industriel, l'étendre pour obtenir un environnement dédié aux application parallèle/distribué et ainsi prendre en compte les aspects spécifique à ces environnement comme discuté dans la section 2.1. C'est le cas de *GridCCM* (c.f. 2.2) et *Proactive* (c.f. 2.3). La deuxième tendance est d'établir un standard servant de base pour construire des environnements pour ce types d'applications. C'est le cas de l'architecture *CCA*(c.f. 2.4). Nous avons choisi d'aller dans cette voie. Les prochains chapitres détaillerons cette démarche.

Chapitre 3

L'environnement CCADAJ

3.1 Introduction

Nous avons souligné, dans les chapitres précédents, l'impact de l'approche composant sur la conception et la construction d'applications parallèles/distribuées.

La construction d'une application à base de composants nécessite une architecture logicielle dans laquelle les composants sont fabriqués, composés et ensuite exécutés afin d'obtenir le fonctionnement voulu. Cette architecture logicielle,que nous avons décrite dans la section 1.3.4, est constituée de plusieurs couches. Nous nous sommes intéressés au niveau du framework qui fournit les services nécessaires pour la construction de l'application et la composition de composants. Le modèle de composants sur lequel nous nous basons est CCA. Comme nous l'avons dit dans le chapitre 2, nous avons choisi ce modèle CCA parce qu'il répond aux besoins de la construction des applications de calcul scientifique (c.f. section 2.4). Le modèle CCA doit être implémenté en Java afin de l'intégrer dans la plate-forme des couches les plus basses de l'environnement CCADAJ. Le framework est en effet construit au dessus d'un environnement d'exécution des applications Java distribuées. Cet environnement est l'environnement DG-ADAJ (*Desktop Grid - Adaptive Distributed Applications in Java*)

Dans ce chapitre, nous présentons la plate-forme ADAJ incorporant le framework de composants que nous avons nommé CCADAJ (CCA-ADAJ framework).

Vue générale de CCADAJ

CCADAJ (CCA-ADAJ) a été conçu comme un environnement d'exécution basé sur le langage de programmation Java, qui inclut un framework permettant la construction d'applications à base de composants. Notre framework de composants est basé sur les normes du modèle CCA. La plate-forme ADAJ (Adaptive Distributed Applications in Java)[Bou03, BOT01a, FT02] est un environnement d'exécution pour applications distribuées en Java et est construite au-dessus de *JavaParty* [PZ97, HMP97, NPH99]. Ce dernier permet l'exécution transparente des applications distribuées Java où l'accès aux objets est distant.

L'environnement CCADAJ se présente comme décrit dans la figure 3.1. C'est un environnement comportant plusieurs couches :

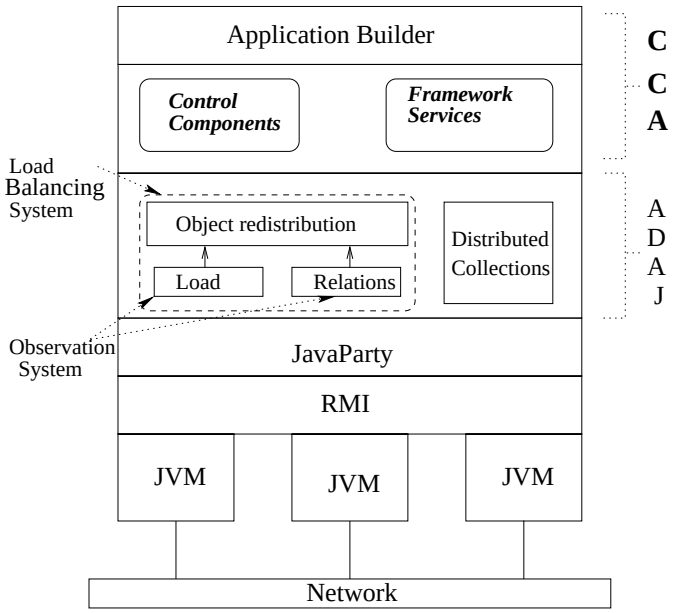

FIG. 3.1 – L'environnement CCADAJ

- JVM : la machine virtuelle Java dont l'intérêt est d'offrir une base homogène pour les applications distribuées sur le réseau et de masquer en conséquence l'hétérogénéité de la plate-forme.
- RMI (Remote Method Invocation) [Sun98c] : les outils RMI permettent aux objets qui sont distribués sur plusieurs machines de communiquer entre eux en utilisant le mécanisme de souche/squelette.
- *JavaParty* fournit un environnement pour l'exécution d'applications distribuées sur des stations de travail connectées par l'intermédiaire d'un réseau. JavaParty introduit le concept d'objets distants (*remote objects*) qui peuvent être distribués de manière transparente. JavaParty offre également un mécanisme de migration des objets à condition qu'ils ne soient pas actifs.
- ADAJ fournit des outils nécessaires pour exprimer la programmation parallèle distribuées ainsi qu'un mécanisme d'équilibrage de charge qui est basé sur la redistribution des objets et qui exploite le résultat de l'observation des activités des objets et des charges des stations.
- le framework CCADAJ fournit un environnement pour construire et assembler des applications à base de composants. Les services du framework aide le programmeur à construire son application dans un atelier de composition et d'assemblage afin d'obtenir l'application voulue. Des composants de contrôle sont conçus également pour aider à la conception et à la construction d'une application parallèle.

Dans la suite, de ce chapitre nous présentons les différentes couches de l'environnement CCADAJ en mettant l'accent sur la mise en oeuvre du framework de composants.

3.2 Les couche RMI et JavaParty

RMI a été introduit dans la section 1.2.2. Dans cette section nous expliquons plus en détail le mécanisme et le fonctionnement de JavaRMI.

3.2.1 Mécanisme RMI

Le protocole RMI permet de créer des objets distants dont les méthodes peuvent être appelées à distance.

L'implémentation de RMI est basée sur les notions de *souche* (*stub*) et de *squelette* (*skeleton*). Les étapes de l'invocation d'une méthode sur un objet distant sont représentées dans la figure 3.2 ([RD00]).

Pour que l'objet distant soit rendu accessible à distance, il est *exposé*, tout d'abord, dans un serveur d'objets (aspect réalisé de manière implicite, par l'extension de la classe *java.rmi.server.UnicastRemoteObject*) (opération 1, appelée *exposition d'objet distant*). L'exposition de l'objet consiste à le rendre accessible dans le serveur d'objets. Ensuite, pour pouvoir être localisé, il est inscrit dans un serveur de noms[2] qui gère une table d'associations de références et de noms (opération 2, appelée *publication d'objet distant*).

Du côté client, le talon, correspondant à l'objet distant, est récupéré par une interrogation du serveur de noms (opérations 3 et 4). Le talon simule l'objet distant : il offre les mêmes fonctionnalités que l'objet distant. Lors de l'appel d'une méthode sur le talon (opération 5), ce dernier se connecte sur le serveur hébergeant l'objet distant et émet ensuite, vers ce serveur une suite d'octets comprenant les identificateurs de l'objet distant et de la méthode appelée, suivis des arguments sérialisés (opération 6). Cette opération est appelée *les marshalling*.

Du côté serveur, une fois l'objet localisé par le serveur, son squelette se charge de désérialiser les arguments et d'appeler la méthode souhaitée : cette opération est appelée *unmarshalling* (opération 7). Le squelette simule un appel local du côté serveur et engendre l'exécution de la méthode sur l'objet. Une fois la méthode exécutée, c'est le squelette qui reçoit le résultat (opération 8). Il se charge de le retourner en direction du talon qui a transmis l'appel. Cette transmission est réalisée par marshalling et la suite d'octets transmise contient la forme du résultat, valeur ou exception, suive de sa valeur sérialisée (opération 9). Finalement, le talon reçoit ce résultat qu'il reconstruit (par unmarshalling) avant de le retourner au client (opération 10).

Les outils de programmation offerts par le modèle d'objets répartis JavaRMI imposent un style particulier de programmation, respectant les étapes suivantes :

- déclaration des interfaces des objets distants, pour rendre accessibles les services de l'objet distant,
- implémentation des interfaces des objets distants, pour définir les fonctionnalités des objets distants,
- définition d'une application serveur accueillant les objets distants (création et publication des objets distants),
- définition d'une application cliente en utilisant les objets distants.

[2]le serveur de noms est lancé par l'outil Java *rmiregistry*

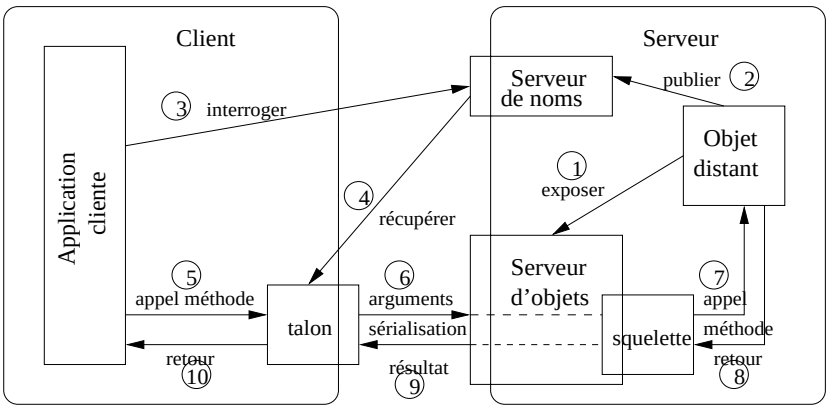

FIG. 3.2 – Le schéma de l'environnement d'exécution Java RMI

L'interface de l'objet distant doit être connue par le client et le serveur, contenant la définition des services accessibles à distance. Les fonctionnalités de l'objet distant sont définies du côté serveur, ainsi que l'application serveur. Le client définit l'application qui utilise les objets distants.

3.2.2 JavaParty

JavaParty est un environnement de programmation et d'exécution pour les applications Java sur les grappes de stations accueillant des machines virtuelles Java.

L'environnement de programmation *JavaParty* permet de transformer un programme Java en un programme distribué, et permet d'identifier les objets nécessaires au déploiement de l'application sur les machines virtuelles de l'environnement distribué. L'identification est réalisée grâce au mot clé `remote` introduit dans le langage. Ces objets sont appelés objets *remote*(objets distants). La compatibilité avec le langage Java est réalisée grâce à un précompilateur spécifique.

JavaParty offre un espace d'adressage partagé, permettant aux objets distants d'être placés dans des machines virtuelles Java différentes. En palliant les inconvénients du modèle d'objets répartis RMI, JavaParty cache les mécanismes d'adressage et de communication à l'utilisateur et traite de manière interne les exceptions réseau. Ainsi, le programmeur n'est pas censé concevoir ni implémenter des protocoles de communication explicites.

JavaParty introduit dans son environnement de programmation une nouvelle sémantique d'objets distants, appartenant à une classe distribuée (*classe remote*). Une classe distribuée est la transposition d'une classe Java dans un environnement distribué. Ses instances, les objets remote, ont deux caractéristiques principales :

- ils sont accessibles à distance (à partir de tout l'environnement distribué),
- ils sont migrables (peuvent se déplacer d'une machine virtuelle à une autre, en conservant la cohérence).

Le modèle d'objet JavaParty

Deux types d'objets sont retenus dans l'environnement JavaParty : les objets remote et les objets locaux, détaillés par rapport à leur sémantique, la création et l'accès, la gestion de la partie statique et passage des paramètres.

Caractéristiques des objets distants

- **Sémantique :** Les objets distants sont accessibles depuis tout l'environnement JavaParty sans les exporter explicitement ou les publier dans un service de noms comme en RMI.
- **Création et accès :** La création et l'accès aux objets distants sont syntaxiquement similaires avec ceux des classes Java. Les instances sont créées n'importe où dans l'environnement c'est-à-dire dans n'importe quelle JVM en utilisant le mot clé *new*. Les méthodes ou attributs d'instance d'objets distants peuvent être accédés comme s'ils étaient des objets Java locaux. Il n'existe pas de traitement d'exceptions supplémentaires autres que celles déclarées par le programmeur.
- **Partie statique :** Similaires aux classes Java, les classes distantes ont aussi une représentation, à l'exécution, dans l'environnement distribué et sont accessibles au travers des constructions identiques à celles de Java.
- **Passage des paramètres :** Les objets distant, arguments ou résultats des invocations de méthodes, sont passés par référence comme pour tout objet Java.

Caractéristiques des objets locaux

- **Sémantique :** Les classes et objets locaux ont le comportement suivant : les instances des classes locales sont liées à la machine virtuelle où elles ont été créées. Elles ne peuvent pas être référencées par d'autres machines virtuelles de l'environnement et ne sont pas migrables.
- **Création et accès :** La création et l'accès aux méthodes et attributs des objets locaux sont identiques à ceux de Java, valables à l'intérieur de la machine virtuelle dans laquelle l'appel est effectué. Dans le cas d'un migration, une copie de l'objet est créée.
- **Partie statique :** Une classe locale est chargée et initialisée séparément (à la demande) dans toute machine virtuelle qui participe à l'environnement distribué JavaParty. L'accès à des méthodes et attributs statiques concerne seulement la classe chargée dans la machine virtuelle courante.
- **Passage des paramètres :** Les objets locaux peuvent être passés en paramètre pour une méthode distante et peuvent être retournés comme résultats, le passage étant réalisé par copie.

La migration des objets

La transparence de localisation d'objets remote est une caractéristique importante pour le développement d'applications réparties. Pour être plus efficace dans l'exécution transparente, deux solutions existent : un bon placement lors de l'instanciation et une politique de migration pendant l'activité de l'objet, au cours de l'exécution de l'application.

En JavaParty, le placement est soit implicite (aléatoire pour les instances et cyclique pour les objets classes), soit explicite, par la spécification de l'utilisateur.

Ce placement ne répond pas aux problèmes apparus lors des fluctuations dynamiques de l'exécution. La migration d'objets remote est une solution pour adapter la distribution aux besoins de localité de l'application. En JavaParty, des outils nécessaires pour la migration sont offerts. Un objet remote peut migrer vers une machine virtuelle arbitraire de l'environnement d'exécution, les références précédentes peuvent encore être utilisées, grâce au mécanisme de *forwarding*.

Une contrainte importante est imposée pour migrer avec succès un objet remote. Un objet remote ne peut pas migrer tant qu'une méthode est en cours d'exécution. Une méthode est considérée en cours d'exécution si son code a commencé à être exécuté par le processeur, mais n'a pas encore été fini.

Création et Migration d'objets distants

Les classes et les objets distants en JavaParty sont introduits à l'aide du mot clé *remote*. Le code JavaParty n'est pas directement compatible avec le code Java, parce que le mot clé qui marque les objets remote n'appartient pas au langage Java. Un autre compilateur (voir figure 3.3), fourni par JavaParty, génère du code compatible avec la machine virtuelle Java. Toutes les classes générées par le précompilateur de JavaParty sont utilisées de manière transparente par le programmeur.

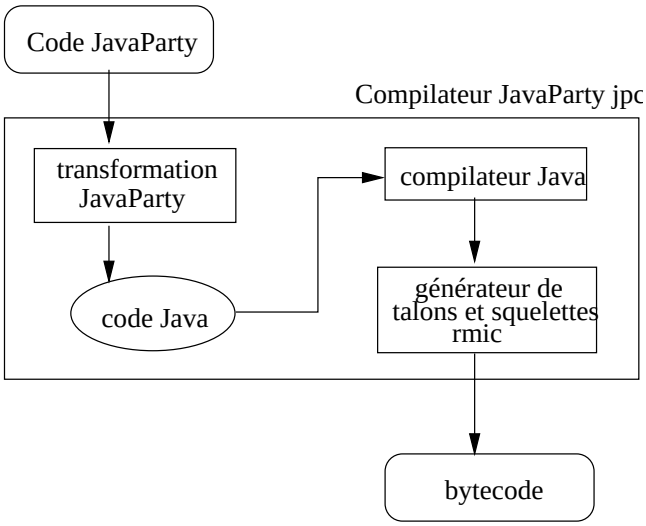

Figure 3.3 – Compilateur JavaParty

En Java, un objet est créé sur la machine virtuelle sur laquelle le *new* a été exécuté. Deux solutions existent pour instancier un objet à distance dans un environnement distribué :

- le constructeur est appelé sur la machine locale et l'objet ainsi créé est ensuite migré sur la machine destinataire,
- le constructeur est directement appelé sur la machine cible.

JavaParty a choisi la deuxième solution, à cause de la complexité de la migration, du nombre d'appels distants et du coût de la migration.

La migration en JavaParty est réalisée au niveau d'un objet distant. Les étapes de la migration en JavaParty, se résument :

– à la construction de l'état des données,

– au transfert de l'état sur la machine destinataire.

Le contrôle de la migration permet d'acheminer les invocations de méthodes sur un objet, arrivant pendant sa migration, par le mécanisme de redirection. Ainsi, un objet qui migre laisse une trace de son déplacement par l'intermédiaire d'un proxy, et les méthodes arrivant après la migration de l'objet seront redirigées, grâce au proxy, vers la nouvelle localisation de l'objet. Les méthodes arrivant pendant la migration elle-même ne sont pas exécutées, elles sont gardées chez l'appelant jusqu'à ce que le proxy de l'objet soit mis à jour.

L'environnement d'exécution

L'environnement d'exécution JavaParty est composé par plusieurs JVM qui sont accueillies sur une même ou sur différentes machines physiques (stations). Les machines physiques utilisées sont connectées en réseau.

L'environnement d'exécution est formé par un composant central (*RuntimeManager*) et des machines virtuelles appelées *locales* (*VirtualMachine*), qui s'enregistrent auprès du composant central. L'application distribuée est lancée dans une nouvelle JVM et la distribution des objets distants est décidée par le composant central, en fonction de la politique de placement.

La figure 3.4 montre le schéma des dépendances de différents composants de l'environnement distribué d'exécution, leur description est détaillée ci-dessous.

RuntimeManager Le RuntimeManager est le composant central d'un environnement distribué en JavaParty. De manière interne, il détient :

- une liste des machines virtuelles appartenant à l'environnement distribué, accessibles à distance,
- une table des objets classes (utilisés pour l'initialisation de la partie statique d'une classe),
- une table des objets constructeurs (utilisés pour l'initialisation de la partie instance d'un objet distant).

Ce composant central démarre et arrête l'environnement distribué et gère également le lancement de l'application dans l'environnement. Il enregistre les nouvelles machines virtuelles et leur offre des services de type envoi d'un objet constructeur ou d'un objet classe. Si une nouvelle classe est chargée dans l'environnement distribué pour la première fois, l'objet classe correspondant n'est pas disponible sur la machine d'instanciation, ni auprès du composant central. Dans ce cas, le *RuntimeManager* se charge de la création d'un nouvel objet classe.

VirtualMachine La machine virtuelle *VirtualMachine* offre les services suivants :

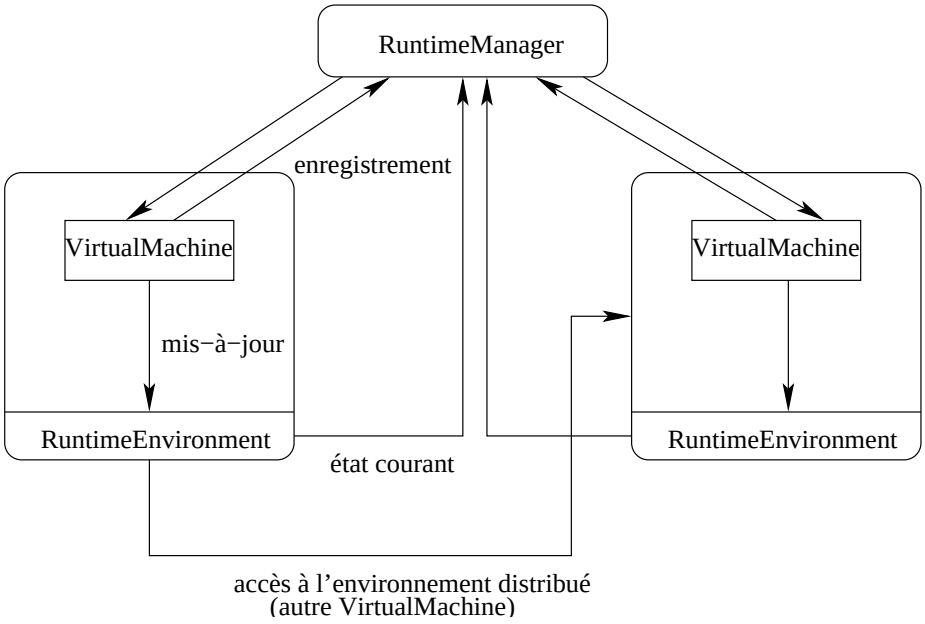

FIG. 3.4 – Le schéma de l'environnement d'exécution JavaParty

- création d'un nouvel objet constructeur pour une classe distante sur la machine virtuelle locale,
- création d'un nouvel objet classe pour une classe distante sur la machine virtuelle locale,
- initialisation de l'environnement local d'exécution (présenté dans la suite).

Lors de la migration d'un objet distant, la machine virtuelle gère la création de la nouvelle instance (à travers le mécanisme de désérialisation).

Les machines virtuelles sont identifiées par des entiers, commençant par zéro.

RuntimeEnvironment L'environnement d'exécution local *RuntimeEnvironment* permet l'accès aux objets distant et locaux dans l'environnement distribué, ayant des références vers la machine virtuelle locale et le composant central (voir figure 3.4). Essentiellement, le *RuntimeEnvironment* offre les services de :

- récupération d'un objet classe particulier (en passant éventuellement par le composant central s'il n'existe pas localement),
- récupération d'un objet constructeur particulier (passage possible par le composant central).

Toutes les méthodes offertes par l'environnement local d'exécution sont statiques, accessibles par le biais d'un appel local, dans le cadre de la machine virtuelle.

Déploiement de l'application

Une application distribuée JavaParty s'exécute dans l'environnement présenté auparavant, composé de plusieurs machines virtuelles. Le déploiement de l'application dépend

de la politique de placement des objets distant en JavaParty à l'instanciation, et des demandes explicites de migration au cours de l'exécution de l'application.

A l'instanciation, le placement des objets distants peut être décidé de trois manières différentes :

- Si le programmeur décide de ne pas se soucier du problème de placement des objets, l'objet est placé de manière aléatoire, sur une des machines virtuelles de l'environnement d'exécution.
- Des classes de distribution peuvent être développées et installées dans l'environnement d'exécution, pour orienter le placement des objets distants et des objets classes. De cette manière le programmeur interagit avec l'environnement, sans changer son application. Ce type de distribution est un paramètrage de l'environnement d'exécution, explicitement donné par le programmeur.
- Les décisions d'allocation peuvent être influencées directement par des appels de l'API JavaParty dans le code de l'application. Dans ce cas, chaque instanciation d'un objet distant est accompagnée d'une indication sur la localisation voulue pour l'objet (sous la forme de numéro de JVM). L'inconvénient de cette approche est le mélange entre le code du programmeur et le code de distribution. Cette manière manque de flexibilité par rapport aux caractéristiques de l'environnement (ressources, etc.).

Le placement, à l'instanciation, des objets distants peut être changé pendant l'exécution de l'application par des directives de migration des objets d'une machine virtuelle à une autre.

3.3 La plate-forme ADAJ

La conception d'une application parallèle distribuée et son exécution efficace sont des aspects étudiés surtout dans le contexte des réseaux hétérogènes, dynamiques et de grande taille. Rendre la conception plus aisée et la plus indépendante possible du support d'exécution, assurer une distribution automatisée et optimisée de l'application sur la plateforme d'exécution sont les problèmes soulevés auxquels ADAJ [FTD03, Bou03] apporte une réponse.

Les applications visées sont dynamiques et irrégulières. Par rapport aux algorithmes statiques dont l'exécution est connue par avance et ne change pas quelles que soient les données d'entrée, les algorithmes dynamiques sont caractérisés par une variation d'exécution de l'application en fonctions de plusieurs paramètres. L'irrégularité d'une application est liée au type du graphe de dépendance entre les objets de l'application. Pour le cas d'une application statique, le graphe est complètement prévisible. Lors d'une application irrégulière, le graphe est :

- sémi-prévisible : il est possible de structurer l'algorithme dans une succession d'étapes exécutées en parallèle (appelées *phases*). L'irrégularité dans ce cas est liée au nombre de phases ou au nombre de tâches créées dans chaque phase.
- imprévisible : le graphe est corrélé aux données et ne peut être construit que lors de l'exécution des tâches. La même situation se retrouve dans les applications coopératives où le nombre d'acteurs et leurs actions peuvent varier de façon imprévisible. Dans ce cas, l'irrégularité est difficile à maîtriser.

Les solutions envisagées pour une conception aisée, et une exécution efficace, reposent sur l'introduction, au niveau applicatif, des frameworks (cadres logiciels) ou bibliothèques de développement et, au niveau des middlewares, des mécanismes qui peuvent dynamiquement adapter aux modifications du support d'exécution et aux évolutions des calculs, la granularité des traitements, le degré de parallélisme et la distribution.

Dans ce cadre d'ADAJ (Adaptive Distributed Applications in Java), les objectifs sont les suivants :

- simplifier le travail du programmeur en cachant des problèmes liés à la gestion du parallélisme, en fournissant une interface de programmation (API[3]) complète pour que le programmeur puisse mener facilement le développement des applications parallèles,
- faciliter le développement des applications et permettre leur déploiement automatique ou quasi-automatique dans des environnements a priori hétérogènes, de manière la plus transparente possible pour l'utilisateur,
- assurer une exécution efficace des traitements, par des mécanismes d'équilibrage de charge inter et intra-applications.

ADAJ est ainsi conçu comme un environnement de programmation et de déploiement d'applications Java distribuées et parallèles.

Dans le contexte complexe des applications irrégulières et dynamiques, c'est-à-dire dont le comportement est imprévisible, qui s'exécutent dans un environnement distribué, ADAJ offre des outils qui permettent de bénéficier de l'existence de plusieurs processeurs. L'utilisation des opérations parallèles peut fournir une grande efficacité et augmenter la performance d'une application Java dont les traitements sont décomposables explicitement en sous traitements indépendants.

Les objectifs de ADAJ ne sont pas de faire de l'extraction automatique du parallélisme, mais de proposer un modèle de programmation parallèle, apportant un maximum de transparence, à la fois pour simplifier le travail du programmeur (en cachant les problèmes complexes dus à l'écriture d'un programme parallèle) à travers les APIs et pour améliorer l'efficacité d'exécution des applications.

Conçu comme un support d'exécution d'applications orientées objets parallèles et distribuées, ADAJ est une plateforme s'exécutant dans un environnement multi-utilisateurs, multi-applications pour un ensemble de stations de travail inter-connectées par un réseau, stations capables d'accueillir des machines virtuelles Java. ADAJ est un gestionnaire de placement et de migration chargé de répartir automatiquement les applications parallèles sur un ensemble de stations.

L'efficacité de l'exécution des programmes distribuées et parallèles est obtenue, en ADAJ, non seulement à travers des outils conceptuels, mais aussi grâce aux mécanismes qui peuvent dynamiquement adapter l'exécution aux caractéristiques de l'environnement d'exécution et au comportement de l'application elle-même.

ADAJ vise à résoudre les deux types de déséquilibres à travers des mécanismes d'équilibrage de charge. La recherche d'une exécution efficace des traitements distribués passe par l'introduction de mécanismes d'observation. Ces mécanismes permettent d'acquérir une connaissance du comportement du traitement et de la plateforme support durant

[3]Application Program(ming) Interface

l'exécution. Ils s'appuient sur deux sources d'information : un dispositif d'observation des relations entre les objets distribués proposé par [BLL00], et un outil d'estimation de la charge et de son évolution [BOT01b]. ADAJ adapte la distribution de charge à la fois aux variations du support d'exécution et à l'évolution des calculs et de la quantité de données traitées. Cette approche s'associe aux outils de conception proposés pour accroître l'efficacité de l'exécution des traitements Java répartis.

ADAJ est un environnement conçu au-dessus de JavaParty, héritant ainsi des améliorations apportées au modèle standard d'objets Java distribués de RMI.

3.4 La mise en oeuvre du framework de composants CCADAJ

Un framework offre un cadre de programmation dans lequel l'utilisateur doit redéfinir certaines fonctionnalités pour pouvoir profiter des services offerts. Par rapport au modèle de conception traditionnel, le framework est plus couramment implémenté dans un langage d'implémentation. Les deux outils d'aide à la programmation, le modèle de conception et le framework, se rejoignent dans les techniques d'assistance à la conception et la construction des applications.

3.4.1 Cahier de charges

Pour réaliser un framework qui permet de fabriquer des applications à base des composants, il faut d'abord déterminer quels sont les services à définir que doit mettre en place ce framework. Les services peuvent être utilisés par deux acteurs principaux, le développeur des composants et l'utilisateur de composants (voir le constructeur de l'application). Fraemwork doit faciliter et fournir les opérations suivantes :

- l'implémentation du modèle de l'architecture logicielle.
- la possibilité de créer des instances de composants. Cette opération est appelée service de création. En même temps, il faut aussi fournir un service de destruction de l'instance qui ne participe plus à l'application.
- un mécanisme d'interaction entre les composants à travers leurs points de connection (les ports) que ce soit pour des composants qui sont locaux ou distribués sur plusieurs machines.
- l'espace de construction de l'application dite de composition, et l'interface graphique qui aide à la composition de l'application et à la connexion des différents composants participants.
- l'intégration avec l'environnement d'exécution et de déploiement.
- les composants d'aide à la construction des applications sous forme de bibliothèques de composants de contrôle ou de composants de structuration de l'application.

Les besoins du développeur des composants

Le développeur de composants doit connaître les spécifications du modèle de composant et la relation entre le framework et les composants. Il faut aussi fournir des services

pour que le composant en cours de développement puisse bien interagir avec l'environnement et les autres composants. Le modèle de composant spécifie le mode d'interaction entre les composants à travers les interfaces qui représentent les ports. Le développeur doit donc implémenter les ports et les services que fournit ce composant et le déclarer au framework. Ce dernier tient compte des informations déclarées par chaque composant afin de fournir au constructeur de l'application une vue du composant avec ces points de connection.

Les besoins du constructeur de l'application

Comme nous l'avons expliqué auparavant, le modèle de composant sur lequel se basent l'environnement CCADAJ est le modèle CCA [AGG+99]. Selon les spécifications de ce modèle, le framework fournit des services sous forme de composants ou sous forme de ports qui peuvent être utilisés par les composants de l'utilisateur ou les composants du framework.

Le développement du framework nécessite la prise en compte des aspects propres au parallélisme et la performance visée par les applications de calcul parallèles et distribuées. Le framework constitue l'implémentation des spécifications de l'architecture et du modèle de conception ainsi qu'une implémentation propre à l'environnement dans lequel les composants sont exécutés. Le framework fournit aussi des services de construction de l'application (*Application Builder*) et une interface utilisateur textuelle ou graphique permettant à l'utilisateur un accès aux composants de l'application et à l'ensemble des services du framework, afin de composer son application et l'exécuter.

3.4.2 Une implémentation Java du noyau CCA

Les spécifications de CCA supposent que les services sont soit des composants soit des ports. Le framework peut être lui même un composant avec des services fournis par ses ports. Avec cette spécification, les frameworks CCA peuvent être inter-opérables dans le cas où il y a lieu de connecter plusieurs applications construites dans plusieurs framework différents.

Dans le cycle de vie d'un composant, deux parties peuvent être identifiées : la partie fabrication (par un développeur de composant) et la partie utilisation (par un programmeur). La fabrication du composant consiste en la conception, l'implémentation et la diffusion du composant. L'utilisation consiste à composer ce composant avec d'autres composants du même modèle dans un environnement de composition. Le rôle du framework est de fournir cet environnement en respectant les spécifications du modèle et des services nécessaires pour la mise en œuvre de cette composition.

Dans les spécifications CCA nous distinguons plusieurs interfaces de définition qui doivent être implémentées par le composant ou par le framework :

- L'interface `Port` est implémenté par le composant et utilisé par le framework pour connecter les composants.
- L'interface `Component` est implémenté obligatoirement par le composant afin d'incorporer la partie framework dans le composant. Cette partie est responsable de la déclaration des ports du composant qu'ils soient des ports *uses* ou des ports *provides*.

- L'interface `Services` dont l'implémentation est l'objet utilisé par le composant pour informer le framework de ces ports.

Un framework conforme CCA doit impérativement implémenter ces spécifications ainsi que donner un ensemble de services qui sont propres à ce framework et qui prend en compte l'environnement d'exécution dans lequel les applications construites avec ce framework tournent. Nous expliquons ci-dessous le mécanisme d'interaction entre les composants à travers le framework et ses services. Cela demande de préciser certains détails concernant les interfaces : `Port`, `Component` et `Services`.

- L'interface `Port` : l'utilisation des "ports" est l'acte le plus important dans les spécifications CCA. Conceptuellement, les ports sont des interfaces. Tous les ports CCA doivent hériter de l'interface `gov.cca.Port`. La façon dont l'interface `gov.cca.Port` est définit en SIDL (*Scientific IDL*) indique que les ports CCA sont exactement des interfaces SIDL, mais en réalité les ports sont seulement implémentés comme des interfaces SIDL. En utilisant la programmation orienté objet, une forme typique d'utilisation serait d'instancier un objet qui hérite de l'interface, faire la conversion de typae sur un pointeur ou une référence à cette interface (c.f. la classe abstraite de base) et puis le délivrer à d'autres codes qui ne connaissent rien du type exact. CCA a un couplage faible entre l'utilisateur et le fournisseur d'un service de calcul. Connecter les deux se fait en plusieurs étapes

  1. l'utilisateur enregistre la demande au framework
  2. le fournisseur enregistre sa disponibilité dans le framework
  3. l'utilisateur et le fournisseur sont connectés (par des moyens variés)
  4. l'utilisateur fait ce dont il a besoin avec le service qui lui a été accordé.

- L'interface `Component` : Un composant CCA est une classe qui implémente cet interface.

```
public interface Component {
   void setServices(Services svc);
}
```

  Par définition, n'importe quelle classe qui implémente cette interface est un composant CCA. L'idée est que le framework va créer un seul objet `Services`, créer le composant et puis appeler cette méthode sur le composant en lui donnant la forme de communication qu'il désire avec le monde extérieur, au travers de cet objet `Services`.

- L'interface `ComponentID` : un composant n'interagit pas directement avec d'autres composants CCA. Il doit traiter avec l'interface `ComponentID`. Cet interface est en lecture seule, ce qui signifie que les utilisateurs et les développeurs du composant peuvent acquérir de l'information sur cette interface mais ils ne peuvent jamais changer son état interne.

```
public interface ComponentID implements gov.cca.ComponentID {
  public String getInstanceName();
}
```

Le nom de l'instance de `ComponentID` est un identificateur unique pour toutes les instances de composants encore en vie dans le framework. D'autres méthodes peuvent être définies dans cette interface selon les besoin du framework.

- L'interface `Services` : il faut souligner que la plupart des composants cachent l'objet `Services` qui leur est donné et l'utilisent pour enregistrer/désenregistrer les ports pendant leur cycle de vie et ne pas seulement en réponse à l'appel de `SetServices()`. Un composant peut ne pas fournir de ports tant que toutes ses requêtes de ports utilisateurs ne sont pas traitées. Une interface graphique d'utilisateur GUI sert comme outil de développement. Les utilisateurs peuvent ajouter des liens entre les ports déclarés pour les connecter. Si nous voulons considérer que toute l'application soit un composant CCA, il faut qu'elle implémente la méthode `setServices()`. Pour écrire un composant CCA utilisable il faut comprendre les méthodes essentielles de l'interface `Services` concernant l'enregistrement des ports utilisateurs et fournisseurs dans le framework, le détachement des ports du framework, l'accès à son propre `ComponentID`, et l'accès aux propriétés des ports.

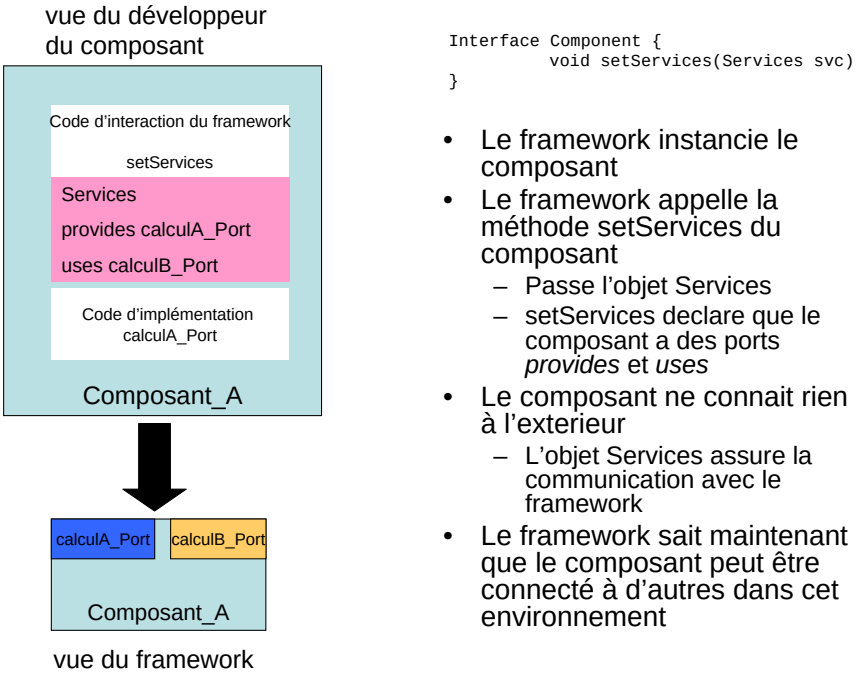

FIG. 3.5 – Vue de composant avec l'implémentation de l'interface Component

– La méthode `registerUsesPort()` est la notification au framework de l'utilisation d'un port de type spécifique. La connexion est traitée par un mécanisme extérieur au composant. Pour `registerUsesPort()`, le composant demandeur doit donner un nom sous forme de chaîne de caractères appelée *portName*. Ce nom est unique pour tous les ports que ce composant enregistre pour cet objet *Services*. Le deuxième paramètre est une chaîne de caractères qui représente le type du port. Si cette demande pour un port est satisfaite, le framework garantit que le type du

port fourni est compatible avec la demande. La méthode `unregisterUsesPort()` supprime simplement la demande de l'utilisation de ce port en se basant sur le nom du port. Cela signifie que le *`portName`* d'un port lors de la suppression doit être identique au *`portName`* donné dans son enregistrement.

- La méthode `addProvidesPort()` représente la manière dont le composant notifie au framework la disponibilité d'un port d'un type spécifique. Les méthodes d'enregistrement et de suppression des ports *provides* sont identiques à celles utilisées pour les ports *uses* à la différence que `addProvidesPort()` a besoin de la référence de l'implémentation du `Port` comme premier paramètre. Dans cet appel, le framework sauvegarde cette référence et le passe à n'importe quel port *uses* avec qui il se connecte.
- La méthode `getPort()` renvoie les propriétés associées au port *provides*. Le récepteur de ce port *provides* qui est le composant, qui a enregistré le port *uses* correspondant, est aussi appelé le *composant utilisateur*. Le composant utilisateur accède à la référence du port quand l'appel du port *uses* est retourné. L'appel de `getPort()` est bloquant jusqu'à ce que l'appel soit satisfait. Une méthode alternative `getportNonBlocking()` rend la main même si l'appel n'est pas satisfait. Quand le composant utilisateur ne veux plus de ports fournis pour lui, il le signale au framework en appelant le `releasePort()`.

Nous avons présenté précédemment dans cette section les étapes pour connecter deux composants. Puisque nous avons évoqué l'interface `Service` nous pouvons donner plus d'explications sur le mécanisme de connexion.

1. Un composant qui a besoin d'un port, enregistre ce besoin dans le framework en invoquant la méthode `registerUsesPort` sur l'objet *`Services`*. Cette invocation spécifie le type de port voulu et le nom que le composant utilise pour identifier ce port d'une façon unique.
2. Un composant qui fournit un port notifie au framework qu'il peut être utilisé pour fournir ce port là à travers l'invocation de `addprovidesPort` sur l'objet `Service`. Cette invocation spécifie au framework le type du port fourni, la chaîne de caractères que ce composant utilise pour identifier ce port et la référence à l'objet en vie.
3. Les composants utilisateur et fournisseur sont connectés par des moyens variés (qui sont définis dans le framework). Le framework garantit que les types des ports utilisateur et fournisseur sont compatibles. La chaîne de caractères du nom de port utilisée au cours des étapes 1 et 2 n'est pas incluse dans le mécanisme de connexion.
4. Le composant utilisateur accède à l'objet en vie de `Port` en appelant la méthode *`getPort()`* en utilisant la même chaîne de caractères déclarée dans l'étape 1.

- l'interface `AbstractFramework` : cet interface est implémentée par un framework CCA pour accéder aux capacités de composition des composants. La plupart des manipulations du framework sont faites avec le port `BuilderService`. Cette classe existe en tant que démarreur pour récupérer l'objet `Service` nécessaire pour rechercher le `Port` y compris le `BuilderService` du framework. Il n'est pas spécifié

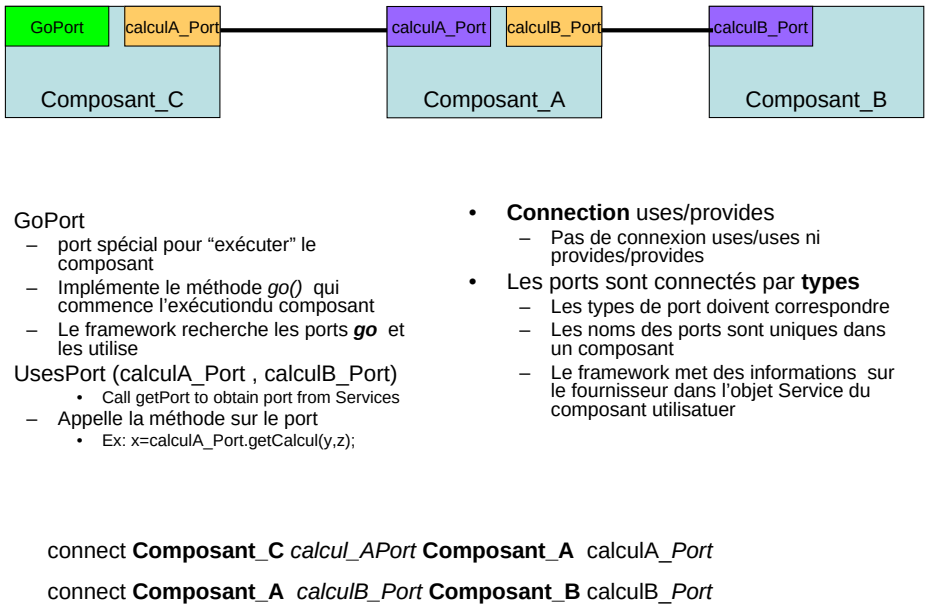

FIG. 3.6 – L'interaction entre les composants est assurée par le framework

comment cette interface et le framework sont créés. Cela dépend totalement du programmeur et du fournisseur du framework.

```
interface AbstractFramework {
    AbstractFramework  createEmptyFramework();
    Services getServices(String selfInstanceName,
                String selfClassName, TypeMAp selfProperties);
    void releaseServices(Services svc);
}
```

Exemple :
Ici on suppose qu'une instance de `AbstractFramework` est créé dans le `main()` à partir d'une implémentation hypothétique. L'idée est de permettre un échange complet du choix du framework en changeant la classe d'implémentation du framework.

```
main() {
    cca.reference.Framework fwkimpl = new cca.reference.Framework();
        // changer le fwkimpl au dessus pour utiliser des
        // implémentations différentes de cca
    AbstractFramework fwk = (AbstractFramework)fwkimpl;
    Services svc = fwk.getServices("instance0","AppDriver",null);
    // à partir d'ici, un acces aux services, composants, etc
    // à travers svc.
    ...
```



```
        // quand c'est fait
        fwk.releaseServices(svc);
        fwk.shutdownFramework();

    }
```

Les Ports par défaut de CCA

En plus du noyau de CCA , certains ports sont définis pour aider la construction d'une application.

- `GoPort` : le premier et le plus simple des ports. Ce port a une relation spéciale avec le framework. Il est utilisé comme un déclencheur de l'application. Par exemple, il peut déclencher un calcul.
  ```
  public interface GoPort extends Port{
    int go();
  }
  ```
  Le retour de l'appel de la méthode `go()` a trois valeur : 0 si il n'y a pas de problème, -1 si il y a un problème mais le framework continue a s'exécuter, -2 si il existe un problème grave qui oblige le framework à arrêter son exécution. `GoPort` est un port *provides* implémenté par le composant qui déclenche l'application (c.f figure 3.6). Plusieurs composants dans l'application peuvent implémenter le `GoPort` s'il faut déclencher plusieurs composants au début de l'application.
- `BuilderService` : cet interface est la plus compliquée des interfaces CCA. Elle définit les méthodes de création, de connexion/déconnexion et de destruction des composants.
  ```
  public interface BuilderService  extends Port {
  ...
  }
  ```
  Les composants sont créés par le `BuilderService` par la méthode `createInstance()`. Pour traiter la requête de création d'une instance, le framework va créer d'une façon invisible une instance `Service` et appelle la méthode `setServices` sur ce nouveau composant. A ce moment là, le composant a l'opportunité d'enregistrer ses ports.
  Les ports *uses* et *provides* sont connectés par la méthode `connect()`. La connexion entre deux ports doit vérifier les types des ports et non pas leurs noms. Une fois la connexion établie, le composant utilisateur peut appeler la méthode `Service.getPort` et utiliser le port fourni. Les connections peuvent être interrompues individuellement par un appel de la méthode `disconnect` ou globalement par la méthode `disconnectAll()`. Les instances du composant peuvent être détruites en utilisant la méthode `destroyInstance()`
  L'interface `BuilderService` définit également les méthode d'affectation et d'obtention des propriétés des composants (*set/get*), des propriétés des ports et des propriétés de la connexion. Les autres méthodes définies dans cette interface sont des méthodes d'obtention (*get*) des propriétés internes du framework.
- `ConnectionEventService` : Parfois, un composant CCA n'enregistre un port *provides* que si tous ses port *uses* sont satisfaits. Pour accomplir ceci, le composant a

besoin de détecter quand les requêtes de ces ports *uses* sont satisfaites et quand ces ports là sont annulés. Cela est fait à l'aide d'un service d'événement de connexion à qui est défini par l'interface `ConnectionEventService`. Ce service d'événement est inspiré du modèle d'événement Java. Pour chaque connexion, il existe un événement défini par l'interface `ConnectionEvent` avec un type spécifique.

```
public interface ConnectionEvent {
    public int getEventType();
}
```

Pour chaque événement, il y a l'émetteur d'événement et le récepteur de cet événement. L'émetteur d'événement est le service de connexion qui connecte deux composants par leurs ports. Le récepteur d'événements est un composant qui veut être notifié qu'une connexion est réalisée. Un composant qui veut recevoir un événement de connexion, doit implémenter l'interface `ConnectionEventListener`.

```
public interface ConnectionEventListener {
    public void connectionActivity(ConnectionEvent evt);

}
```

Le service d'événements de connexion est représenté en tant que port. Ce qui signifie qu'avant qu'un composant reçoive l'événement de connexion, il doit enregistrer un port *uses* de type `ConnectionEventService`. La requête doit être satisfaite et le composant appelle `Services.getPort()` pour accéder au récepteur de l'événement. Le composant qui veut émettre un événement lors d'un connexion à l'un de ses ports doit donc implémenter et déclarer un port *provides* de type `ConnectionEventService`.

```
public interface ConnectionEventService extends Port {
  public void addConnectionEventListener(
            int connectionEventType, ConnectionEventListener l);
  public void removeConnectionEventListener(
            int connectionEventType, ConnectionEventListener l);
}
```

Une fois que le composant a une connexion active, il est libre d'ajouter ou de supprimer un récepteur d'événement.

– `ComponentRepository` : le but de ce port est de retourner une liste des types de composants disponibles à instancier.

```
public  interface ComponentRepository  extends Port {
   public  Vector getAvailableComponentClasses();
}
```

3.4.3 Le Framework CCADAJ : extension du modele CCA et implémentation des services de CCADAJ

Nous avons décrit l'architecture CCA dans le chapitre 2. Dans notre comparatif avec d'autres environnements à base de composants, nous avons cité le manque dans le modèle CCA en ce qui concerne la composition des composants à base d'autres composants

(composition hiérarchique). Le CCA est conçu pour un type de composition de grosse granularité et donc ne traite pas la composition hiérarchique. En implémentant le CCA au dessus de la plate-forme ADAJ, nous pouvons profiter de l'efficacité de ce modèle et de la dynamicité de la configuration, de la connexion de la création des composants. Mais nous pouvons pas imbriquer des composants dans d'autres composants. Pour profiter plus de l'apport de la technologie de composants nous avons créé le super-composant qui un modèle d'un composants pouvant être construit à base d'autres composants. Ainsi nous avons introduit la composition hiérarchique au modèle CCA.

Comme nous l'avons décrit sur la connexion entre les ports des composants CCA, les types d'interface de port doivent etre identique lors de la connexion. Dans certain cas, nous avons besoin d'un type générique d'interface pour pouvoir connecter des composants sans préciser le type de l'interface de port au moment de la connexion. Les types sont raffiné après dans l'implémentation du composant. Pour cela nous introduisons un port spécifique que l'on appelle le port générique.

D'autre part, l'intégration du modèle CCA dans la plate-forme DG-ADAJ demande certaines mise en place de composants et d'outils pour rendre le processus de déploiement des composants possible dans l'environnement DG-ADAJ. Pour cela il était nécessaire d'introduire un composant déployable dans cet environnement et qui encapsule un code qui a éventuellement besoin d'être mis sur un site distant.

La composition hiérarchique

La programmation par composants aide à construire des applications par la composition et la connexion des composants sans changer leur contenu. Nous distinguons deux type de composants d'un point de vue architectural : le composant simple dans lequel il existe un code et que déclare des ports *uses* ou *provides* et un composant plus compliqué que l'on appelle super-composant qui est un résultat de la composition de plusieurs composants simples. Par la suite nous allons utiliser les définitions suivante :

Un composant de base est un composant CCADAJ simple qui ne contient aucune composition hiérarchique d'autres composants CCADAJ.

Un super-composant est un composant CCADAJ qui contient une composition de plusieurs composants de base ou de super-composants.

Il est nécessaire dans la programmation distribuée et parallèle à base de composants d'utiliser une composition hiérarchique pour utiliser un groupe de composants comme une seule entité de calcul et la déployer dans l'environnement d'exécution utilisé pour obtenir un résultat spécifique. CCA fournit un modèle léger de composants de grosse granularité au niveau des applications. Mais il ne fournit pas un mécanisme de composition hiérarchique. Pour cela il est important d'étendre le modèle afin qu'il prenne en compte cet aspect de la composition hiérarchique tout en gardant la compatibilité avec le modèle CCA. L'ensemble des composants regroupés dans un seul composant sera traité comme une entité unique et il n'est pas possible d'accéder aux composants qui se trouvent à l'intérieur de cette entité, ce qui est la propriété principale de l'encapsulation. Pour accéder à l'intérieur de cette entité, il faut que le développeur du composant conçoive, à l'aide du modèle de composition hiérarchique, une interface capable de servir de point d'accès aux services proposés par les composants internes dans cet entité.

La structure du super composant A l'intérieur d'un super-composant existent des composants de base CCA qui peuvent être connectés entre eux normalement. Les composants de bases ont leur propre interfaces définis selon les normes CCA afin de déclarer leurs interfaces des ports *uses* et *provides*. D'autre part, les super-composants doivent disposer d'un mécanisme pour contrôler les accès aux composants de base internes. Le mécanisme consiste à exposer les interfaces internes du super-composant à l'extérieur en gardant sa fonctionalité que ce soit un port *uses* ou un port *provides*, comme le montre le schéma dans la figure 3.7

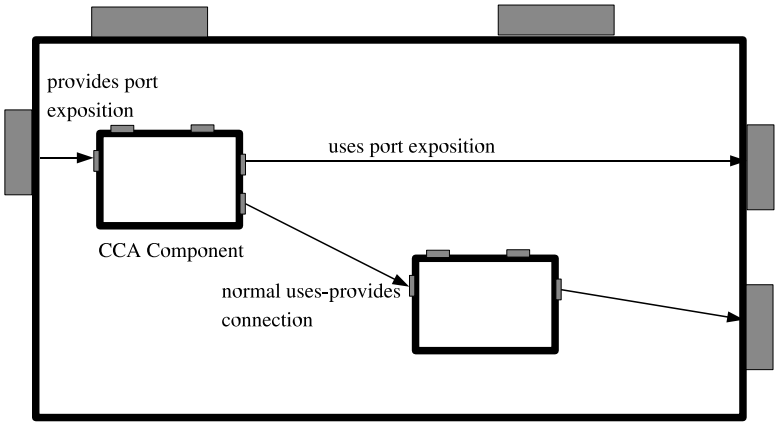

FIG. 3.7 – la composition de composants de base

Le développeur du super composant peut :
- composer l'entité "super-composant" en liant les composants de base afin qu'ils assurent une fonctionalité commune.
- cacher l'intérieur du super-composant
- définir les ports du super-composant en exposant les ports nécessaire des composants de base à l'extérieur.

En effet l'exposition des ports des composants de bases internes au super composant, se fait à l'aide du modèle de super-composant en implémentant l'interface nécessaire.

Les informations concernant les composants de base internes sont traitées de la même façon que les ports. Pour cela nous avons besoin de services supplémentaires dans un super-composant comme les services fournis par le framework. Dans un composant CCA de base, l'implémentation de l'interface `Component` permet au framework de garder des informations sur le composant et ses ports. Dans un super-composant, il est donc nécessaire d'étendre cette interface pour avoir un contrôle sur ses composants de base internes. De plus, un super-composant doit disposer d'un mécanisme d'instanciation et de connexion pour les composants internes. La figure 3.8 détaille cette structure avec les liaisons possibles de l'intérieur vers l'extérieur du super composant.

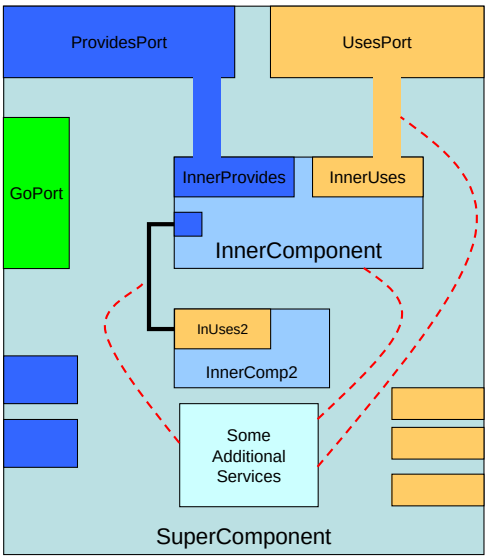

FIG. 3.8 – Le Super Composant

Un super composant donc peut être défini comme suit :

```
public abstract class SuperComponent implements Component {
  framework fwk;

  public abstract void assembleComponents(){
  public ComponentInfo instantiate(String className,
                       String instanceName)
  {...}
  public void connect(String fromName, String fromPortName,
                      String toName, String toPortName)
  {...}

  public void addUsesPort(String instanceComponent,
                       String portName, String portType)
  {...}
  public void addProvidesPort(String instanceComponent,
                          String portName, String portType)
  {...}
  }
```

```
}
```

Nous avons besoin de déclarer les ports du super-composant de façon à ce que les ports des composants internes soient projetés vers l'extérieur du super-composant puissent être connectés avec d'autres composants. Nous distinguons deux façon de projeter les ports internes selon que

- le port interne est un port *provides*
- ou le port interne est un port *uses*

Les deux cas sont illustrés par la figure 3.9 Dans le cas du port *provides* , nous avons une implémentation du port dans le composant interne. Le super-composant utilise une

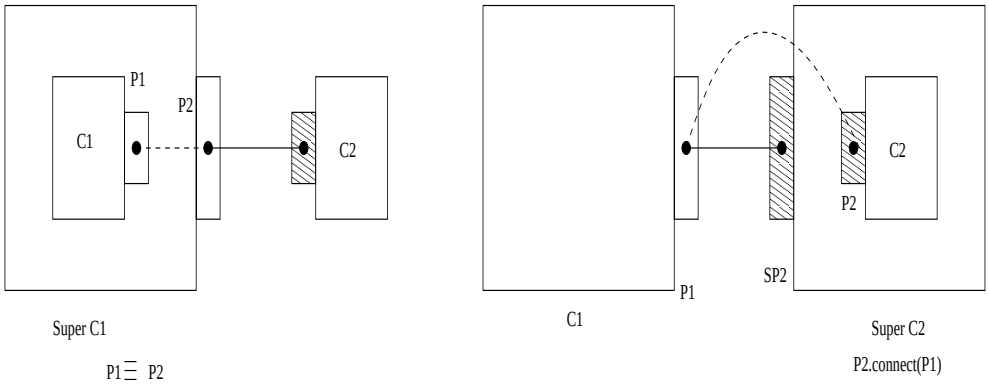

FIG. 3.9 – Projection des ports internes

instance du framework pour pouvoir composer les composants internes. Le framework du super-composant a des informations sur le composant interne. À partir de ces informations, le super-composant récupère le port *provides*. La déclaration du port *provides* est la tâche effectué par le développeur du super-composant. Il implémente la méthode `assembleComponents()` dans laquelle il déclare un port *provides* qui est le port du composant interne en utilisant la méthode `addProvidesPort()`. Dans ce cas il est facile de récupérer l'implémentation du port et de rediriger l'appel vers le composant interne car il s'agit d'une simple récupération d'une référence.

Dans le cas où le composant interne a un port *uses*, le développeur du super-composant déclare le port *uses* dans la méthode `assembleComponent()` à travers l'appel de la méthode `addUsesPort()`. Cette méthode a pour rôle de rediriger l'implémentation qui contient l'appel à la méthode du port *provides* connecté à celui-ci. L'implémentation est du genre *p.m()* telque *p* est une instance du port *uses* et *m* est la méthode à appeler.

On considère un composant *C1* qui déclare un port *providesP1* et *C2* un composant qui déclare un port *uses P2*. Les port *P1* et *P2* ont le même type d'interface, il est possible donc de les connecter selon le modèle CCA. Pour que le composant *C2* s'exécute, il doit se connecter au composant *C1*.

Nous voulons construire un super-composant *SC2* qui contient le composant *C2* et qui déclare un port *uses SP2*. Le super-composant cache ici le composant interne *C2*. Comme nous avons expliqué ci-dessus le composant *C2* a besoin d'une connexion avec un composant *C1*. Ici, c'est le super-composant qui doit se connecter au composant *C1*

à travers son port *uses SP2*. Cette connexion est utilisée ensuite pour récupérer le port *provides* connecté à *SP2*. Une fois le port *provides* connu, nous pouvons l'utiliser pour la connexion avec le composant interne. Le mécanisme est implémenté dans la méthode `addUsesPort()` qui est utilisée par le développeur du super-composant.

En fait, l'implémentation du super-composant et la connexion des composants internes se fait comme si c'était dans le framework de composition des composants de base. L'exposition des ports internes des composants de base vers l'extérieur du super-composant nécessite l'intervention du développeur pour faire cette liaison. Nous avons décidé d'automatiser le mécanisme pour le développeur, en utilisant un proxy pour chaque port interne qui joue le rôle d'un composant connecteur à l'intérieur du super-composant.

Les composants connecteurs et le mécanisme du proxy de super-composant Le langage Java supporte l'utilisation des proxys dynamiques dont l'implémentation de ses interfaces est donnée comme paramètre à la création. Lorsqu'une méthode est appelée sur l'objet *proxy*, cette méthode est convertie en un appel à la méthode `invok()` de l'objet *proxy*.

```
public class ProxyPort implements InvocationHandler {
   public static Object newInstance(Port p) {...}
   private ProxyPort(Object port) {
               this.port = port;
   }

   public Object invok(Object obj, Method m, Object[] params){
     ...
     return m.invoke(port, params);
     ...
   }
}
```

Lorsqu'un port du super-composant est déclaré (avec la méthode `addProvidesPort()` ou `addUsesPort()` du super composant), un composant particulier est instancié à l'intérieur du super composant. Ce composant va faire le lien entre le port du composant interne qu'on veut rendre accessible et le port du super composant. Ce composant intermédiaire va créer un port qui sera en fait un proxy du port à accéder. Ces composants sont définis comme suite :

```
public class ProvidePortComponent implements Component {
   public ProvidePortComponent() {
   }
   public void setPortInfo(String portName, String portType) {}
   public void setServices(Services svc) {}
}

//le cas ou on a besoin d'un port uses
```

```
public class UsesPortComponent implements Component{
   public void setPortInfo(String portName, String portType) {}
   public void setServices(Services svc) {}
   public void setProvides(Port provides) {}
}
```

Nous remarquons que dans le cas où nous avons besoin d'un port *uses*, pour lier un port interne *provides* à l'extérieur, la procédure est différente de l'autre cas car le port *provides* contient l'implémentation du service alors qu'un port *uses* c'est seulement l'appel du service d'où la méthode `setProvides()` existe dans `UsesPortComponent`.

Prenons l'exemple de l'enregistrement d'un port *uses* d'un super-composant. La méthode `addUsesPort()` du super-composant est appelée. Cette méthode instancie le composant "port" qui sert à lier les ports internes aux port externes. Ce composant va instancier en même temps un port *provides* qui sera en fait, une classe proxy sur le port *provides* du composant à connecter.

Exemple de super-composant Nous présentons ici le développement et l'utilisation du super-composant. Comme c'est illustré dans la figure 3.10, nous avons deux composants de base CCA

- le composant *Add* qui a un port *provides* du nom *addition* et un port *uses* du nom *multiplication* . Le port *addition* fourni le service d'addition deux entiers, alors que le port *multiplication* utilise un service de multiplication de deux entiers fournis par un autre composant
- le composant *Mult* qui a un seul port *provides* nommé *multiplication* et qui fournit le service de multiplier deux entiers.

Nous voulons utiliser le composant *Mult* et l'encapsuler dans un super-composant *SCMult*.

L'écriture de ce composant est fait de la façon suivante :

```
public class SCMult extends SuperComponent {
  public SCMult() {
    super();
  }
 public void assembleComponents() throws Exception {
   this.instantiate("Mult","Mult0");
   this.addProvidesPort("Mult0","Multiplication", "PortMult");
 }
}
```

Dans ce cas, l'appel de la méthode `assembleComponents` est exécutée lors de l'instanciation du super-composant expliquée ci-dessus. Il faut remarquer que la déclaration du port *provides multiplication* est la même que celle du port du composant de base `Mult`. Dans ce cas si le composant `Add` a besoin de se connecter à un service de multiplication il peut se connecter au composant `SCMult`.

La méthode `addProvidesPort()` crée le composant intermédiaire. Celui-ci possède un port *provides* du composant qui fournit le port.

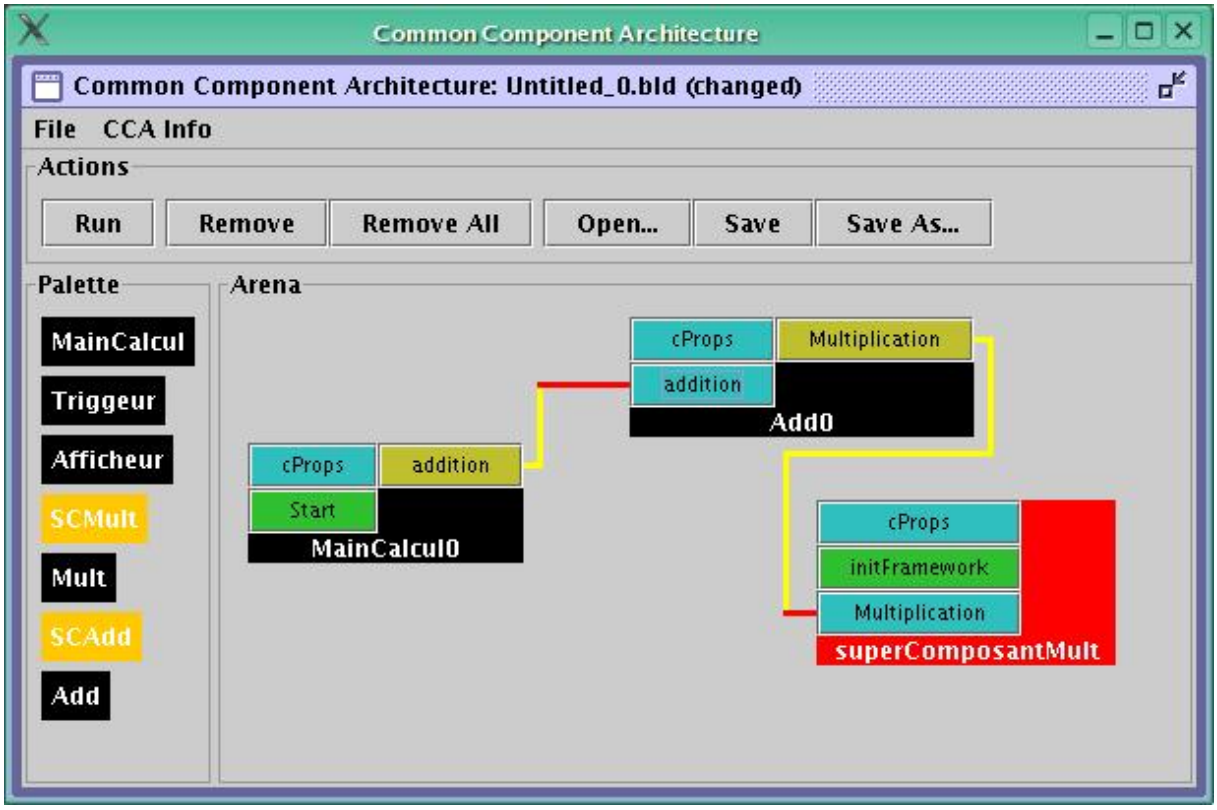

FIG. 3.10 – Exemple de super-composant

```
//connexion de deux super-composant
cfw.connect(''sc2'', ''multiplication'', ''sc1'', ''multiplication'');
```

Lors de la connexion de deux super-composants, on créé un port *provides proxy* du port *provides* du sper-composant à connecter. Le port proxy est donc du même type que le port *provides*. D'un point de vue fonctionnel l'utilisateur du super composant le voit seulement comme un composant de base avec des ports *provides* et *uses*. Il ne peut en aucun cas le modifier.

Les composants distants (remote) et déploiement dans CCADAJ

Le framework CCADAJ est construit au dessus de la plateforme ADAJ qui utilise l'environnement JavaParty pour le déploiement des objets distant sur les machines participantes dans cet environnement. Nous ne pouvons déployer les composants CCADAJ dans cet environnement, que s'ils sont compatible avec les directives de déploiement propre à ADAJ et surtout à JavaParty. Dans ce cas nous ne pouvons pas profiter des mécanisme de transparence d'exécution, de l'observation et de l'équilibrage de charge fournit par ADAJ.

Comme nous avons vu en ADAJ dans la section 3.3, les classes sont définies selon la norme JavaParty. Une classe doit être définie avec le mot clé `remote` afin qu'il soit possible d'utiliser ses objets à distance. Par conséquent, l'écriture des composants CCADAJ en

Java ne permet pas d'utiliser ce mécanisme automatique de distribution d'objets. Les composants CCADAJ doivent être donc écrits en utilisant le mot clé `remote` pour pouvoir être distribué par l'environnement JavaParty. D'où il est nécessaire de définir une structure ou un mécanisme qui rend les composants CCADAJ compatible avec JavaParty.

L'introduction de la notion *distant* (*remote*) à CCA nécessite certaines modifications du modèle de composants ainsi que des services de base du framework. Nous avons choisi de garder la compatibilité avec les composants CCA. Pour cela il faut introduire la notion de composant distant (*remote*) qui est un composant CCADAJ avec la possibilité d'être déployé dans l'environnement ADAJ.

La particularité de la programmation d'une application qui sera déployée dans un environnement ADAJ est que toutes les classes des objets distants doivent être définies avec le mot clé `remote`. Il existe plusieurs façons d'aborder ce problème :

- changer toutes les définitions des classes nécessaires dans le modèle CCA. Elles doivent être changées pour être utilisées avec le mot clé `remote`. Cette solution est simple mais il faut que le développeur du composant indique dans son code Java l'utilisation du mot clé `remote` et puis qu'il fasse une pré-compilation du code pour obtenir les différentes classe en Java.
- adopter une solution pour les composants qui seraient utilisés comme composants *distants*. Dans ce cas, il faut introduire la notion de composant distant tel que existe la notion d'objet distant.

Pour garder la compatibilité entre les composants CCA d'autres frameworks et les composants CCADAJ, il est important d'étendre le modèle CCA et y ajouter les services nécessaires pour supporter l'environnement d'exécution d'ADAJ. Les composants restent toujours des composants CCA, sauf quand on veut les rendre distants(remote). Un mécanisme a été introduit pour rendre les composants distants, sans recompiler leur source. Ce mécanisme a pour but d'encapsuler le composant voulu distant dans un super-composant qui est lui même accessible à distance. Pour cela, un modèle de composant "remote" a été conçu. Nous avons appelé le super composant remote un *Remote Container*. Le schéma dans la figure 3.11 montre le mécanisme permettant de connecter deux composants CCADAJ à distance dans un environnement ADAJ.

Le *Remote Container* est un super-composant spécial. Il est d'une part un super-composant qui encapsule un ou plusieurs composants internes, et d'autre part, il dispose de ports de type spécial que nous appelons des *Remote ports*. Le *Remote container* est caractérisé par la possibilité d'être déployé dans un environnement ADAJ dans le cas des objets distants.

Dans un environnement ADAJ, on distingue deux types d'objet : les objets locaux et les objet distants. Les objets locaux n'ont pas à être déclares comme *remote* et n'ont donc pas la possibilité de migrer entre les machines. Les objets distants doivent être déclarés comme étant *remote*, pour qu'il soit possible de les déployer à distance sur n'importe quel site et les faire ensuite migrer d'un site à un autre si nécessaire. Dans ADAJ, les composants sont déployés à l'aide de l'intégration de nouveaux services dans le framework. Ceux-ci sont responsables de l'instanciation et de la connexion des composants remote. La figure 3.11 illustre le déploiement de deux composants distants et de leur connection dans un framework CCADAJ. Nous voulons déployer deux composant CCA, A et B, dans un environnement CCADAJ. Les deux composants sont encapsulés dans deux remote contai-

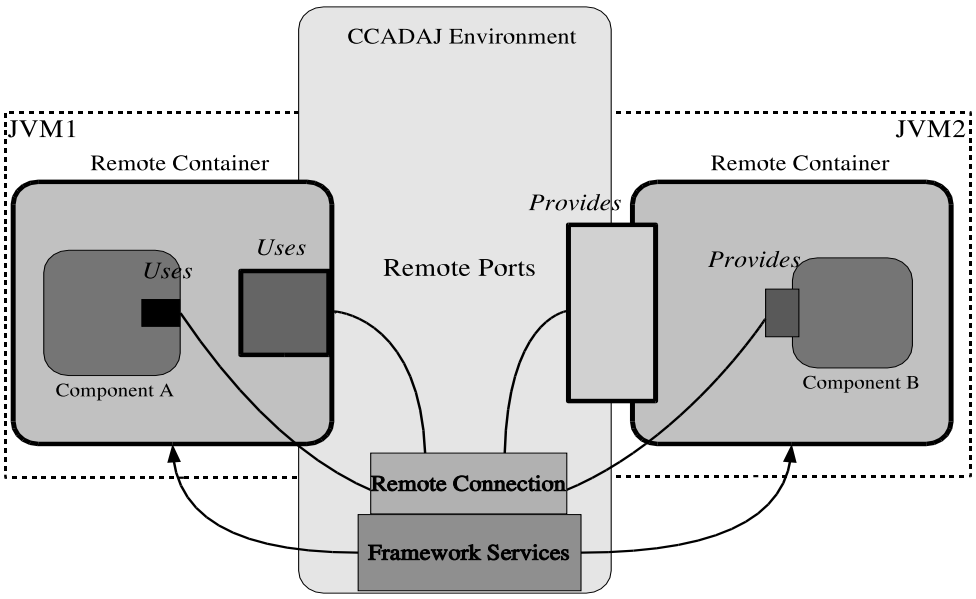

FIG. 3.11 – Remote Container

ners. Le composant A a un port *uses* qui demande un service proposé par le composant B a travers son port *provides*. Le remote container de chaque composant a un port qui est à la fois un port CCA et est caractérisé par la notion remote de ADAJ. Chaque remote container est ensuite instancié dans une JVM. L'instantiation génère les objets qui sont déployés dans cette JVM, y compris les ports des composants.

le remote container est défini comme suit :

```
public abstract remote class RemoteContainer implements Component {
  public abstract void assembleComponents();

  public ComponentInfo instantiate(String className,
                       String instanceName)
  {...}
  public void connect(String fromName, String fromPortName,
                      String toName, String toPortName)
  {...}

  public void addUsesPort(String instanceComponent,
                       String portName, String portType)
  {...}
  public void addProvidesPort(String instanceComponent,
                          String portName, String portType)
  {...}

}
```

Le port remote défini pour un remote container est le suivant :

```
public remote class RemotePort implements Port{

    ComponentFramework cpf;
    String instanceName, portName;

    public RemotePort(ComponentFramework cp,
                      String instanceNam,String portNam){
        cpf = cp;
        instanceName = instanceNam;
        portName = portNam;
    }

    public Object call(String methode,Object[] params,Class[] cs){
        return cpf.call(instanceName, portName,methode,params,cs);
    }

}
```

La définition des composants distants et des ports distants est nécessaire seulement pour déployer ces composants à distance (dans une autre JVM ou dans le cas d'une migration éventuelle lors de l'exécution).

Le composant parallèle

Définition : Un composant parallèle est un composant qui encapsule un code (traitement) parallèle distribué.

Dans la mesure où un super composant peut encapsuler d'autres composants de base ou des composants distants, il peut présneter un composants parallèle dont les composants internes sont déployés chacun sur un site dans l'environnement. Nous pouvons donc distinguer deux type de composants parallèles

- Un super-composant parallèle : qui encapsule des composants distants. Ces composants peuvent utiliser des composant de fine granularité pour effectuer certaines tâches.La figure 3.12 illustre ce cas de composant parallèle
- Un composant CCADAJ parallèle qui encapsule des objets distants. Dans la construction des composants parallèles de ce type, nous pouvons encapsuler une grosse quantité de traitement et l'utiliser par d'autres composants. La figure 3.13 illustre ce cas de composant parallèle

Mise en œuvre des services du CCADAJ

Il existe trois grande parties dans le framework CCADAJ qui assurent le déploiement des composants : la composition des composants qui sont l'instanciation et la connexion des composants.

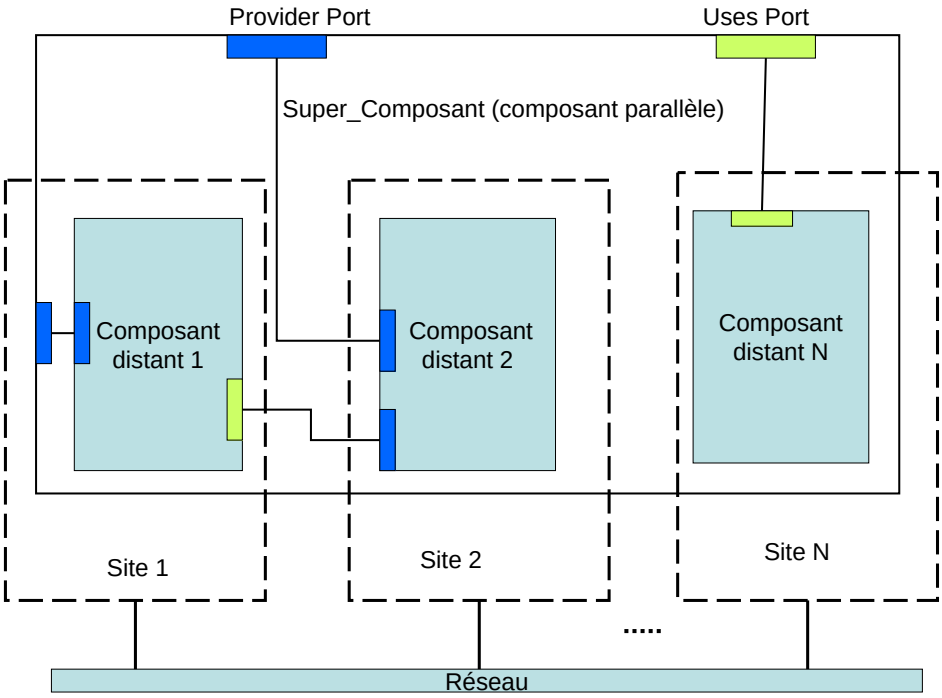

FIG. 3.12 – Composant parallèle qui contient des composants distants

- **la composition** est faite à l'aide d'un outil de composition graphique proposée par le CCA forum(c.f la figure 3.14). L'interface graphique comprend deux volets : la liste des composants récupérés a partir du référentiel de composants et l'espace de composition. Le programmeur construit son programme en glissant les composants de la liste et en les mettant dans l'espace de composition. Par ce fait, le composant est instancié et le programmeur doit lui donner un nom et, si nécessaire, le paramétrer. L'interface graphique a été modifié pour prendre en compt les super-composant.
- **l'instanciation :** est faite d'une façon automatique lors du glissage d'un composant dans l'espace de composition dans l'étape précédente. Lors de l'instanciation d'un composant, il est utile de savoir si le composant est un composant de base ou un super-composant. Cela aidera dans la connexion entre les composants.
- **la connexion** est faite par le service de connexion de base entre les composants de base et les connexions remote entre les composants remote ou entre un composant de base et un composant remote. Le service de connexion entre les composants effectue un test pour savoir si c'est un super-composant ou un composant de base.

Les services proposés par CCA ont été modifiés pour permettre la composition des super-composants et des composants remote. Pour bien comprendre l'implémentation des services de framework, il est important de comprendre le scénario de la composition d'une application qui est le suivant :

1. Au lancement du framework de construction de l'application avec un GUI, le framework cherche, dans le référentiel de composants, les composants CCA pour les mettre sur la liste des composants qui sont prêts à être composés dans une application.

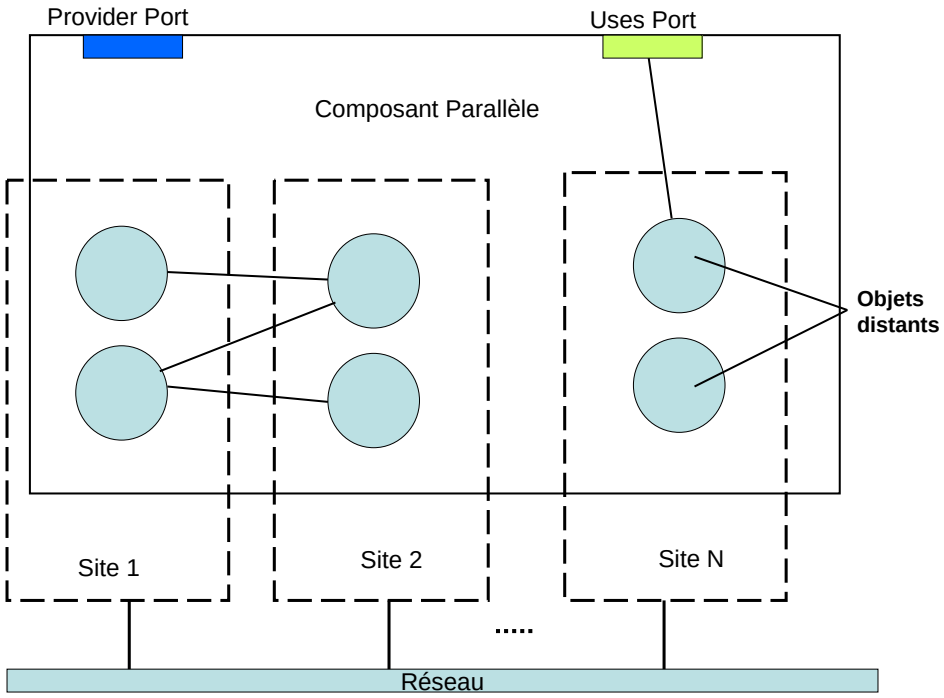

FIG. 3.13 – Composant parallèle qui contient des objets distants

2. l'utilisateur prend un composant de la liste et le met dans l'espace de composition. A ce moment, le framework exécute la méthode `setServices()` sur ce composant. Si le composant est un composant de base, alors l'exécution de cette méthode crée les instances nécessaires pour le framework notamment `Services` qui est le point de liaison entre le framework et le composant. Si le composant est un super-composant cette exécution va créer les objets différents des composants internes et exposer leurs port nécessaire sans exposer ces composants de base à l'extérieur. Si le composant est un composant remote alors la seule différence avec le super-composant est l'instanciation des ports remote déja déclarés dans ce composant en utilisant les proxys de JavaParty. Les méthodes utilisées dans l'instanciation sont :

```
public ComponentInfo instantiate(String className, String instanceName);
public Component getComponentInstance(String instanceName);
public void removeInstantiatedComponent(String instanceName);
```

3. une fois le composants instancié, ses ports sont déclarés et le framework fournit dans son espace de composition une vue de composant avec ses ports comme points de connexion avec une distinction entre les composants de base et les super-composants. Cette distinction est juste informative car elle n'affecte pas le déroulement de la composition. L'utilisateur est maintenant capable de relier les composants entre eux au travers de leurs ports. Le modèle du super composant rend transparent la liaison entre les composants de base et les composants de base internes dans un super-composant. Lors de cette connexion la méthode `Connect` est utilisée.

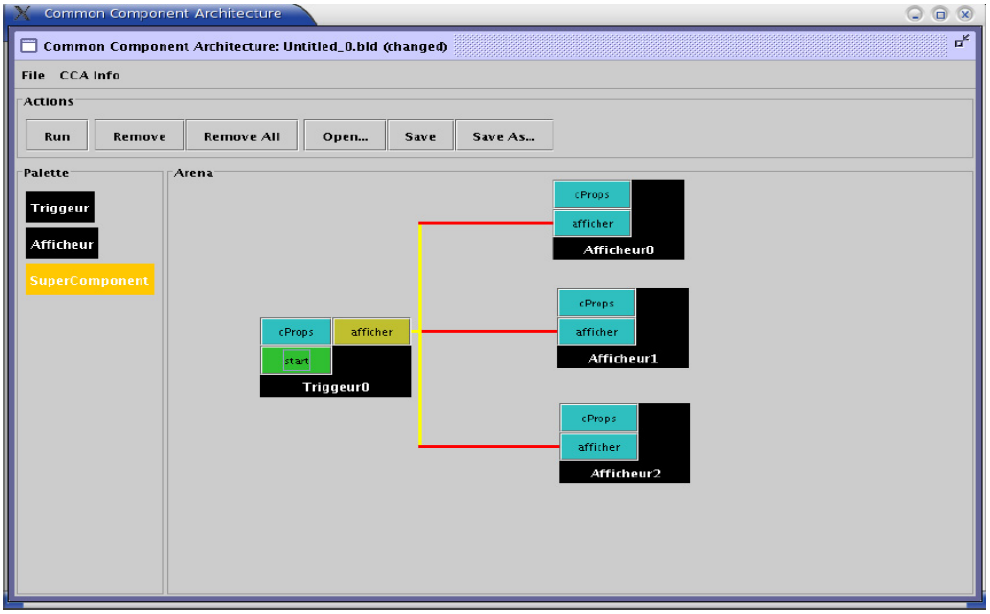

FIG. 3.14 – GUI de composition

```
public void  connect(String fromName, String fromPortName,
                     String toName, String toPortName){
            //...
  boolean isSuperComponent = false;
  CmpBox cmpTo   = ((CmpBox)instances.get(toName));
  CmpBox cmpFrom = ((CmpBox)instances.get(fromName));
  isSuperComponent = cmpTo.comp instanceof SuperComponent;
  if( isSuperComponent ) {
    Port provide = null;
    try {
      provide = cmpFrom.giz.getPort(fromPortName);
      } catch( Exception e ) {
      e.printStackTrace();
      }
  ((SuperComponent)cmpTo.comp).connect(provide, toPortName);
        }
//...
}
```

S'il s'agit de connecter un composant externe à un super-composant qui expose un port *uses* d'un composant interne, alors il faut créer une connexion entre le port *uses* du super-composant à son composant interne par le mécanisme de proxy expliqué précédemment (c.f. 3.4.3)

4. Quand les connexions sont faites, il faut déclencher l'application avec le port spécial `Go` qui est le point de connexion entre le framework et un composant qui l'implémente. Normalement le composant qui déclenche l'application implémente ce port

```
Go.
```

Service d'événements Le modèle CCA ne prend pas en compte les événements entre composants. En revanche, il existe un modèle d'événements sur les connexions entre les composants. Le framework est notifié par chaque connexion entre deux composants. Cette notification est présentée en tant qu'un service d'événements de connexion (c.f. 3.4.2). Il est très utile dans le cas de la création dynamique des ports et la connexion dynamique avec d'autres composant en fonction des besoins du composant. Pour pouvoir travailler avec les événements sur les composants, nous avons ajouté au modèle CCA ce type d'événements. Il est donc possible d'utiliser certaines actions sur les composants en fonction des événements déclenché par d'autres composants. Par exemple un composant qui a besoin d'avoir certains données en fonction de son calcul, émet un événement aux autres composants et les composants qui sont à l'écoute de l'événement répondent avec une action. Ce type d'événement est utile dans certains cas d'asynchronisme entre les composants.

Le modèle d'événements utilisé dans CCADAJ hérite directement du modèle d'événements de JAVA. Les composants CCA sont définis par leurs classes qui implémentent l'interface `Component`. Pour utiliser les événements simples entre les composants nous avons défini l'interface `ComponentListener` et la classe `SimpleEvent` Ce type d'événement est utilisé dans 4.3.4.

```
public interface ComponentListenr extends java.util.EventListener {
   void getNotified(SimpleEvenet evt);
}
public class SimpleEvent extends java.util.EventObject {

    public SimpleEvent (Object source ) {
        super(source);
    }
}
```

La génericité

La construction d'un application à base de composants nécessite parfois la définition des composants ou des ports génériques. Un composant générique est un composant qui traite les différents type de données sans que le programmeur prenne en compte les problème de connexion entre les composants ou faire attention au type de données qu'il faut passer au composant pour le traitement. La généricité est très utile dans le cas des composants de contrôle car les composants de contrôle ne traitent pas de données et il ne font pas de calcul.

Le modèle de composant CCA exige certaines règles au niveau de définition et de connexions entre les ports des composant. Nous rappelons que lors d'une connexion entre un port *uses* et un port *provides* , les deux ports doivent avoir le mêne type d'interface. Si les type d'interface sont les même alors la connexion est réussie, sinon la connexion est échouée et le framework lève une exception. Pour cela, nous nous intéressons au ports qui

sont les points de connexion. Si les ports sont génériques alors la connexion est toujours réussie, mais c'est à l'exécution que l'exception se lève, dans le cas où il y a un problème sur le type de données.

Le port générique Nous avons défini un port générique de plusieurs façons. D'abord nous avons défini des ports spéciaux pour les données génériques comme le port *GetObjectPort* pour récupérer des données sous forme d'objet et *SetObjectPort* pour placer des données de type objet. Le langage JAVA permet ce type de définition et les vrais types de données sont détectés à l'exécution. Prenant par exemple le type de port *GetObjectPort*, la définition du port est la suivante :

```
public interface GetObjectPort extends Port {

   public Object getObject();
}
```

Nous pouvons utilisé ce type de port pour récupérer des données de n'importe quel type mis dans des objets. L'implémention de cet interface est dans le composant qui déclare un port *provides* de ce type. Cette implémention implique l'utilisation des casting des types afin de raffiner les vrais types de données pour les utiliser dans des composants de calcul. Nous distinguons deux type de fonctionnalité de ports génériques

- les ports génériques d'échange de données. Dans cette catégorie, nous avons les ports *GetObjectPort* et *SetObjectPort* que nous avons vu ci-dessus.
- les ports génériques d'invocation de méthode de calculs. C'est un port dont le but est de définir l'appel d'une méthode de calcul qui implémente ce type de port. Le port *InvokerPort* est défini pour cette raison. Sa définition est :

  ```
  public interface InvokerPort extends Port {

    public Object invokePort(Object environment);

  }
  ```

  l'implémention de ce port est demandé par le fournisseur de calcul afin de répondre a l'appel de cette méthode à partir d'un port *uses*.

le port uses générique Avec le port générique, nous avons vu qu'il était obligatoire d'avoir des ports de type générique pour pouvoir connecter deux composants. Si nous avons besoin de connecter deux composants, il faut que le composants clients déclarent un port *uses* de type générique et le composant serveur déclare un port *provides* afin que la connexion entre les deux composant soit réussie. Dans ce cas si nous avons un composant déjà construit qui fournit un service à travers un port *provides* d'un type donné. Dans le cas normal nous sommes obligés de définir un port *uses* du même type que celui du port *provides* du fournisseur pour pouvoir connecter les deux composants.

Afin d'éviter la création des ports *uses* à chaque utilisation d'un composant, nous avons défini un port générique *uses*. Ce type de port est illustré par la figure 3.15

la définition du port uses générique est :

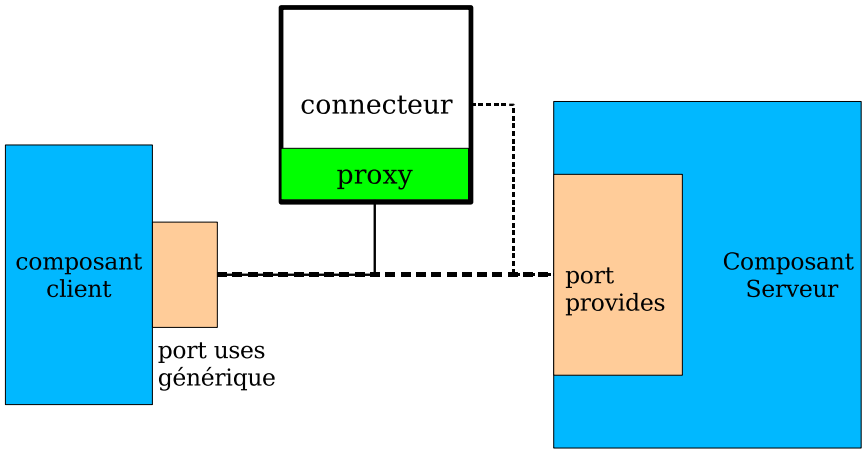

FIG. 3.15 – Le port uses générique

```
public interface GeneralUsesPort extends Port {
        public Object generalInvoke(Object[] margs);
```

Le composant qui déclare un port uses générique utilise la méthode *generalInvoke(Object margs)*. Le service de connexion détecte le type de port uses générique et crée le proxy nécessaire pour récupérer la méthode du port *provides* avec lequel ce port *uses* est en train de se connecter. Ni le nom de la méthode, ni l'appel de cette méthode sont connus à la création du composant. Nous avons utilisé le proxy dynamique de JAVA (à partir de la version 1.3) Nous avons modifié le service de connexion défini par CCA afin qu'il fasse une détection de type de port à connecter pour pouvoir utiliser le proxy du port *provides* connecté au port *uses* générique.

3.5 Conclusion

Dans ce chapitre, nous avons présente le framework CCADAJ, ses différents niveaux conceptuels et la mise en œuvre des services et des outils nécessaires pour faire fonctionner ces niveaux ensemble. La plateforme existante ADAJ est un environnement d'exécution pour des applications parallèles et distribuées. Pour mieux aider le programmeur à construire son application au dessus de ADAJ et pour rendre efficace la réutilisation du code dans ses applications, nous avons implémenté le modèle de composant logiciel CCA. Avec cette architecture, la construction d'une application est plus efficace. Le programmeur voit son code comme des composants indépendants CCA avec leurs interfaces. La connexion entre les composants est efficace dans la mesure ou le programmeur n'intervient pas à l'intérieur du composant.

L'intégration du modèle CCA n'est pas suffisante pour un programmeur qui construit son application au dessus de ADAJ. Il faut intégrer d'autres outils qui relie les différents niveaux de cette plateforme et il faut se servir des propriétés d'efficacité de la distribution et la transparence de JavaParty. Pour ce faire, le modèle CCA a été étendu au travers de deux concepts : le super-composants, le composants distant. Le super-composant, à l'inverse d'un composant de base, peut contenir un ou plusieurs composants de base ou super-composants. La composition hiérarchique est donc introduite dans le modèle CCA. L'intégration avec ADAJ doit aussi introduire de nouveaux concepts dans le modèle sans affecter la compatibilité avec d'autres frameworks basé sur CCA. Le composant distant (*remote container*) a été présenté. Par ce moyen, il est possible donc de profiter des mécanisme d'observation et d'équilibrage de charge fournis par ADAJ dans une environnement distribué avec JavaParty.

L'utilisation conjointe de ces deux concepts permet de définir un super-composant parallèle constitué de composants réparties sur plusieurs JVM. Les services nécessaires sont implémentés dans ce framework qui est un framework compatible CCA. D'autres services propres à l'environnement CCADAJ ont été implémentés aussi comme le service d'événements, le service de connexion et le service d'instanciation afin de prendre en compte les aspects de super-composant et des composants distants (remote) et rendre la construction d'une application transparente. L'utilisation de port générique a été introduit afin qu'il soit possible de connecter des ports de type générique.

Enfin, d'autres questions restent posées autour de cet environnement comme la migration de composants qui n'est pas encore implémenté contrairement à la migration d'objets. La migration est possible pour les objets distant dans l'environnement. Dans ce cas, si on utilise un composant parallèle (le cas de composant encapsulant des objets distants), il est possible de faire migrer ses objets dans l'environnement de l'exécution. Mais la migration elle-même, des composants est en cours d'étude.

Dans le chapitre suivant nous présentons des composants dits composants de contrôle qui font partie des composants de framework qui aide le programmeur à la construction de son application.

Chapitre 4

Bibliothèque de composants pour le parallélisme

4.1 Introduction

Nous décrivons dans ce chapitre les outils nécessaires qui faciliteront la tâche du programmeur pour la création d'une application parallèle à base de composants logiciels. Nous devons donner au concepteur des outils spécifiques : une bibliothèque de composants et des outils pour permettre la connexion de tous les composants (bibliothèques et/ou utilisateurs) entre eux. Il faut également lui fournir une architecture logicielle adéquate

L'architecture logicielle est un ensemble de standards dans lesquels un framework de services et un ensemble de composants de base sont fournis pour permettre la construction de telles applications. Ce framework est enrichi par les composants d'une bibliothèque afin de fournir des composants de contrôle nécessaires pour exécuter l'application.

L'utilisateur doit, dans tous les cas, assembler son application suivant le genre d'algorithme (parallèle, séquentiel ...). S'il s'agit d'un algorithme parallèle, les composants de contrôles viennent faciliter cet assemblage afin d'assurer l'exécution parallèle des composants de l'application.

4.2 Les composants de parallélisation

Pour aider l'utilisateur à assembler une application parallèle à partir des composants logiciels, il faut lui fournir des composants de contrôle de parallélisme nécessaires. Dans le cadre des applications parallèles, il est important de fournir un composant de distribution. Celui-ci distribue des données différentes sur plusieurs instances d'une même tâche ou distribue une même donnée sur plusieurs tâches différentes. Ce composant est appelé *Distributeur*. Pour collecter les données fournies par plusieurs tâches de calcul, nous avons besoin d'un composant de la même nature que le distributeur mais faisant l'opération inverse, ce composant est le *Collecteur*.

L'utilisation de composants de ce genre facilite la construction d'une application parallèle dans laquelle le programmeur instancie les composants de contrôle nécessaires pour la distribution ou la collecte des données.

La conception du composant de parallélisation se fait par le concept de *port multiple*. Le port multiple est un port qui peut se connecter avec plusieurs ports afin de déclencher simultanément plusieurs appels sur plusieurs composants. Le composant parallèle est illustré par la figure 4.1

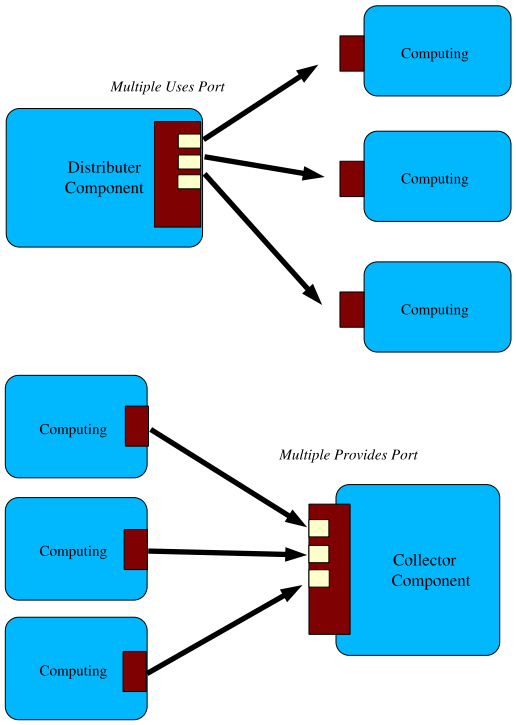

FIG. 4.1 – Le composant de parallélisation - Distributeur et Collecteur

4.2.1 Le port multiple

Le port multiple est un port CCA qui peut être connecté a plusieurs ports de même type. Lors de l'appel du service entre un composant qui en possède un et les composants connectés à ce port, l'appel se fait d'un façon simultanée. La parallélisation de l'appel est géré par le mécanisme de threads en Java.

Ce port est d'abord crée de telle façon que l'utilisateur puisse connecter avec celui-ci le nombre de ports qu'il désire. La connexion est une connexion normale entre deux ports *uses-provides* mais pour multiplier le port, nous utilisons le mécanisme d'événements de connexion défini par l'architecture CCA. Lorsqu'un port de type multiple se connecte avec un autre port, un événement de connexion se déclenche pour informer le framework de cette connexion. Il est nécessaire de mettre le même composant de parallélisation à l'écoute de cet événement pour être informé de cette connexion. Dès que la connexion est faite, ce composant a effectivement eu l'information relative à la connexion et peut donc réagir. L'action du composant parallèle est de créer dynamiquement un nouveau port du même type et de le solliciter pour une autre éventuelle future connexion. L'utilisateur

connecte donc ce port avec un autre port appartenant à un autre composant et il obtient en retour un nouveau port libre qu'il connecte à un deuxième port et ainsi de suite.

Le scénario possible d'une connexion est le suivant :

- le composant parallèle est à l'écoute de l'événement de la connexion
- il implémente l'action qu'il doit fournir lors d'une connexion avec le port multiple
- l'action est de créer un port et de l'enregistrer avec l'ensemble des ports de ce composant sans les connecter.

La définition en CCA d'un port multiple et du composant de parallélisation qui l'emploie est la suivante :

```
// le port
public interface ParallelPort extends  Port {

    public Object [] doCompute (Object params []);
    public int getNumberConnected ();
}

//le composant
public class ParaComponent implements Component,
                ConnectionEventListener {
  ConnectionEventService eSvc;
  MultiplePort mpPort=new MultiplePort();

//implementation du port
  class MultiplePort implements ParallelPort{
    // implementations des methodes
  }

 // implementation de Component
 public void setServices (Services cc) {
 ...
  service.addProvidesPort(mpPort,
     service.createPortInfo("unPortMultiple",
               "intParllelPort", "") );
  PortInfo eSvcPI = cc.createPortInfo(
     "eSvc","ConnectionEventService","");
  services.registerUsesPort(eSvcPI);
  eSvc.addConnectionEventListener(
       ConnectionEvent.Connected, this);
 ...
 }

//implementation de ConnectionEventListener
 public void connectionActivity
                (ConnectionEvent evt){
```

```
 //ajout du port multiple et l'enregistrer
 //avec les services du framework

 }
...

}
```

4.2.2 Le distributeur

L'intérêt de la construction du composant distributeur est d'offrir un mécanisme de lancement des calculs parallèles. Les données sont fournies sur son port d'entrée et sont ensuite distribué sur plusieurs composants connectés sur son port de sortie en activant chaque composant de calcul pour une donnée. Le composant distributeur est donc un composant qui a les ports suivants :

- Un port *provides* d'entrée dans lequel on entre les données sur lesquelles le programmeur souhaiterait faire des traitements parallèles
- Un port *uses multiple* de sortie qui est potentiellement connecté à plusieurs composants
- Un port de configuration
- Un port de service d'événements internes

Le port de configuration est un port *provides* et le port *uses* qui est connecté à ce port est le framework.

Le port de service d'événements est un port enregistré comme *uses* par le composant et dont l'implémentation est au niveau du framework.

Le port d'entrée est un port *provides*, ce qui signifie qu'il implémente un service fourni par le composant distributeur. Le service fourni consiste à dispatcher les données d'entrée et à lancer l'appel des port *uses* sur les composants connectés au distributeur ; la démarche de dispatcher est la suivante :

- récupérer le nombre de composants connectés au distributeur
- récupérer les données passées comme argument dans la méthode du dispatcher
- activer un thread d'appel pour chacun des port uses connectés sur une donnée.

L'activation parallèle des composants de calcul se réalise grâce à la création d'un thread pour chaque connexion. Le retour de résultats se fait par le retour de l'appel de port de chaque composant de calcul sur son thread dédié. C'est le thread dédié qui va remplir un tableau avec les résultats obtenus. Une synchronisation est réalisée pour attendre que la totalité des threads activés soit terminés. Une fois que tous les résultats des calculs sont obtenus, le distributeur retourne un tableau qui contient les résultats des calculs réalisés par les différents composants de calcul.

La méthode utilisée par le dispatcher dépend de l'utilisation de ce port par le composant appelant. Si le programmeur utilise un composant qui a un *port multiple*, c'est l'utilisation dans son composant qui décide quelle méthode est à appeler.

4.2.3 Le composant adaptateur

Avec le composant distributeur qui vient d'être présenté, nous avons imaginé un autre mode de fonctionnement qui se rapproche d'un flux de données. En effet lors d'un appel au composant distributeur pour le calcul, l'appelant attend le retour des données, et chaque composant traversé par la donnée est en attente du retour (comme lors de l'utilisation classique des objets). De plus si la composition est dans un contexte distribué, les ports d'entrées servant aussi de port de sortie peuvent consister un "goulot d'étranglement". Nous avons imaginé que les données pouvaient transiter par les différents composants et avoir un chemin de retour des données différent. Les composants prennent donc la donnée à l'entrée, libèrent l'appelant du port, réalisent une opération sur les données si besoin, et transmettre la donnée sur le ou les ports de sortie. Pour avoir ce type de fonctionnement nous avons développé un composant qui joue le rôle de l'adaptateur pour ne pas bloquer le retour de résultat. Ce composant adaptateur est représente par la figure 4.2

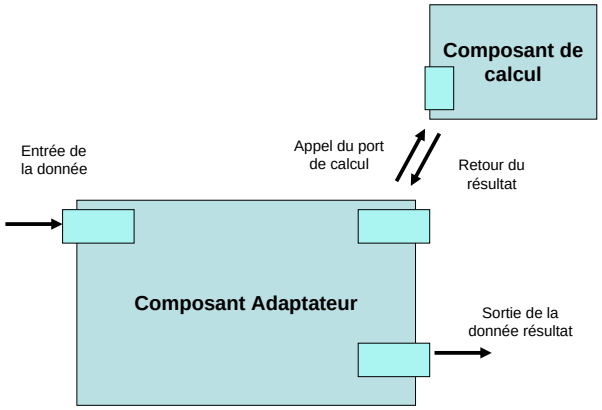

FIG. 4.2 – Le composant Adaptateur

Ce composant récupère une donnée sur un port d'entrée, réalise l'appel au composant de calcul qui lui est branché. Ensuite pour récupérer les calculs réalisé en parallèle c'est le composant collecteur qui s'en charge. Avec le retour des données différencié de l'activation des calculs, on peut maintenant envisager profiter d'un résultat dés qu'il est obtenu.

4.2.4 Le collecteur

Pour mieux profiter du parallélisme, il est intéressant de pouvoir exploiter les résultats des traitements au fur et à mesure de leur disponibilité. Le rôle du collecteur est complémentaire de celui du distributeur comme il est illustré par la figure 4.3. Le composant collecteur sert à recueillir les résultats des activations parallèles sur les données. L'activation parallèle entraîne le retour de plusieurs résultat, et ceux-ci peuvent être dans un ordre quelconque. Pour tirer un gain plus important du parallélisme il faut des primitive des récupérations du premier résultat calculé pour traiter les résultats au fur et à mesure de

leur disponibilité. Le collecteur ici joue le rôle de gestionnaire des résultats. Le composant collecteur est illustré par la figure 4.3

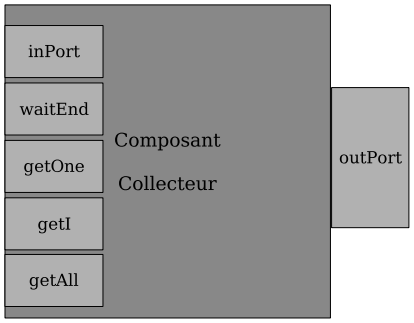

FIG. 4.3 – Le composant Collecteur

Le composant collecteur dispose des ports suivants :
- **inPort** est un port d'entrée *provides multiple*
- **waitEnd** est un port *provides* : l'appel à ce port lance la réception de toutes les données sur les ports d'entrée.
- **getOne** est un port *provides* et l'appel à ce port est bloqué jusqu'à ce qu'un nouveau résultat soit reçu sur le port d'entrée, dans ce cas c'est ce résultat qui est renvoyé.
- **getI** est un port *provides* qui permet l'obtention d'un résultat d'une entrée particulière
- **getALL** est un port *provides* qui permet d'attendre le retour de tous les résultats sous forme d'un tableau de résultats.
- **outPort** est un port *uses* de sortie

Le port d'entrée est conçu de la même façon que le port de sortie du distributeur. Il est de type *multiple*, créé à la connexion comme une réaction à un événement de connexion.

4.2.5 Exemple de compositions des composants de parallélisation

Nous avons mis en œuvre une composition pour tester les composants de parallélisation notamment le distributeur, le collecteur et l'adaptateur. Dans cette composition nous avons une distribution de données par le distributeur et collecte asynchrone de données en utilisant le collecteur et l'adaptateur. les différents composants qui inter-agissent dans l'application sont :

- le composant **Main** qui est le composant débutant de l'application. L'application commence à s'exécuter par le déclenchement de ce composant à travers son port *go* . Il dispose des ports suivants :
  1. *start* est un port de type *go*. Ce port est un port *provides* qui implémente l'activation du composant main. Le framework utilise le port "start" pour commencer l'application.

2. *returnResultat* est un port *provides* de type *intFinalPort* dont le rôle est de récupérer les résultats fournis par le composant collecteur

3. *PortSource* est un port *uses* qui est enregistré pour utiliser le port *dispatcher* du composant distributeur. Ce port fourni des données à paralléliser au distributeur

- le composant **componentPara** est un composant distributeur. Ce composant implémente la fonctionnalité d'un distributeur et il a les ports suivants :

  1. *dispatcher* est un port *provides* qui fournit le service de ce composant qui consiste à paralléliser les données. Le composant `Main` vu ci-dessus utilise ce port en passant des données à paralléliser. L'implémentations de ce port est d'utiliser un calcul pour chaque donnée passée. L'utilisation de calcul est déclenché par un appel sur le port *uses multiple* dont le nom est *aUsesPort*

  2. *aUsesPort* est le port *uses* multiple qui va être connecté à des composants pour faire le calcul. Ce port va être multiplié au fur et à mesure des connections de nouveaux composants de calcul.

  Le composant de parallélisation est connecté ici au travers le port *aUsesPort* multiple aux composants de calcul mais par l'intermédiaire d'un composant adaptateur.
- le composant **adaptateur** est utilisé pour le retour asynchrone de résultats. Pour cela il a trois ports essentiels :

  1. *input* dirige l'appel au composant de calcul pour ne pas bloquer le composant parallèle pour l'attente du résultat.

  2. *adapter* appelle le composant de calcul et récupère le résultat.

  3. *output* récupère le résultat fourni par le port *adapter*.

  Le résultat fourni par le composant adapter est utilisé par un collecteur.
- le composant **Join** est le collecteur qui sert a récupérer les données fournis par les composants de calcul par l'intermédiaire du composant adaptateur.
  Ce composant dispose des ports suivants :

  1. *inPort* qui est un port multiple qui sera multiplié au fur et à mesure des connexions avec les composant de calculs. il y a toujours un port libre pour la connexion de ce port avec un composant de calcul.

  2. *outPort* est un port de sortie qui sert a envoyer les données regroupé dans un tableau vers un composant qui en a besoin.

Dans notre exemple, le distributeur est connecté à deux composants de calcul différents. Un composant (`squareComponent`) calcule le carré d'un entier et l'autre (`doubleComponent`) multiplie un entier par 2. Le distributeur donc distribue chaque 2 données d'un tableau vers ces deux composants. Nous demandons que le résultat de calcul soit retourné au composant principale *Main* qui a débuté l'exécution de l'application.

L'exemple est illustré par la figure 4.4

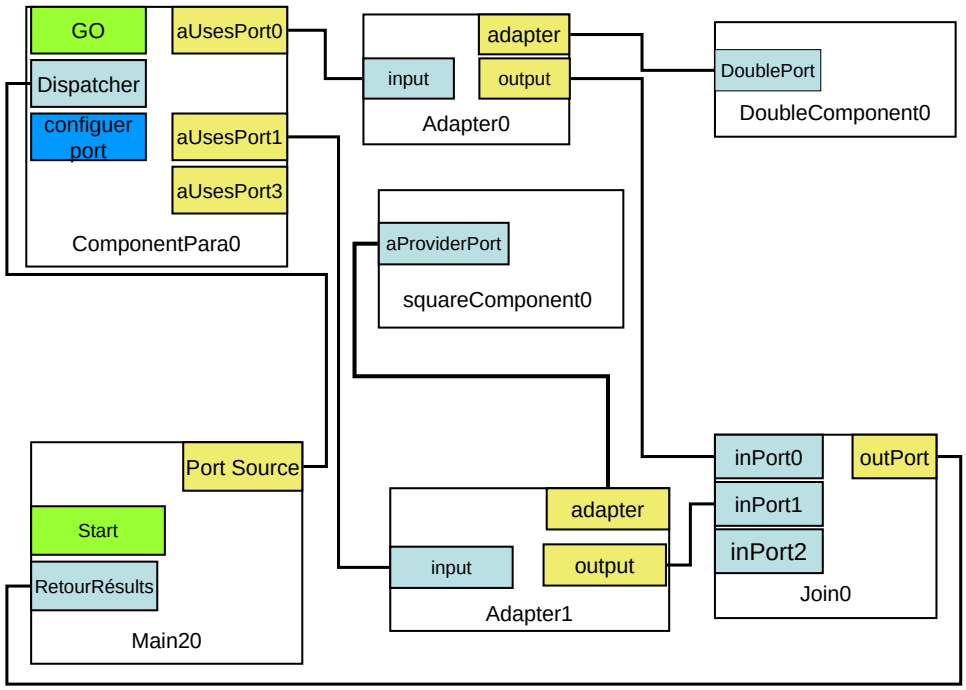

FIG. 4.4 – Exemple de composition de composants de parallélisation

4.3 Pipeline

Le pipeline est à l'origine, une technique qui a largement été utilisée dans le domaine de l'architecture des ordinateurs pour accélérer l'exécution séquentielle de certains codes. Cette technique est applicable à un vaste ensemble de problèmes de nature séquentielle. Alors, cette technique permet à plusieurs instructions de se chevaucher pendant l'exécution. Aujourd'hui, cette technique est capitale pour rendre les processeurs rapides. Un pipeline ressemble à une chaîne de montage : à chaque étape du pipeline comme de la chaîne de montage s'effectue une partie de la tâche globale. Les ouvriers sur une chaîne de montage de voitures effectuent de petites tâches, telles que l'installation des housses de sièges. La puissance d'une chaîne de montage réside dans le fait qu'un grand nombre d'ouvriers effectuent de petites tâches pour finalement produire collectivement un grand nombres de voitures par jour.

4.3.1 Fonctionnement du pipeline logiciel

Le pipeline logiciel est une conception du pipeline utilisé dans la programmation. La programmation parallèle s'inspire de l'architecture de l'ordinateur pour construire des applications parallèles. Le pipeline est un des composants matériels qui a été modélisé dans la programmation afin d'offrir une fonctionalité parallèle. Dans la technique de pipeline, la tâche principale est subdivisée en une série de tâches qui doivent s'exécuter les unes après les autres. Chaque tâche débute lorsque la précédente est terminée. Par rapport

à un fonctionnement séquentiel classique l'avantage du mode pipeline est que les tâches peuvent être exécutées simultanément. Chaque processus du pipeline est exécuté dans un étage de pipeline. Chaque étage contribue dans tout le problème et passe des informations nécessaire pour les étages suivants. Ce parallélisme peut être vu comme une forme de *décomposition fonctionnelle*.

La tâche est divisée en fonctions séparées qui doivent être exécutées successivement. Dans un programme séquentiel, le pipeline est formé par la succession de plusieurs processus dont chaque sortie est l'entrée du suivant.

A partir d'un problème qui est divisé en une série de tâches séquentielles, l'approche pipeline offre une accélération des temps de calcul dans trois types de cas :

1. si plusieurs instances du problème doivent être exécutées,
2. si une série de données doit être traitée

Le premier cas est largement utilisé dans la conception de l'architecture interne des ordinateurs. Il est également utilisé dans les exercices de simulations lorsque les simulations doivent s'exécuter avec des paramètres différents.

Le second cas considère une série de données qui doit être traitée dans l'ordre. Ceci apparaît dans les calculs arithmétique comme la multiplication des éléments d'un tableau où les éléments individuels entrent dans le pipeline en suite séquentielle de nombres . La figure 4.5 illustre un exemple mettant en œuvre 10 processus d'un pipeline et comportant 10 éléments de données $d_0..d_9$ qui entre dans le pipeline. Si nous considérons que nous avons p processus et n données et si nous considérons des temps de calcul de chaque étage du pipeline égaux, le temps d'exécution total peut se calculer comme $(p\text{-}1) + n$ cycles de pipeline .

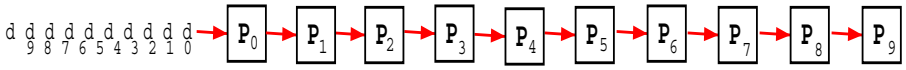

FIG. 4.5 – Pipeline traitant 10 éléménts de données

4.3.2 Pipeline : un composant logiciel

La conception d'un pipeline selon un modèle de composant logiciel nécessite en premier lieu, de déterminer les information suivantes :

- quelles sont les parties générique du pipeline.
- quelles sont les information d'entrée et de sortie et les paramètres de chaque étage
- quelles sont les parties communes entre les étages du pipeline

Il y a deux façon de concevoir un pipeline qui utilise des composants : la première est de mettre le pipeline dans un seul composant et la deuxième et de faire des composants par étage du pipeline.

Un composant intégrale de pipeline Dans ce cas, nous utilisons un seul composant de pipeline avec toutes ces fonctionnalités. La figure 4.6 montre les parties du composant :

- un port d'entrée P_e à gauche du pipeline où les données sont introduites
- un port de sortie P_s situé sur le côté droit du pipeline où des données sont délivrées au composant de droite suivant
- des port de calcul $Pc_1..Pc_n$ qui sont connectés aux composants de calcul dont chacun effectue une fonction sur la donnée passé et il délivre le résultat au port de calcul suivant
- une partie interne du composant qui est responsable de la synchronisation des données.

Les ports servent ici comme des points de contact pour le pipeline avec l'ensemble des composants de l'application. Les ports d'entrée et de sortie sont des ports de transmission de données. Les ports du calculs sont des ports d'invocation des méthodes de calcul situées dans d'autres composants.

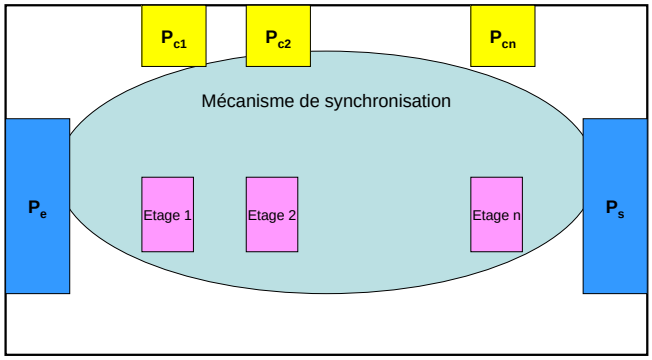

FIG. 4.6 – Schéma général du pipeline en un seul composants

Un composant par étage du pipeline Dans ce cas nous utilisons un composant pour chaque étage du pipeline. Avec cette conception, chaque composant étage est constitué, comme la figure 4.7 illustre, des parties suivantes : Chaque étage est un composant dans lequel nous distinguons trois ports :

- port d'entrée P_e situé sur le côté gauche du composant qui peut être connecté au port de sortie de l'étage précédent.
- port de sortie P_s situé sur le côté droite du composant qui peut être connecté au port d'entrée de l'étage suivant
- port d'invocation de calcul P_c qui passe les données fourni par le port d'entrée comme arguments et fourni un résultat au port de sortie.

Nous introduisons la réalisation de ces deux types de composants CCADAJ dans la section 4.3.4 Le mécanisme de synchronisation dans les deux cas repose sur les notification

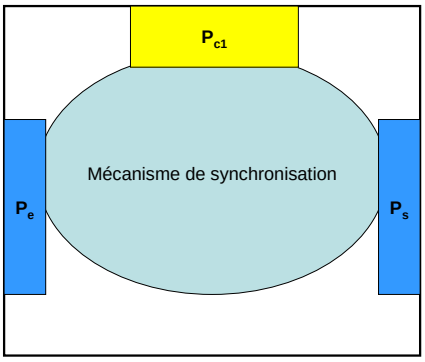

FIG. 4.7 – Schéma général d'un étage du pipeline

passé d'un étage à l'autres sur la disponibilité de données dans l'entrée. Nous pouvons aussi utiliser le super-composant pour pouvoir créer un composant pipeline à partir des composants étages.

4.3.3 Le composant pipeline et les flux de données

Dans une conception d'un pipeline logiciel, ce qui nous intéresse est le traitement de données dans ce pipeline avec la synchronisation entre tous les étages du pipeline.

Le traitement sous forme de pipeline des grandes quantités de données est présente par un flux de données qui entre par le port d'entrée du premier étage du pipeline. Etant donné un pipeline qui est construit de N étage et un flux de données de M données. La données d_N entre l'étage $e_1$quand l'étage e_N traite le données d_1

Les activations des étages du pipeline se succèdent dès l'arrivé de la première donnée. Nous avons deux modes possible d'activation illustrés par les figures 4.9 et 4.10 : un mode repose sur la disponibilité de données dans le flux, nous avons alors un pipeline piloté par les données ou *Data Driven*. Dans ce mode nous pouvons utiliser deux cas : avec ou sans notification d'événements. Les événements servent à notifier les étages du pipeline sur la disponibilité ou la demande des données. Dans le cas ou les événements ne sont pas implémentés, les demandes ou les disponibilités des données sont effectuées directement par les méthodes de ports. L'autre mode est le cas de la nécessité de la donnée par le pipeline ; nous avons dans ce cas un pipeline piloté par la demande ou *Demand Driven*. la figure 4.8 représente les différents cas possible pour mode d'échange de données entre les composants.

Le mode *Data Driven*

Dans ce mode de fonctionnement, les composants sont connectés mais les processus de traitement de données dans chaque étage du pipeline ne sont pas encore activés. Etant

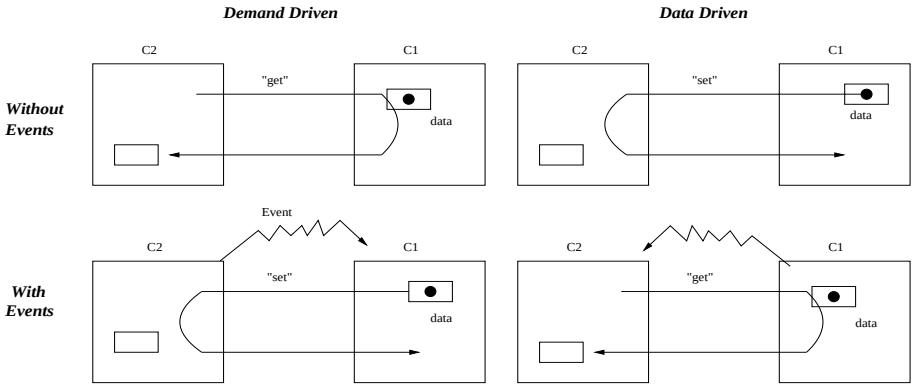

FIG. 4.8 – Les modes d'échange de données

données une source de données qui est liée à un pipeline de N étages et un consommateur de données. Dans un premier temps la source de données exprime la disponibilité de ces données au premier étage du pipeline a travers la connexion avec cet étage. L'expression de la disponibilité est fait par une notification sur le premier étage.

Chaque étage du pipeline doit gérer la synchronisation entre la réception de données et l'invocation du calcul sur le composant de calcul et l'envoie de données à l'étage suivant ou au consommateur de données. Le fonctionnement se déroule comme suit :

1. Le composant de source de données teste son buffer de sortie si il n'est pas vide, alors il informe l'étage de la disponibilité des données dans son buffer de sortie.
2. Le composant étage reçoit la notification et il teste donc son buffer pour voir si il est vide. S'il est vide alors il récupère la données du composant de source par un appel sur la connexion *uses-provides*, il lance le calcul sur cette donnée et il met le résultat dans son buffer. Sinon il notifie le composant consommateur de la disponibilité de donnée dans son buffer et il attend qu'il la récupère.
3. Quand l'étage récupère la donnée, la source maintenant met la données suivante dans son buffer de sortie
4. si le consommateur veut récupérer une donnée il le fait a travers la connexion *uses-provides* et dans ce cas il déclenche un nouveau cycle en vidant le buffer de l'étage.

Un schéma général du pipeline en mode *Data Driven* est illustré par la figure 4.9.

Le mode *Demand Driven*

Ce mode de fonctionnement est l'inverse du mode précédent. Le traitement est déclenche selon les besoins de chaque composant. Le fonctionnement de ce mode est le suivant :

1. le composant consommateur teste son buffer d'entrée si il est vide et il exprime son besoin des données qui sont à la sortie du pipeline. Alors il envoie une notification au dernier étage du pipeline pour que celui-ci lui délivre la donnée.
2. le composant du p^{ieme} étage du pipeline reçoit la notification soit de consommateur soit de l'étage p+1 et teste son buffer s'il n'est pas vide, il déclenche la méthode de

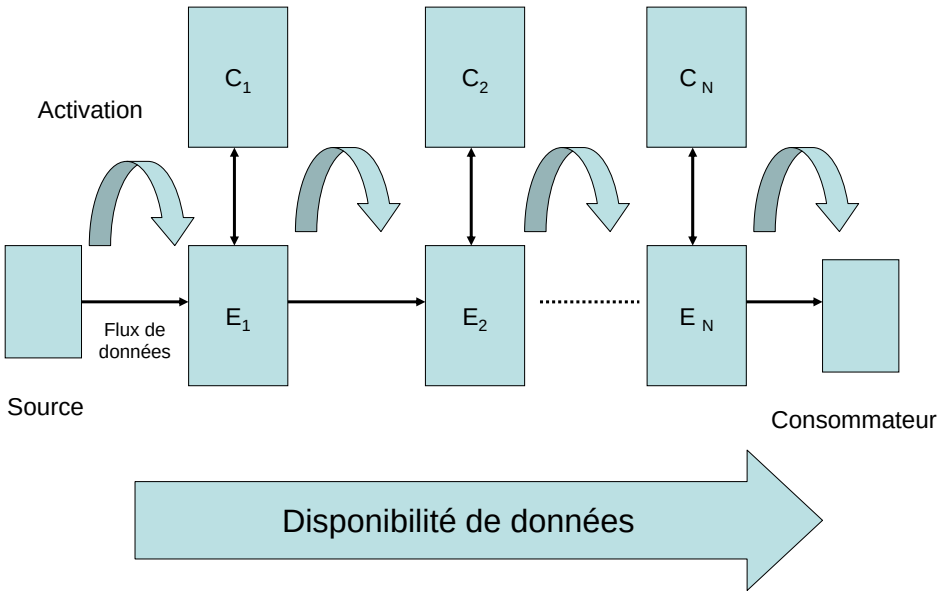

FIG. 4.9 – Pipeline en mode *Data Driven*

calcul sur la donnée qui est dans le buffer et livre le résultat au consommateur ou à l'étage suivant. Sinon, il notifie le composant source ou l'étage précédent qu'il a besoin d'une donnée à traiter.

3. Quand le composant source reçoit la notification de demande de données, il teste son buffer si il n'est pas vide il envoie la donnée au composant qui la demande

Un schéma général du pipeline en mode *Demand Driven* est illustré par la figure 4.10.

4.3.4 Exemple d'un composant pipeline en CCA

Nous avons réaliser une composition des composants en fonctionnement pipeline.
Un pipeline peut être présenté sous deux formes différentes :

- un seul composant dans lequel nous avons plusieurs ports d'invocation de calcul, un port d'entré et un port de sortie
- plusieurs composants qui représentent les différents étages de pipeline

Dans le premier cas nous avons un composants pipeline qui est presenté par la classe java suivant :

```
public class Pipeline implements Component {
  GetObjectPort inputPort;
  PortInfo inInfo;
  GetObjectPort outputPort;
  PortInfo outInfo;
```

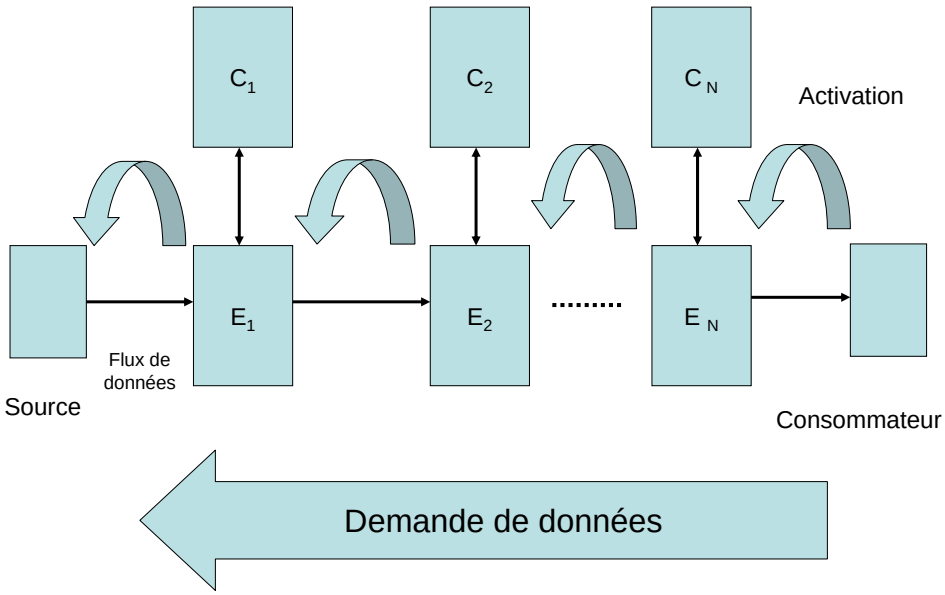

FIG. 4.10 – Pipeline en mode *Demand Driven*

```
  InvokerPort invokPort;
  PortInfo invocInfo;
  ...
  public void setServices(Services svc) {
    inInfo=svc.createPortInfo(‘‘Entree de Données’’,
                          ’’GetObjectPort’’,null);
    outInfo=svc.createPortInfo(‘‘Sortie de Données’’,
                          ’’GetObjectPort’’,null);
    invocInfo=svc.createPortInfo(‘‘Sortie de Données’’,
                          ’’GenericInvocPort’’,null);

    svc.registerUsesPort(inInfo);
    svc.addProvidesPort(outputPort, outInfo);
    svc.registerUsesPort(invocInfo);
    ...
  }

}
```

La figure 4.11 illustre un pipeline de deux étages avec une source de données et un consommateur de données.

Le port d'invocation générique est un port qui peut être dupliqué selon le nombre

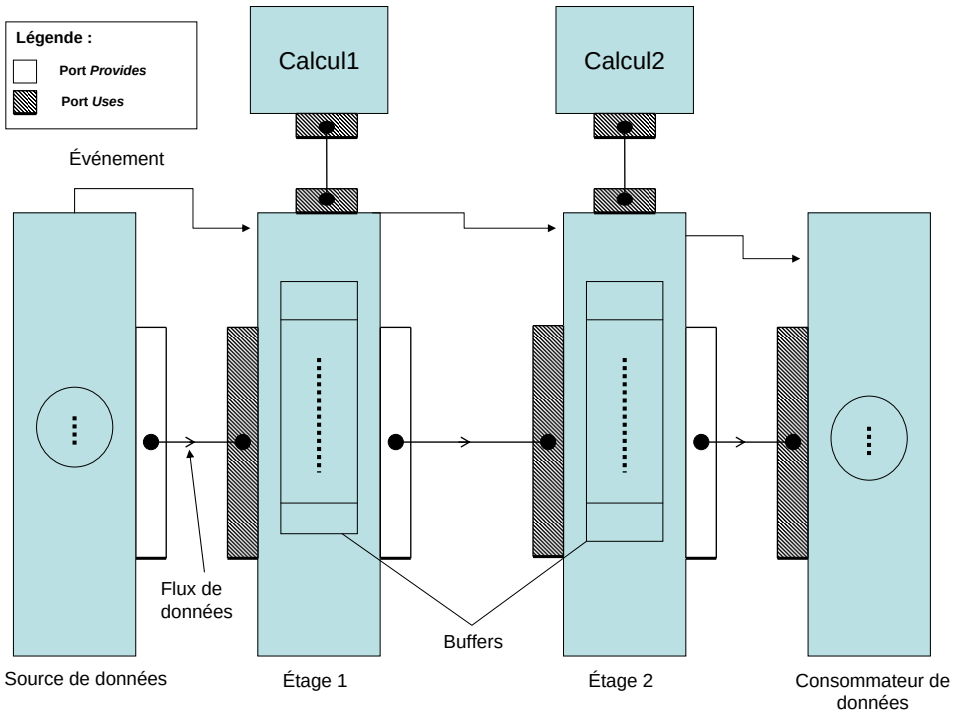

FIG. 4.11 – Pipeline de 2 étages

d'étage du pipeline. La définition de ce port est représenté par l'interface :

```
public interface GeneralUsesPort extends Port{
  public Objetc generalInvoke(Objet[] margs );
}
```

La génération dynamique de ce port est géré par les événements des propriétés de composants et les événements de connexion. La définition de port d'invocation générique se trouve dans la section 3.4.3. les ports d'entrées et de sortie de données sont de type *GetObjectPort* qui est définit comme suit :

```
public interface GetObjectPort extends Port{
  public Objetc getObject();
}
```

Les composants sources de données et de consommateurs ont les même type de port pour les données. Les composant de calculs ont un type de port différent. Dans notre exemple nous effectuons un calcul simple qui fait : $A.X^2 + B$. Le premier composant de calcul il effectue la racine carré de X et le deuxième récupère ce résultat, la multiplie par une constante A et y ajoute une constante B. Le composant source fournit un flux de données qui sont passées en tant que valeurs de X. Le consommateur obtient un tableau de données de résultats.

4.4 Dynamicité des applications en CCADAJ

Le concept de dynamicité de l'application est un des points essentiels de la construction d'une application distribuées. Cette dynamicité permet au développeur de tester les différentes configurations de l'application, d'étudier son évolution et de pouvoir ainsi obtenir la meilleur configuration en termes de performance ou d'occupation mémoire. D'un autre coté, les applications de calcul scientifique ont une durées d'exécution longue. Pouvoir améliorer les composants logiciels de l'application distribuée sans être obligé de l'arrêter est l'un des critères les plus importants dans la construction d'un environnement à base de composants.

Cette dynamicité peut être vue selon les aspects suivant :

- le développement incrémental : l'aspect incrémental du développement de l'application distribué permet de construire et de déployer cette application en plusieurs étapes. Un composant qui représente un processus peut être ajouté au cours de l'exécution. Une connexion entre deux composant peut être créée au cours de l'exécution et un nouveau composant peut être créé (génération automatique du code) et ajouté à l'application
- le remplacement de composant : permet d'améliorer la performance d'une application distribuée comme les application de calcul scientifique qui ont un temps d'exécution important. Dans ces applications, nous sommes parfois amené à remplacer un composant de l'application par un autre pour échanger la fonction de traitement ou pour utiliser un composant plus spécialisé.
- la reconfiguration d'un composant : qui traite les interaction entre le composant et son environnement extérieur et de son exécution.

L'environnement à base de composants CCADAJ repose sur les spécifications du modèle CCA qui présent un environnement d'évènements sur les connexion et la création de composant. Le développeur d'un composant pourrait programmer son composant de façon à ce qu'il puissent se connecter à d'autres composants présents dans l'environnement en temps réel et selon ses besoin adaptatif. Ce cas est utilisé dans le composant parallèle présenté dans le chapitre 4. La création dynamique des ports permet d'utiliser directement les composants en les connectant à d'autres composants pour faire le calcul nécessaire. Ce système d'événement est basique dans une architecture CCA. Le framework ainsi que les composants peuvent l'utiliser afin de présenter une dynamicité dans la création et l'utilisation des composants. Les autres modèles de composant tel que CORBA et EJB ne fournissent pas une telle dynamicité. Ils doivent être étendus et adaptés spécifiquement pour la fournir.

Le programmeur peut aussi dans son application et à travers le framework de composition ajouter le composant voulu pour qu'il participe à l'application et le connecter ou remplacer par un autre composant. Le changement de la configuration de composant se fait à l'aide de l'utilisation du port de configuration spécialement conçu pour modifier la configuration du composant.

4.5 Conclusion

Dans ce chapitre nous avons présenté des structures spéciales que nous avons appelées des composants de contrôle. Le but de ces composants est d'offrir au constructeur de l'application un moyens simple d'intégrer certaines fonctionnalités de contrôle de parallélisme. Le constructeur de l'application dispose des structures de types scatter/gather ou pipeline pour contrôler son applications et là manière comment le calcul et données sont paralléliser. Les composants de parallélisation sont définit en deux types : distributeur et collecteur. Des ports spéciaux sont définit pour permettre la distribution ou la collecte du calcul. Nous avons nommé ces ports les *ports multiples*. La caractéristique principale de ce type de port est que le port est dupliqué au moment ou ils connecté à un port d'autre composant. Cette duplication profite de la dynamicité apportée par le modèle CCA au travers du mécanisme d'événements de connexion. Le distributeur est un composant qui implémente un port multiple et les ports dupliqués sont gérés dans le distributeur. Le distributeur prend un tableau de données à son entrée pour le dispatcher à la sortie vers plusieurs composants de calcul. Le retour des méthodes des ports connectés est synchronisé par la méthode du port multiple.

Pour pouvoir traiter un retour asynchrone de calcul, nous avons introduit le composants adaptateur qui est sert comme intermédiaire entre le composant distributeur et un composant de calcul. Le composant collecteur utilise le même principe que le composant distributeur. Le collecteur utilise un port multiple pour pouvoir être connecté à plusieurs composants de calcul. Il regrouper les différents données à son entrée et les met dans un tableau pour les offrir à un composant utilisateur. Le collecteur offre une fonctionnalité complémentaire à celle offerte par le distributeur.

Un autre type de composant de contrôle que nous avons créé : le pipeline. Il fournit une fonctionnalité de pipeline sous forme de composant pouvant être connecté à plusieurs composant de calculs pour effectuer des traitements en mode pipeline. IL dispose de plusieurs étages, d'un port d'entrée et d'un port de sortie et d'un mécanisme de synchronisation du flux de données et de résultats entres les étages. Nous avons discuté les différentes possibilités de mode d'opération pour un pipeline. Des modes comme disponibilité de données ou demande de données ont été présentés.

Finalement, ces composants de contrôle servent comme structure de base à offrir au constructeur de l'application à coté de ses composants pour permettre plus de coordination dans son application sans être obligé à réécrire des composants pour le faire.

Chapitre 5

Évaluations et réutilisation des composants CCADAJ

Dans ce chapitre, nous abordons les questions de performances du framework CCADAJ. Dans un premier temps, nous introduisons les apports du modèle CCA à cet environnement. Nous expliquons en détail les considérations de performance dans le modèle CCA. Ensuite, nous montrons comment réutiliser les composants de CCA dans cet environnement. Nous montrons aussi la construction des composants avec CCADAJ et la composition d'une application à partir de composants réutilisables. De plus, nous évaluerons l'utilisation des applications à base de composants logiciels. Celle-ci a un double effet, d'abord au niveau de la facilité de réutilisation des composants construits comme des boites noires avec des points de connexion et la réutilisation des composants de contrôle et d'aide au parallélisme, puis au niveau du surcoût de la performance lors de l'utilisation des composants.

Les évaluations de performance d'exécution sont réalisées par une série de mesures sur le temps d'exécution des applications. Nous effectuons des comparaisons entre des différentes implémentations : des implémentations réalisées en utilisant les composants CCADAJ et des implémentations réalisées en utilisant les objets Java. Nous examinons les différentes structures proposées précédemment dans cette thèse comme les composants de base CCADAJ, le super-composant et les composants distants. Nous comparons les mesures pour obtenir le surcoût dû à l'utilisation du framework CCADAJ. Finalement nous testons l'utilisation de framework en implémentant une application de TSP (Problème de Voyageur de Commerce) avec des composants CCADAJ.

5.1 Performances d'une architecture CCA

Dans la conception de l'architecture CCA, plusieurs facteurs sont pris en compte :

- Performance : le surcoût introduit par les couches supplémentaires liées aux composants et au framework doit être faible vis à vis des performances globales des composants et de l'application
- Portabilité : elle doit supporter divers langages et plate-formes pour bénéficier des codes existants pour le calcul scientifique

- Flexibilité : elle doit supporter une large gamme de paradigmes de programmation parallèle et en particulier le modèle SPMD, le modèle multithreading et les modèles distribués.
- Intégration : elle ne doit imposer qu'un minimum de conditions pour que les logiciels qui existent déjà pour qu'ils puissent fonctionner dans l'environnement spécifique des composants.

Pour satisfaire les conditions évoquées, les spécifications [AGG+99] développées dans le CCA Forum définissent :

- le comportement minimal pour un composant CCA,
- le comportement requis minimal pour un framework CCA,
- l'interface entre les composants et les frameworks,

Mais il faut qu'elles soient extensibles sans être spécifiques.

Le cœur de CCA comporte le concept de *ports*, à travers lesquels les composants interagissent entre eux et avec le framework.

5.1.1 Le framework

Un framework joue le rôle de médiateur lors de l'utilisation d'un composant par un autre au travers d'un patron de conception qui relie les ports *uses-provides* des composants CCA. Ce patron de conception est au centre de la flexibilité et des performances du modèle pour les appels inter-composants. Quand un composant utilisateur invoque la méthode `getPort()`, l'objet `Port` que retourne le framework, est un proxy pour les appels sur un composant distant dans un environnement distribué. C'est alors au framework de gérer les arguments et de faire l'invocation à distance. Les composants n'ont pas besoin de savoir qu'ils appartiennent à un environnement distribué. Bien évidemment, le calcul distribué n'est pas toujours considéré comme étant un environnement à haute performance, et l'utilisateur de CCA qui crée l'application est informé de la fréquence de l'utilisation et du volume de données transférées entre les ports quand il assemble son application.

Dans le cas où les deux composants sont locaux, l'appel `getPort()` peut retourner une référence de l'implémentation réelle. Ce fonctionnement est utilisé dans le prototype CCAFFEINE[AAW+02] qui met l'accent sur les applications parallèles à haute performance, écrites en C++. Les composants sous forme de librairie d'objets partagés sont chargés dans des espaces de nommage distincts dans un seul espace d'adressage (processus). L'utilisation de différentes espaces de nommage assure que les composants ne peuvent pas interférer entre eux et que le framework est la seule partie de l'environnement qui peut "voir" tous les composants. Puisque tous les composants sont dans un seul espace d'adressage, le framework peut facilement retourner une référence directe à l'implémentation du port de `getPort()`. Ceci est désigné sous le nom d'environnement à *connexion directe* qui permet à un composant d'appeler les méthodes sur d'autres composants avec un coût équivalent à un appel de fonction en C++.

Puisque les appels `getPort()` ne sont pas souvent utilisés (ils ne sont invoqués qu'une seule fois par port utilisé) leur coût n'est pas important. Le surcoût du framework CCA est totalement dû aux coûts des appels inter-composants comparables aux coûts des mêmes appels dans un langage de programmation natif. Puisque ce surcoût est de l'ordre du coût

de l'appel d'une fonction dans un langage natif, il ne joue pas un rôle significatif dans le cas des appels inter-composants

5.1.2 Interoperabilité de langage via Babel

L'utilisation de Babel [BEKE02] pour rendre les langages interopérables (c.f. 2.4.1), introduit un surcoût supplémentaire. Babel utilise une représentation d'objets internes basé sur le langage C qui fournit une interface entre les différents langages. En général, le surcoût peut être assimilé à deux appels de sous-routines. Le client appelle une routine souche qui traduit les arguments en C. La routine souche appelle la routine squelette qui traduit les arguments dans le langage d'implémentation et le squelette appelle l'implémentation. Dans certains cas, il existe un surcoût supplémentaire dû à la conversion entre les langages (spécialement pour les chaînes de caractères). Avec le code existant, le développeur peut avoir besoin d'insérer une couche supplémentaire pour adapter la représentation basée sur les objets et utilisée par Babel, à la spécificité du code existant. Comme on peut s'y attendre, quand les méthodes fournissent une grosse quantité de calcul, lors des appels de méthodes via Babel, le surcoût du système Babel n'est pas important[SKR01]. Il faut évidement être vigilant dans le cas d'utilisation de Babel avec des méthodes qui peuvent être appelées un grand nombre de fois et qui contiennent une petite quantité de calcul. Quand Babel est intégré dans un framework CCA le surcoût de l'environnement total CCA repose essentiellement sur le surcoût dû à Babel. L'appel de fonction virtuelle du framework est simplement effectué dans l'environnement Babel.

La figure 5.1 montre les mesures de performance de CCA en utilisant le langage Babel par rapport aux langages natifs et CORBA.

Function Group	F77 Time (ns)	C Rel. F77	C++ Rel. F77	Babel C to C++ Rel. F77	Ccaffeine Rel. F77	OmniORB Rel. F77
A	18	1.0	1.2	2.6	2.4	91.1
B	10-16	1.0-2.2	2.4-3.8	3.2-3.9	3.5	130.8
C	18	1.1	1.1-3.7	2.1-14.4	2.2-4.3	90.8
Overall Avarage	17	1.1	1.8	3.8	2.8	97.6

FIG. 5.1 – Temps réel pour les appels fonction F77 et coûts relatifs d'autres environnements [BEKE02]. *A : Mapping direct de Babel en langage d'implémentation (avec des arguments natifs comme* `Int` `Float`, `Double` *et* `Long`*) B : Des fonctions simples supplémentaires montrant le coût qui diffère significativement du groupe A dans des langages natifs (sans arguments). C : Les fonctions qui demandent certaines adaptations entre les langages en Babel et donc montrent plus de variations dans le coût (des arguments plus complexes comme* `Array`, `Complex`, `OrderedArray` *...etc).*

5.2 Performances des composants en CCADAJ

Dans cette section, nous parlons des performances des composants dans l'environnement CCADAJ et du surcoût du framework ainsi que du super composant. Dans l'environ-

nement CCADAJ, le modèle de composants choisi est le modèle CCA avec un framework qui s'exécute au dessus de l'environnement ADAJ (introduit dans le chapitre 3). Les considérations de performance des composants CCA, qui ont déjà été présentées dans la section précédente, concernent les composants et le framework CCA avec l'inter-opérabilité de langages via Babel. Ces considérations peuvent être pris en compte par notre framework, mais seulement en ce qui concerne les composants avec une seule implémentation en Java. Nous ne traitons pas du cas de l'interopérabilité avec d'autre langages. L'utilisation d'un environnement à base de composants mène à plusieurs évaluations de performance.

- Le surcoût apporté par l'utilisation des composants par rapport à la programmation par objets ainsi que le surcoût apporté au niveau de la composition hiérarchique et donc le surcoût de l'introduction du super composant.
- Le surcoût induit par la couche de parallèlisme/distribution pour le système de communication intégré en dessous du framework
- l'avantage de la facilité d'utilisation des composants et de l'environnement de composition ainsi que de la performance apportée par l'utilisation d'une bibliothèque de composants de contrôle qui aident à la construction d'une application parallèle et distribuée.

Dans l'environnement CCADAJ, la performance de l'interopérabilité de langage n'est pas pris en compte du fait de l'utilisation exclusive du langage Java. Ceci n'entraîne pas le surcoût supplémentaire considéré par CCA au niveau de l'interopérabilité de langages.

5.2.1 Mesures du surcoût de l'utilisation des composants CCA-DAJ

Comme nous l'avons déjà souligné dans la section 5.1, CCA a été conçu pour respecter les besoins en calcul scientifique. Les interactions entre les composants CCA utilisent les connexions *uses-provides*. Ce type de connexion permet au framework d'accéder à un service fourni par un composant et de le transmettre à un autre composant. Nous avons effectué des comparaisons entre deux applications : l'une mettant en oeuvre l'environnement CCADAJ et l'autre utilisant directement les objets Java. Dans la suite nous présentons une série de mesures sur le surcoût apporté par l'utilisation des composants et du framework CCADAJ. Ces mesures concernent dans un premier temps les composants locaux, puis les composants distants dans l'environnement d'exécution de CCADAJ. L'utilisation de cet environnement entraîne un surcoût apporté par le framework intermédiaire entre les composants et par la couche de communications.

Le surcoût d'utilisation des composants CCADAJ par rapport aux objets Java

Pour effectuer cette mesure, nous avons choisi de mettre simplement dans un composant une fonction qui fournit un certain service et nous avons placé dans un autre composant une fonction qui utilise ce service. Le code suivant montre l'écriture de ces deux classes en langage Java.

```
//main
public   class MainCalcul {
  public static void main(String [] args){
        Add add=new Add();
        System.out.println("-->"+add.add(5,2,6));
  }
}
//add
public class Add{
  public long add(int n1,int n2,int n3){
        return n1+n3+Factorial.fact(n2);
  }
}
//Factorial
public  class Factorial {
  public long fact(long n){
         return ;
  }
}
```

La classe `Add` utilise la classe `Factorial` pour faire un calcul sur la valeur passée par paramètre à la méthode `fact()`. La classe `Main` affiche le résultat du calcul fournit par la classe `Add`.

L'utilisation de ces classes en tant que composants dans CCADAJ nécessite certaines modifications pour que le framework puisse jouer son rôle d'intermédiaire entre le fournisseur de service et l'utilisateur de ce service. Dans ce cas nous avons trois composants : `Main, Add, Factorial`. `Factorial` est un composant fournisseur de service à travers la méthode `fact()` qui calcule le factoriel d'un nombre passé en argument. `Add` utilise le service proposé par `Factorial` et fournit un service d'addition comme présente dans le code. `Main` est un composant utilisateur d'un service d'addition proposé par Add et est un déclencheur de l'application. La figure 5.2 représente la connexion entres ces trois composants.

Pour transformer les classes que nous venons de décrire en composants CCA il faut leur ajouter la définition des ports et l'implémentation de l'interface `Component`. Le code suivant est ajouté aux différents composants à cet effet.

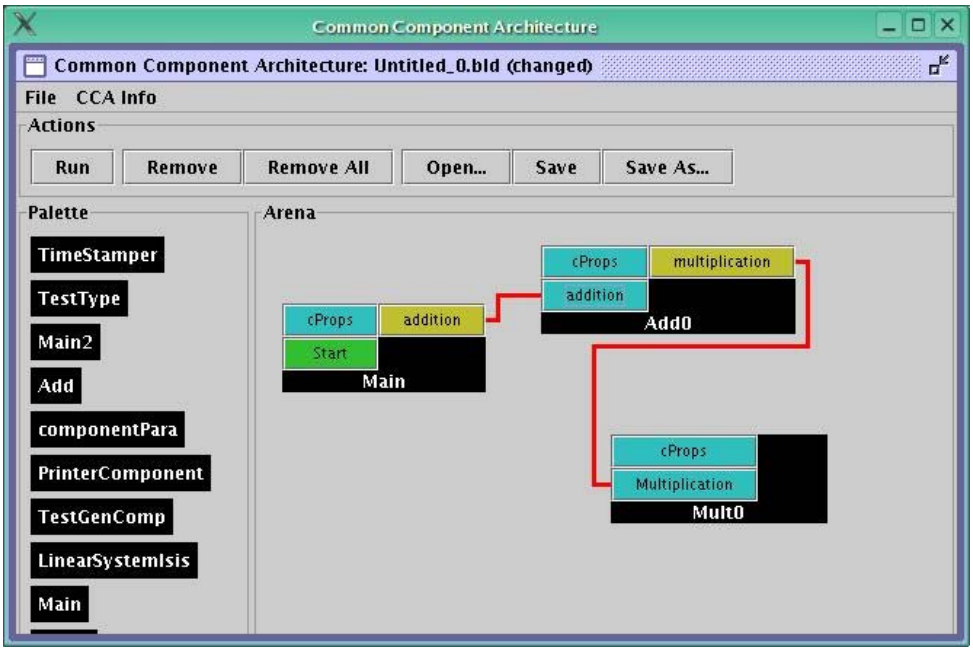

Figure 5.2 – Composants de test et composition

```
//Main
public   class MainCalcul  implements Component {
  PortAdd add;
  GoPortCalcul gpc=new GoPortCalcul();

  public class GoPortCalcul implements GoPort {
      public void go(){
         add=(PortAdd)services.getPort("addition");
         System.out.println("--------------->"+add.add(5,2,6));
      }
  }
 public void setServices(Services services){
   ...
   services.addProvidesPort (gpc,
      services.createPortInfo("Start", "GoPort",props));
            PortInfo pinfo= services.createPortInfo("addition","PortAdd",props);
            services.registerUsesPort(pinfo);
   ...
  }
}
```

Le composant `Add` est un composant qui fournit un service au travers le port `PortAdd` dont l'interface est implémentée à l'intérieur.

```
// Add
public   class Add  implements Component {
...
  public long add(int n1,int n2,int n3){
   ...
   PortFactorial fact = (PortFactorial) (services.getPort("Factorial"));
   return n1+n2+n3+fact.fact(20);
  }
...
 }
 public void setServices(Services services){
  ...

 }
}
```

Le composant `Factorial` avec l'implémentation de l'interface de port `PortFactorial` :

```
//Factorial
public   class Factorial  implements Component {
...
 public class thisPort implements PortFactorial {
   CalcFact cf=new CalcFact();
   public long fact(long n){
   ...
   }
 }
 public void setServices(Services services) {
  ...
  services.addProvidesPort (aProvidePort,
    services.createPortInfo("Factorial", "PortFactorial",""));
 }
}
```

Les définitions des interfaces des ports sont les suivantes :

```
//les ports
public interface PortAdd  extends Port
 {
     public long add(int n1,int n2,int n3);
 }
 public interface PortFactorial  extends Port
 {
     public long fact(long n);
 }
```

Pour encapsuler du code dans un composant en utilisant ses interfaces pour l'accès aux services, il existe des appels faits par le framework pour obtenir le service et le délivrer au composant utilisateur. Par exemple, l'appel dans le composant `Add` prend la forme

```
fact=(PortFactorial) (services.getPort("Factorial")) ;
 return n1+n2+n3+fact.fact(20) ;
```

Dans nos mesures nous avons considéré les trois composants avec un grand nombre d'appels de l'ordre d'un million sur la même fonction (`fact()`) et avons comparé les appels avec les appels dans l'implémentation en objets Java. Nous avons obtenu les résultats placés dans le tableau 5.1

	temps (ms)					moyenne
Java	362	358	360	359	357	359
Composant CCADAJ	408	346	364	369	375	372
surcoût						3.6%

TAB. 5.1 – Surcoût des composants CCADAJ : Une boucle d'appels de la méthode du port

Nous avons fait le même calcul, mais au lieu de faire un million d'appels sur le port `fact`, nous avons mis une boucle d'un million d'étapes pour faire le calcul à l'intérieur de la fonction `fact()`. Ce faisant, nous évitons les appels multiples du framework sur le port avec le même type de calcul que celui fait précédemment. Les résultats sont illustrés dans le tableau 5.2

	temps (ms)					moyenne
Java	309	311	308	309	308	309
Composant CCADAJ	330	325	311	319	311	319
surcoût						3.3%

TAB. 5.2 – Surcoût des composants CCADAJ : Un appel de la méthode du port

Malgré l'optimisation faite par le framework lors de l'obtention de la référence du port avant de l'utiliser, nous remarquons l'effet du choix d'introduire une grande quantité de calcul à l'intérieur des composants par rapport à celui de les mettre dans la connexion entre les composants. Cependant, la différence de surcoût est de 0,3% qui est le surcoût des communications entre les ports dans le même espace d'adressage. Le développeur du composant doit être conscient de ce fait pour optimiser l'écriture de composants et ne pas inclure dans une boucle les références des ports qui vont être utilisées dans le composant.

Dans le premier cas, le composant utilisateur appelle un million de fois le composant fournisseur pour faire un calcul. Dans le second cas, c'est le même calcul qui doit être obtenu mais la boucle d'un million de fois se trouve dans la méthode du port fournisseur. L'utilisateur a simplement besoin d'un seul appel pour faire le calcul nécessaire et donc un seul appel par le framework qui intervient pour délivrer le port fournisseur au composant utilisateur.

En prenant en compte ces résultats, l'appel du framework peut être moins important dans le cas où cet appel n'est pas répété. Il est de l'ordre d'un appel supplémentaire

d'une méthode en Java. Le développeur de composants doit être conscient du surcoût apporté par le framework et il doit donc faire attention aux données transférées entre les composants et mettre la boucle qui ne peut pas être traitée à l'intérieur de la fonction appelée.

Pour les expérimentations suivantes, nous avons choisi d'implémenter le premier choix pour réaliser nos mesures. C'est-à-dire, de mettre la boucle dans le composant appelant. Dans ce cas, nous pouvons mesurer l'impact des communications au travers des ports de connexion entre les composants.

Surcoût du super-composant

Nous avons introduit dans le chapitre 3, le modèle du super-composant utilisé pour la composition hiérarchique et exploité pour créer les containers qui permettent le déploiement des composants à distance. Le super composant utilise des proxys afin d'assurer la connexion entre les ports des composants internes et les ports des composants extérieurs au super composant. Dans la conception de tels composants, il existe les appels suivants :

- L'appel du framework sur le proxy du port de super composant pour récupérer le port d'un composant interne.
- L'appel du super-composant pour obtenir le port interne et le délivrer au framework.

Dans ce cas, le proxy et les services internes du super-composant jouent un rôle d'intermédiaire entre le framework et le composant interne. Le nombre d'appels successifs correspond au nombre d'imbrications de cette composition. Par exemple si le composant est dans un super composant qui lui même est encapsulé dans un autre super composant, nous avons le cas d'une imbrications à 2 niveaux. Il faut donc deux appels supplémentaires aux appels du framework, soit un appel par proxy. Pour les mesures effectuées sur les mêmes exemples que précédemment `(Main, Add, Mult)`, le composant `Mult` est encapsulé dans un super composant et est connecté au composant `Add` à travers le proxy dynamique lors de la composition. La figure 5.3 illustre la composition de super-composant et des composants de base.

Dans les mesures effectuées, nous avons obtenu un résultat comparable à celui obtenu avec l'hypothèse faite auparavant sur le surcoût dû à un appel supplémentaire du proxy utilisé dans le super-composant. Nous reprenons le même exemple que celui présenté dans le paragraphe précédent, sachant que la boucle de calcul est toujours à l'extérieur de composant `Factorial`. Ce qui veux dire que le composant `Factorial` sera appelé 100000 de fois par le composant add. Le composant `Factorial` se trouve encapsulé dans un super composant, le `SCFact` comme illustré dans la figure 5.3. Nous avons appelé un million de fois le composant `Factorial`. Le temps d'exécution et le surcoût sont présenté dans le tableau 5.3

Nous remarquons que les couches introduites dans la construction du super-composant affectent la performance de l'accès aux composants internes du super-composant. Mais cette construction reste un avantage dans la fabrication des composants à partir d'autres composants. Nous remarquons aussi que c'est pour problème de performance que le forum CCA n'a pas pris en compte la composition hiérarchique dans les spécifications de l'architecture CCA.

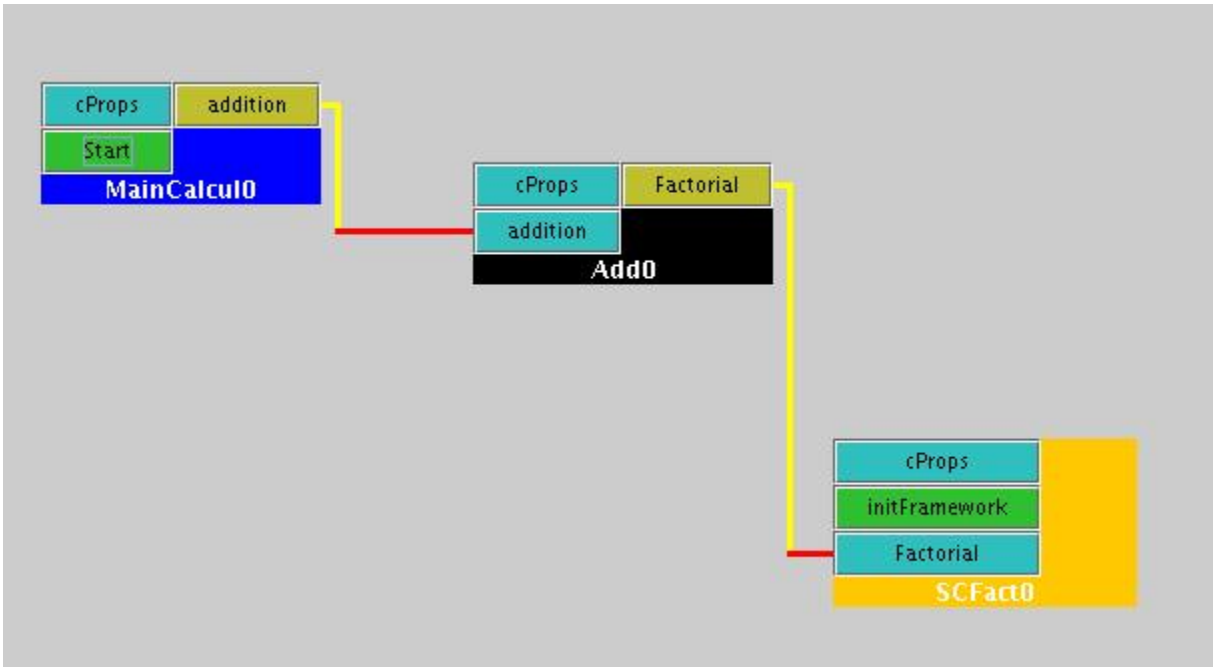

Figure 5.3 – composition de super composant

5.2.2 Surcoût du composant distant (remote)

Dans la conception du composant distant (remote) présenté dans le chapitre 3, nous considérons que ce composant est capable d'être déployé dans un environnement DG-ADAJ (c.f. 3.3). Par cette considération, tout composant qui veut être déployé à distance doit être encapsulé dans un composant distant. Cette encapsulation se fait de la même façon que pour un super composant, à la différence qu'ici le port est un port distant. La génération du port distant dû à l'utilisation du mot clé `remote` de *JavaParty* implique la création des classes nécessaires pour la transparence de la distribution dans l'environnement *JavaParty* et en particulier les classes souches et squelettes. Dans cet environnement, le surcoût de l'encapsulation d'un composant CCADAJ dans un composant distant est du même ordre que pour un super composant. Evidemment, il existe un surcoût dû à l'utilisation de l'environnement *JavaParty* et notamment le surcoût des proxys générés par le compilateur de *JavaParty*. L'appel d'un port distant est de l'ordre d'un appel fonction java et donc son coût n'est pas très important, s'il se passe dans la même JVM. Dans le cas où l'appel est passé par les proxys générés par JavaParty, il existe un surcoût de communication qui n'est pas dû à l'utilisation des composants mais au système de communication *JavaParty*.

Dans les mesures effectuées sur les mêmes cas que précédemment, le composant `Fact` est encapsulé dans un composant distant et peut être déployé dans l'environnement *JavaParty*. Nous avons fait une comparaison entre le temps d'exécution des classes JavaParty de l'exemple précédent et avec ces même classes en tant que composants CCADAJ avec le support de composants distants. Le tableau 5.4 représente les résultats obtenus.

	temps (ms)					moyenne
Composant CCADAJ	408	346	364	369	375	372
Super-Composant CCADAJ	399	403	393	386	398	395
surcoût						6.3%

TAB. 5.3 – Surcoût de Super-Composant CCADAJ

	temps (ms)					moyenne
Objets JavaParty	322633	254741	303589	150424	147919	235861
Composant distant CCADAJ	338764	321809	271535	201945	184362	263683
surcoût						11,79%

TAB. 5.4 – Surcoût de Composant distant CCADAJ

Ces résultat nous montre le surcoût apporté par l'intégration du CCADAJ dans le système de communication de JavaParty. Dans le développement basé sur les composants les données et l'accès aux services proposés par les composants se font au travers les ports. Pour cela, nous remarquons l'impact direct de la couche de communication lors de l'utilisation des composants. Il existe un surcoût qui pourrait augmenter dans la même application avec deux implémentations différentes de ses composants. Le choix de passer plus d'information au travers les ports ou d'utiliser plusieurs ports pour l'échange de données, peut affecter la performance de l'application.

5.3 Mesures de performance sur une application parallèle distribuée

Dans le cadre des mesures effectuées pour mesurer les performances du framework CCADAJ et le surcoût apporté par l'utilisation de composants, nous nous somme intéressés à l'évaluation d'une application parallèle distribué pour mesurer les performances globales de cette application par rapport à son implémentation sans l'utilisation de composants. Nous avons choisi une application de benchmark utilisé par la communauté *JavaParty*. Nous l'avons modifiée et implémenté sous forme de composants CCADAJ pour pouvoir effectuer des évaluations et des comparaisons entre les deux implémentations : implémentation orienté-objets avec JavaParty et une implémentation avec des composants

CCADAJ. L'application est une implémentation d'un algorithme *Branch and Bound* pour résoudre le problème de voyageur de commerce ou *Travelling Salesman Problem (TSP)* connu dans la littérature du parallélisme.

5.3.1 Une application TSP parallèle utilisant un algorithme Branch and Bound

Il s'agit du problème du voyageur de commerce. Il existe un nombre connu de villes avec les distances (coût) entre ces villes. Le voyageur de commerce doit effectuer une visite par ville et revenir au point de départ (un circuit) avec un minimum de coût. Le problème est modélisé par un graphe dont les nœuds sont les villes et les arcs sont les coûts de voyage pour circuler entre elles.

L'algorithme *Branch and Bound* construit un arbre de recherche dans lequel un nœud est le chemin qui correspond au début d'un circuit dans le graphe des villes. A chaque étape une ou plusieurs solutions partielles sont améliorées jusqu'à obtenir le meilleur circuit dans le graphe. L'élimination des nœuds dans l'arbre se fait quand les successeurs de ces nœuds ne contiennent pas de solution meilleure que celle qui est connue. Par ce moyen, le nombre de calcul est réduit.

La version parallèle de l'algorithme, implémentée pour nos mesures, utilise un contrôleur et des travailleurs distribuées. Chaque travailleur est sur une machine.

- Le contrôleur divise l'arbre en M sous arbre et envoie les racines de sous arbres à explorer aux M travailleurs.
- Chaque travailleur commence à explorer son sous-arbre et retourne ses solutions (c'est à dire, les solutions partielles qu'il a calculé) au contrôleur.
- Le contrôleur gère l'ensemble des solutions retournées par les travailleurs.
- Tous les travailleurs échangent l'information avec le contrôleur concernant la meilleur solution trouvée.
- Le contrôleur coordonne la distribution de travail aux travailleurs, obtient les résultats et fait une sélection des meilleurs solutions trouvées..

Il faut noter que dans notre implémentation l'algorithme n'est pas optimisé dans le sens où tout passe par le contrôleur. C'est à dire, qu'il n'existe pas de duplication de résultats temporaires dans chaque travailleurs pour optimiser la communication avec le contrôleur. Ainsi, nous pouvons mieux tester l'impact de la couche du framework et des communications entre les composants et les comparer avec l'approche objets.

Implémentation en JavaParty

L'implémentation de l'algorithme utilise le langage Java avec l'environnement de distribution JavaParty. Les travailleurs sont implémentés en tant que classes distantes (remote). Le contrôleur est, par contre, une classe locale. Au lancement de l'environnement JavaParty, tout objet d'un classe distante est placé sur une machine. Ainsi, si nous avons N machine participant à l'environnement JavaParty, l'application crée N objets de la classe travailleur qui sont automatiquement placés sur une machine par l'environnement JavaParty.

Implémentation avec des Composants CCADAJ

L'implémentation du programme en composants CCADAJ nécessite de définir deux composants un (contrôleur et un travailleur) comme suit :

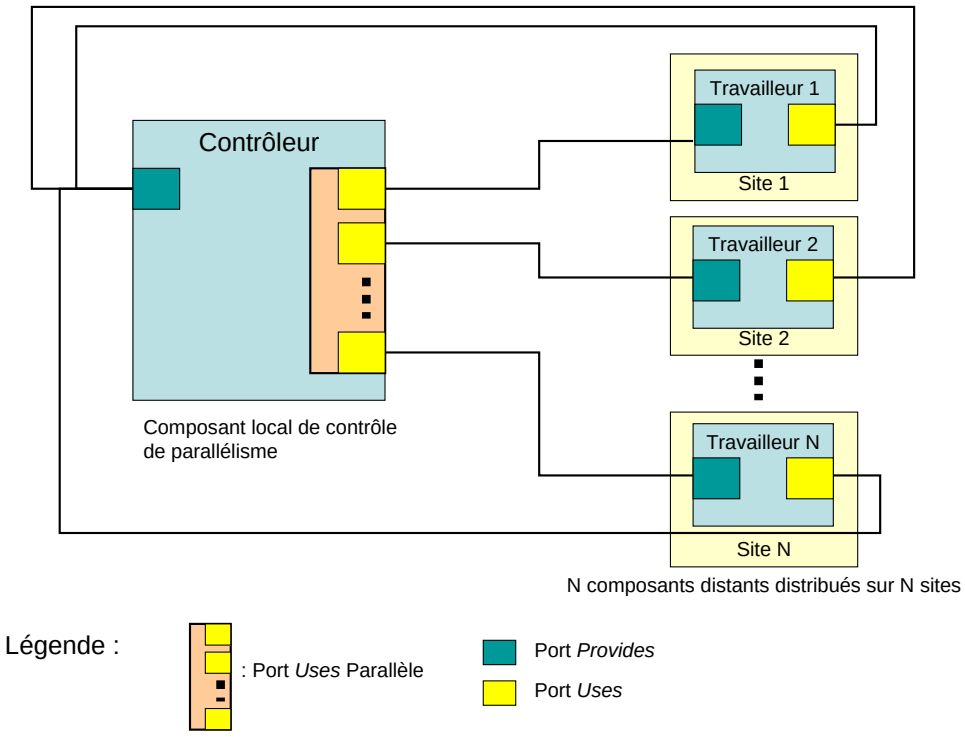

Figure 5.4 – Schéma général des Composants TSP

- **Contrôleur** : il assure l'ordonnancement de travaux entre les travailleurs, il implémente un port *uses* parallèle (présenté dans 4.2). A travers ce port de parallélisation le contrôleur distribue les tâches aux travailleurs en utilisant leurs services pour effectuer un traitement. Le retour de résultat et les conditions de demande de travail sont gérés à l'intérieur du composant travailleur. Un travailleur qui finit son traitement retourne le résultat au contrôleur et attend sur la file d'attente (queue) de travail pour effectuer un autres travail d'exploration de sous-arbre. Le contrôleur implémente un port *provides* de type `RemotePort` pour offrir aux travailleurs les informations nécessaires pour leurs traitements.

- **Travailleur** : il fait un traitement local qui est offert par le port *provides*. Ce port est l'implémentation de la procédure de l'exploration du sous-arbre envoyé comme argument par le contrôleur. Le résultat de cette procédure est la meilleure solution trouvée dans ce sous arbre ou il ne rend pas de solution si le traitement du sous arbre donne un résultat temporaire qui n'est pas mieux que la meilleure solution communiquée par le Contrôleur. Les informations nécessaires au traitement du sous-arbre sont utilisées par le Travailleur à travers son port *uses* qui utilise les informations fournies par le contrôleur (comme la meilleure solution connue).
  Le composant Travailleur est un composant distant, il est donc déployable à distance. Il est défini en utilisant le composant distant présenté dans 3.4.3.

Lors de l'initialisation de l'application et en se basant sur le nombre de machines, le framework instancie autant de Travailleurs qu'il existe de machines participant à l'environnement de l'exécution. Le composant Contrôleur est instancié avec un port *uses* parallèle qui est capable de se connecter à un nombre de travailleurs égal au nombre de machines. La connexion entre les composants est faite et l'application ensuite est lancée. Les composants Travailleur sont alors instanciés par site. Le schéma général des composants utilisés est illustré dans la figure 5.4

5.3.2 Résultats des expérimentations

Nous avons effectué des mesure sur la grille de calcul *Grid5000*[GRI06]. Nous avons utilisé 16 machines dont la configuration est donnée par le tableau 5.5.

Model	IBM eServer 326 x 53
CPU	AMD Opteron 248
	2.2 GHz
	1 MB
	800 MHz
Mémoire	4 GB
Network	Gigabit Ethernet

TAB. 5.5 – Configuration matérielle du cluster

Nous avons effectué une série de mesures sur les deux implémentations de l'application. Le tableau 5.6 montre les résultats obtenu avec l'implémentation *JavaParty*. Nous avons calculé la moyenne de chaque ligne selon le nombres de villes traité par cette application. L'augmentation de la taille du problème TSP (selon le nombre de villes) fait que le temps d'exécution augmente d'un façon exponentielle. Ce qui implique aussi un nombre de communications important entres les différents objets.

Nombre de villes	Moyenne (sec)	Temps (sec)		
16	34	39,2	29,343	32,534
17	49	36,609	60,598	48,695
18	79	94,735	65,706	75,76
19	611	653,737	471,364	706,562
20	2967	2737,064	3287,83	2876,53

TAB. 5.6 – Mesures de l'implémentation JavaParty

Une autre série de mesures a été faite sur l'implémentation CCADAJ de l'application. Le tableau 5.7 montre le temps d'exécution selon la taille du problème.

Nombre de Villes	Moyenne (sec)	Temps (sec)		
16	38	42,939	36,149	34,14
17	53	45,77	55,36	57,442
18	82	95,751	59,737	89,657
19	669	953,211	489,452	564,06
20	3491	3483,122	4029,542	2961,315

TAB. 5.7 – Mesures de l'implémentation CCADAJ

Pour comparer les deux temps d'exécution et mesurer le surcoût induit par le framework CCADAJ, le tableau 5.3.2 a été créé pour illustrer ce surcoût ainsi que les variations.

Nombre de villes	JavaParty	CCADAJ	Surcoût %
16	34	38	10,42
17	49	53	8,68
18	79	82	3,79
19	611	669	9,56
20	2967	3491	17,65
Surcoût moyen			10,02

TAB. 5.8 – Surcoût du framework CCADAJ

Nous remarquons que le surcoût moyen du framework est de environ 10,2%.

La figure 5.5 montre l'évolution de temps d'exécution.

Nous remarquons certaines variations dans le surcoût du framework. Il est à 3,79% pour 18 villes. Il atteint le 17,65 dans le cas où l'application traite 20 villes. Ces variations

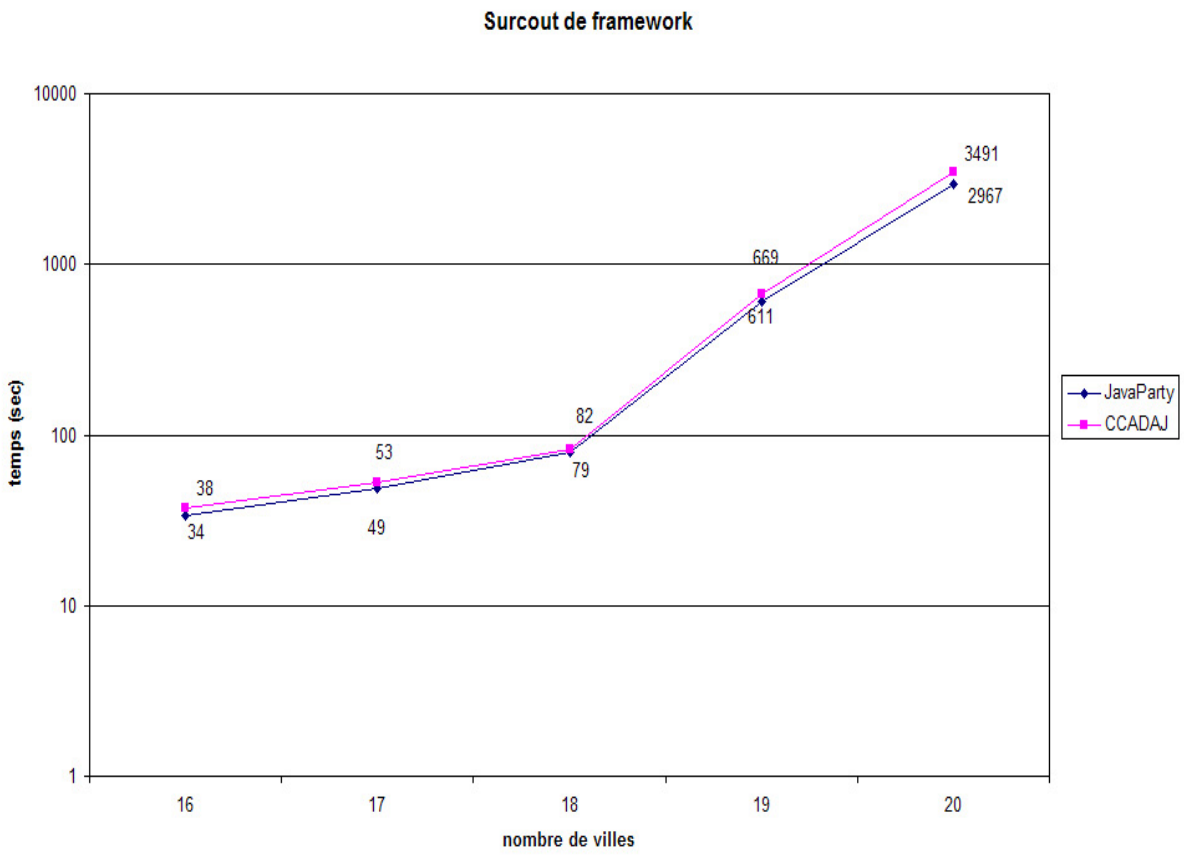

Figure 5.5 – CCADAJ contre objets Java avec JavaParty

dépendent de la façon dont les accès entre le contrôleur et les travailleurs se font. Plus nous avons d'accès et des communications distantes, plus il existe des traversées de couches différentes du framework, qui est centralisé. Chaque interrogation du framework fait l'objet d'une communication distante. Nous remarquons aussi que pour une série d'exécutions pour la même taille de problème, nous trouvons des variations dans le temps d'exécution. Ces variations sont liées au type de problème considéré. En général, nous pouvons dire que le framework a un surcoût qui n'est pas loin du surcoût trouvé dans la section 5.2.2. Ce surcoût est lié à l'encapsulation des composants dans des super-composants distants pour pouvoir être réutilisés à distance. Pour éviter l'encapsulation des composants de base dans des composants distants, nous travaillons pour améliorer le mécanisme d'accès distant entre les composants sans passer par l'encapsulation et donc éviter une traversée de couches liées aux super-composants. Une autre amélioration de la performance, est obtenu en distribuant des parties du framework. Au début de l'exécution du framework des parties du framework sont copiées et placées dans les différents sites qui accueillent les composants. Avec cette procédure l'interrogation du framework est faite localement. Seules les informations globales sont à transmettre au framework central.

5.3.3 Réutilisation des composants CCADAJ

Le framework CCADAJ a été conçu pour rendre la réutilisation de logiciels plus facile et aisée. D'une part, il implémente un modèle de composants dont le caractéristique principale est d'être destiné à des applications de calcul distribué et/ou parallèle. La communauté CCA a fourni des spécifications *légères* pour ne pas affecter la performance de l'exécution de l'application. CCADAJ implémente ces spécifications en y ajoutant la construction d'un composant à partir d'autres composants tout en respectant la compatibilité avec les composants de base. De plus, CCADAJ fournit des structures de parallélisation sous forme de composants qui facilitent la construction d'application distribuée et/ou parallèle avec des composants préfabriqués.

La réutilisation des composants existants avec un minimum de surcoût est le but de la construction d'un environnement à base de composants. Les composants de contrôle sont un moyen d'aide à la construction d'une application dans un environnement parallèle réparti. Dans la section précédente, nous avons estimé le surcoût de l'utilisation des composants logiciels dans un environnement de distribution, basé sur les objets Java. Le surcoût de l'utilisation des composants pourrait être moins important par rapport à l'utilisation directe des objets Java. Le surcoût dû à l'utilisation du framework est donc faible. L'utilisation de certains composants peut améliorer la performance de la création d'une application. Des composants déjà construits et présentés dans une bibliothèque de composants de contrôle du type composant de parallélisation et pipeline aident le constructeur d'une application en lui permettant d'utiliser ces composants sans qu'il ait à modifier les codes pour le paralléliser.

Les composants de parallélisation et pipeline présentés dans le chapitre 4 sont un moyen simple de distribuer un traitement sur plusieurs composants. La composition d'une application, dans laquelle le composant de parallélisation est utilisé, est réalisée à l'aide du framework de composition et les connections entre les composants sont transparentes.

La séparation entre le framework de composition et de connexion et l'environnement d'exécution rend la construction transparente. Le framework initie seulement l'environnement d'exécution alors que la composition de l'application se fait dans un seul espace.

5.4 Conclusion

Dans ce chapitre nous avons traité des questions de performance sur l'utilisation de la programmation par composants. La performance des applications est affectée par l'utilisation de couches supplémentaires dans l'architecture du logiciel (c.f. 1.3.4). Il existe un surcoût dû à l'insertion de ces différentes couches. Cependant le gain que l'on obtient en temps de conception et programmation de l'application est très important.

Nous nous sommes intéressés aux propositions de la communauté CCA, qui donne des spécifications pour une architecture logicielle pour les applications du calcul parallèle et/ou distribué. Avant d'entrer dans les évaluations que nous avons menées sur le framework CCADAJ (basé sur le modèle CCA), nous avons voulu montrer les considérations de performance et les évaluations effectuées par les créateurs de CCA, afin de mieux comprendre l'impact de l'utilisation des composants et du framework de composants sur

la performance d'une application. CCA repose sur la définition de la connexion entre les composants qui est un modèle de patron du type *uses/provides*. Ce mode de connexion est très efficace dans la mesure où les deux composants connectés sont dans le même espace d'adressage. L'appel entre les port s'effectue avec la méthode `getPort()` avec un coût équivalent à celui d'un appel d'une fonction. Quand les deux composants connectés sont distribués, le coût devient plus important car le nombre de couches à traverser augmente. Dans ce cas il existe un coût supplémentaire dû au framework et à son intégration avec le système de communication et de distribution.

Nous avons effectué plusieurs séries de mesure concernant l'impact de l'utilisation de composants CCADAJ en local. L'utilisation de composants CCADAJ introduit un surcoût dû à l'utilisation de ce framework. Nous avons comparé l'utilisation d'objets simples Java et l'utilisation des composants CCADAJ. Certes il existe un surcoût de l'ordre de 3.3% mais la réutilisabilité des composants est plus importante et les composants sont bien masqués et mieux modélisés. Ceci est valable dans le cas d'une utilisation du super-composant qui a un surcoût de l'ordre de 6.3%, dû aux couches introduites dans ce composant. Cependant, le développeur d'un super-composant peut avoir le choix de ne pas encapsuler des composants dans un super composant s'il y accède beaucoup. Cela évitera les traversées de couche à chaque fois qu'il veut accéder aux composants internes.

Pour intégrer le framework de composants dans l'environnement ADAJ, nous avons étendu le modèle pour prendre en compte les connections des composants distribués et pour encapsuler des composants dans des conteneurs distribués sous forme de super-composant distant. L'encapsulation a un surcoût équivalent au surcoût du super-composant auquel on ajoute le surcoût apporté par l'intégration du framework dans le système de communications de JavaParty. Le surcoût d'utilisation des composants distants est autour de 11,79%.

Pour mieux évaluer le framework nous avons implémenté une application de TSP (*Travelling Salesman Problem*) distribuée/parallèle en maître/esclave qui demande beaucoup de communication entre les différents composants esclaves distants et le maître central. Nous avons remarqué une variation dans le surcoût de framework due à la taille de problème et aux communications entre les esclaves et le maître d'un côté et les communications pour des informations demandées au framework centralisé, d'un autre côté. Une série de mesures a été faite pour tester cet impact. Les mesures montrent un surcoût moyen autour de 10,02%. Nous expliquons cette variation par le fait que les communications dans le cas de TSP sont imprévisibles par rapport à un simple programme qui fait une boucle d'appel aux composants distants.

Finalement, nous pouvons déduire que l'évaluation n'est pas seulement celle de la performance de l'exécution de l'application. Certes, il existe un surcoût. Mais faut-il en tenir compte lors du choix pour construire son applications avec des méthodes de programmation traditionnelle ou en se basant sur des composants ? Dans ce cas, le surcoût n'est pas très important en comparaison du gain apporté par développement à base de composants concernant la réutilisabilité, la modularité, la construction et la transparence de l'utilisation des composants.

Conclusion générale

Les travaux que nous avons réalisés et qui sont présentés dans ce mémoire, concernent la construction du framework CCADAJ (*CCA-ADAJ*) au dessus de la plate-forme ADAJ (*Adaptive Distributed Applications in Java*). Le framwork CCADAJ est un framework de composants qui permet la réutilisation et la composition de composants logiciels dans un environnement distribué/parallèle. Les applications concernées par ces travaux sont les applications de calcul distribué/parallèle. Par l'introduction du framework CCADAJ dans la plateforme ADAJ, il est possible de créer et de bénéficier des avantages qu'apporte l'approche composants. De plus les constructeurs des applications bénéficient des mécanismes offerts par la plateforme ADAJ concernant l'observation, le placement et l'équilibrage de charges des objets distribués.

Dans le contexte des applications distribuées parallèles, la construction d'un framework de composants et son intégration dans un environnement distribué ne doit pas avoir un surcoût très important pour la performance de l'application. La construction d'un framework de composants respecte aussi certaines règles d'architecture logicielle à base de composants. Ces règles sont présentées selon un modèle de composants. Dans le cadre du développement à base de composants, plusieurs standards industriels ont été conçus pour répondre à des besoins spécifiques dans la construction des applications à base de composants. Mais les technologies industrielles ne s'adressent pas aux applications de calcul distribué/parallèle. Des standards de recherche ont été présentés pour prendre en considération les besoins de ce type d'application. CCA (*Common Component Architecture*) est un standard qui vise les applications de calcul scientifique distribué et/ou parallèle. Ces spécifications promettent une performance d'exécution lié à la définition d'un patron de conception de type *uses/provides* pour les connexions entre les composants. Motivés par les considérations de performance présentées par le forum CCA, nous avons donc implémenté notre framework CCADAJ selon les spécifications CCA.

Avantages et surcoûts

Les besoins de construction des applications à base de composants nous ont incité à fournir de nouveaux éléments dans l'implémentation du CCADAJ. Il existe un gain important au niveau de la conception et de la construction des applications découlant de la possibilité d'utiliser et de relier des composants existants. Les facilités apportées par le framework CCAADAJ lors de la construction d'applications sont :

Le super-composant : Le modèle CCA ne traite pas la composition hiérarchique. Cette composition permet de construire des composants par les compositions d'autres

composants. Nous avons ajouté au modèle de base une définition d'une structure que l'on appelle un super-composant. Pour cela, nous distinguons deux types de composant : des composants de base et des super-composants. Un super-composant contient des composants de base ou des super-composants, alors qu'un composant de base est un composant simple qui encapsule des objets et définit des interfaces de connexion (ports) avec d'autres composants. Les services de framework comme la connexion et l'instanciation ont été modifiés pour prendre en compte les connexions entre les différents types de composants.

Le composant distant : Pour intégrer le framework de composants CCADAJ dans l'environnement ADAJ nous avons besoin d'un mécanisme pour rendre des composants déjà développés accessibles à distance sans devoir les réécrire. Nous avons défini pour cela, un composant spécial basé sur la définition d'un super-composant mais qui peut être accédé à distance dans l'environnement ADAJ. Nous avons appelé ce composant "conteneur distant". Ce composant utilise des ports spéciaux que nous avons défini et que nous avons appelé "ports distants". Avec ce mécanisme, il est possible d'encapsuler un composant dans des conteneurs distants pour qu'il soit déployable dans l'environnement ADAJ au travers de JavaParty. Ce dernier rend transparente la distribution des composants distants.

Le super-composant parallèle : La définition d'un super composant permet de fabriquer un composant à partir d'autres composants. Cette considération est utilisée dans le cas où les composants internes sont des composants distants. Nous construisons alors un composant parallèle dont les composants internes sont répartis dans l'environnement de distribution. Si un composant contient des objets distants définis selon l'environnement ADAJ, nous sommes dans un cas de composant parallèle.

Les composants de contrôles : Des composants spécifiques aux applications distribuées/parallèles sont définis pour permettre de structurer plus facilement une application parallèle. Nous avons appelé ces composants "composants de contrôle". Ces composants spéciaux facilitent la création de structures parallèles. Ces composants :

- le distributeur : distribue un ensemble de données vers différents composants de calcul et coordonne le retour de résultats. Il peut être connecté à un composant adaptateur pour le retour asynchrone de résultats.
- le collecteur : est un complémentaire du distributeur. Il collecte les données fournies par les composants de calcul et envoie l'ensemble à d'autres composants qui en ont besoin. Il peut aussi être connecté à un composant adaptateur comme pour le distributeur.
- le pipeline : offre une fonctionnalité du pipeline logiciel. Un flux de données passe par les étages du pipeline. Les étages sont connectés à des différents composants de calcul. Il existe deux modes d'opération du pipeline : disponibilité de données et demande de données.

De plus, le framework CCADAJ offre un moyen de définir des ports générique pour les connections entre des composants sans prendre en compte les types d'interfaces des ports lors de la connexion. Cela permet de faire la connexion entre deux composant sans tester les types d'interfaces. Les types d'interfaces sont donc génériques. L'utilisation de cette généricité rend la construction de composants plus facile. Mais lors de l'exécution,

les types sont raffinés. Si l'utilisateur connecte deux composants dont les méthodes internes des ports ne sont pas compatibles, une erreur se produit. Le framework CCADAJ fournit, aussi, un moyen de distribution transparente des composants distants apporté par l'utilisation de l'environnement JavaParty dans la plate-forme ADAJ.

L'utilisation du framework et des structures proposées facilite la conception et la construction des applications. Mais l'introduction d'un framework dans l'environnement affecte la performance des applications. Le framework CCADAJ intégré au dessus de l'environnement ADAJ induit un surcoût lié aux couches ajoutées. Les activations et les communications des objets dans un environnement à objets sont directs alors que les interactions entre les composants passent par des couches de framework.

Nous avons voulu tester l'impact de l'utilisation des composants et du framework CCADAJ sur la performance des applications. Le framework CCADAJ introduit un surcoût lors de l'utilisation des composants CCADAJ simple par rapport à des objets Java. L'utilisation du super-composant a un surcoût plus important par rapport à l'utilisation des composants de base CCADAJ. Avec le surcoût du super-composant, l'utilisation du composant distant a un surcoût similaire au super-composant en y ajoutant le surcoût apporté par les couches liées à l'intégration dans l'environnement de distribution d'ADAJ.

Perspectives et améliorations

Actuellement le projet DG-ADAJ (Desktop-Grid ADAJ) est en cours d'amélioration et d'intégration de l'ensemble des éléments offerts. Les mécanismes d'observation, de placement et d'équilibrage de charge que propose ADAJ sont offerts sous forme d'API pour la conception des applications orientées objets distribuée/parallèle. Nous sommes en train de travailler sur ces API pour les écrire sous forme de composants et les ajouter à la bibliothèque de composants que nous avons proposée. L'utilisation de ces composants va rendre la construction des applications plus facile et permettra de bénéficier des mécanismes offerts par ADAJ pour améliorer la performance de l'application distribuée.

De plus, en se basant sur les résultats des expérimentations que nous avons faites, nous pouvons apporter plus de performance pour les applications en utilisant les propriétés de JavaParty qui sont offertes sous forme orientée objets. Nous pouvons profiter de certains aspects qu'offre JavaParty en terme de migration d'objets et l'appliquer à des composants. JavaParty offre la migration d'un groupe d'objets. Nous pouvons prendre en considération cet aspect pour l'appliquer à un composant (qui contient plusieurs objets). De plus un mécanisme de duplication d'objets distribués est offert par JavaParty et dont ADAJ bénéficie dans l'implémentation de ses API concernant l'observation, la migration et l'équilibrage de charge, peut être utilisé dans des parties du framework pour être dupliquées dans chaque site participant à l'environnement. Cette duplication de quelques parties du framework peut rendre les communications entre les composants dans le même site avec le framework moins coûteuse.

Bibliographie

[AAW+02] B. Allan, R. Armstrong, A. Wolfe, J. Ray, D. Bernholdt, and J. Kohl. The CCA Core Specification In A Distributed Memory SPMD Framework. *Concurrency and Computation : Practice and Experience*, 14(5) :323–345, 2002.

[ABP04] F. André, J. Buisson, and J.-L. Pazat. Dynamic parallel codes : toward self adaptable components for the Grid. In *ICS'04 Workshop on Component Models and Systems for Grid Applications*, June 2004.

[Acl99] E. Acly. Middleware and Businessware : 1999 Worldwide Markets and Trends. Technical report, International Data Corporation, 1999.

[AF05] Alcatel and FPX. MicoCCM. http ://www.fpx.de/MicoCCM, revisited on february 2005.

[AGG+99] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *HPDC '99 : Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.

[Alm97] P.S. Almeida. Balloon types : Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241 :32–59, 1997.

[AMR99] D. Acremann, G. Moujeard, and L. Rousset. *Développer avec CORBA*. CampusPress, 1999.

[Ant02] Inria Sophia Antipolis. *ProActive : A Java library for parallel, distributed, and concurrent computing*. http ://www-sop.inria.fr/oasis/ProActive/, 2002.

[AOT04] I. Alshabani, R. Olejnik, and B. Toursel. Parallel tools for a distibuted components framework. In *International Conference On Information and Communication Technologies : From Theory To Applications*, Damascuc, Syria, April 2004.

[AOT06] I. Alshabani, R. Olejnik, and B. Toursel. A Framework for Desktop GRID Applications : CCADAJ. In *International Conference On Information and Communication Technologies : From Theory To Applications*, Damascus, Syria, 2006.

[Asp05] Aspectj home page. http ://www.aspectj.org, 2005.

[AT03] I. Alshabani and B. Toursel. CFDA : A CCA- based Component Framework for Distributed Applications. In *Concurrent Information Processing*

and Computing, the NATO Advanced Research Workshop, Sinaia Romania, July 2003.

[AWB+93] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In *ECOOP Workshop*, pages 152–184, 1993.

[BAA04] D. Bernholdt, R. Armstrong, and B. Allan. Managing Complexity in Modern High End Scientific Computing through Component-Based Software Engineering. In *Proc. of HPCA Workshop on Productivity and Performance in High-End Computing (P-PHEC 2004), Madrid, Spain*, 2004.

[BBC02a] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible and typed group communications for java. In *Joint ISCOPE 2002 Conference*, Washington, Novembre 2002. ACM Java Grande.

[BBC02b] L. Baduel, F. Baude, and D. Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Joint ACM Java Grande - ISCOPE Conference*, pages 28–36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.

[BBD+98] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan. OVERTURE : An object-oriented framework for high-performance scientific computing. In *IEEE/ACM Conference on Supercomputing*, pages ? ?–? ?, 1998.

[BCD+00] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M.i Yechuri. A component based services architecture for building distributed applications. In *HPDC*, pages 51–, 2000.

[BCH+02] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere. Interactive and descriptor-based deployment of object-oriented grid applications. In *HPDC-11 : High Performance Distributed Computing*, pages 93–102, Edinburg, juillet 2002.

[BCL+04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in java. In *Seventh International Symposium on Component-Based Software Engineering*, volume 36, pages 11–12, 2004.

[BCM03] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components, 2003.

[BCS04] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model specification. The ObjectWeb Consortium, February 2004. version 2.0-3.

[BDV+98] F. Breg, S. Diwan, J. E. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java rmi performance and object model interoperability : experiments with java/hpc++. *Concurrency - Practice and Experience*, 10(11-13) :941–955, 1998.

[BEK+00] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1. Technical report, W3C, may 2000.

[BEKE02] D. Bernholdt, W. Elwasif, J. Kohl, and T. Epperly. A Component Architecture for High-Performance Computing. In *Proceedings of the Workshop on*

Performance Optimization via High-Level Languages and Libraries (POHLL-02), 2002.

[Ber96] P. Bernstein. Middleware : A model for distributed services. *Communications of the ACM*, 39(2) :86–97, February 1996.

[BFT04] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin : A Theorem Prover for the Model Evolution Calculus. In Stephan Schulz, Geoff Sutcliffe, and Tanel Tammet, editors, *IJCAR Workshop on Empirically Successful First Order Reasoning (ESFOR (aka S4))*, Electronic Notes in Theoretical Computer Science, 2004.

[BGVW99] R. Bramley, D. Gannon, J. Villacis, and A. Whitaker. Using the grid to support software component systems, 1999.

[BJPW99] A. Beugnard, J-M. Jezequel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7) :38–45, July 1999.

[Ble01] D. Blevins. *Overvieuw of the Enterprise JavaBeans Component Model.* Component-Based Software Engineering, 2001.

[BLL00] A. Bouchi, E. Leprêtre, and P. Lecouffe. Un mécanisme d'observation des objets distribués en Java. In *Rencontres Francophones du Parallélisme des Architectures et des Systèmes (RenPar'12)*, pages 171–176, Besançon, France, Avril 2000.

[Bor06] Borland. Delphi web page. http ://www.borland.com/us/products/delphi/index.html, 2006.

[BOT01a] A. Bouchi, R. Olejnik, and B. Toursel. Java tools for measurement of the machine loads. In *NATO Advanced Research Workshop. IWCC : International Workshop on Cluster Computing*, Mangalia, Romania, Septembre 2001. IEEE.

[BOT01b] A. Bouchi, R. Olejnik, and B. Toursel. Java Tools for Measurements of the Machines Loads. In *"NATO Advanced Research Workshop Romania - Advanced Environments, Tools and Applications for Cluster Computing"*, Mangalia, Roumanie, Septembre 2001.

[Bou03] A. Bouchi. *Proposition d'un mécanisme d'observation dynamique de l'exécution d'applications Java distribuées.* PhD thesis, Université des Sciences et Technologies de Lille, 2003.

[BR00] E. Bruneton and M. Riveill. Javapod : une plate-forme à composants adaptable et extensible. Technical Report 3850, INRIA, Janvier 2000.

[BWD+93] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra : A structure description language for developing distributed applications. *IEEE Software Engineering Journal*, pages 83–94, march 1993.

[Car89] D. Caromel. Service, asynchrony and wait-by-necessity. *JOOP : Journal of Object Orientated Programming*, pages 12–22, Novembre 1989.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). Technical report, W3C, Ariba, IBM, Microsoft, 2001.

[CDF+02] F. Cappello, A. Djilali, G. Fedak, C. Germain, O. Lodygensky, and V. Néri. *Calcul réparti à grande échelle Metacomputing*, chapter XtremWeb : une plate-forme de recherche sur le Calcul Global et Pair à Pair. Lavoisier, 2002.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems : Concepts and Design*. Addison-Wesly, 2001.

[Cha96] D. Chappel. *Understanding ActiveX and OLE*. Microsoft Press. ISBN 1-57231-216-5, 1996.

[CKV98] D. Caromel, W. Klauser, and J. Vayssière. Towords seamless computing and metacomputing in java. *Concurrency : Practice and Experience*, 10(11–13) :1043–1061, Novembre 1998.

[CPR03] Thierry Priol Christian Pérez and André Ribes. A parallel corba component model for numerical code coupling. 2003.

[Crn01] I. Crnkovic. Component-based software engineering - new challenges in software development. *Focus review*, 2(4) :127–133, winter 2001.

[Dal96] Rogerson Dale. *Inside COM, Microsoft Component Object Model*. Microsoft Press, 1996.

[Dan02] Jérôme Daniel. *Au cœur de CORBA*. Vuibert, 2002.

[DGK+98] Jack Dongarra, Al Geist, James Arthur Kohl, Philip Papadopoulos, and Vaidy Sunderam. HARNESS : Heterogeneous adaptable reconfigurable networked systems. In *Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998.

[DK76] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Trans. Software Eng.*, 2(2) :80–86, 1976.

[DPP03] A. Denis, C. Pérez, and T. Priol. PadicoTM : An Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19 :575–585, 2003.

[EE99] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1999.

[EG95] R.Johnson J.Vlissides E.Gamma, R.Helm and G.Booch. *Design Patterns : Elements of reusable Object-Oriented Software*. Addison-Wesley, 1995.

[FIA96] B. FIAT. *Le client-serveur(1) : concepts de base*. Computer Channel. En collaboration avec l'AFPA, septembre 1996.

[FK97a] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[FK97b] I. Foster and C. Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[FK00] I. Foster and C. Kesselman. Computational Grids. In *VECPAR'2000 - 4th International Meeting on Vector and Parallel Processing*, Porto, Portugal, 2000.

[FKT01] I. Foster, C.l Kesselman, and S. Tuecke. The anatomy of the grid : Enabling scalable virtual organizations. *International Journal Supercomputer Applications*, 15(3), 2001.

[FR97] E. Frank and I. Redmond. *DCOM : Microsoft Distributed Component Object Model*. IDG Books worldwides, 1997.

[FT02] V. Felea and B. Toursel. Methodology for java distributed and parallel programming using distributed collections. In *IPDPS : International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, Avril 2002. ACM.

[FTD03] V. Felea, B. Toursel, and N. Devesa. Les collections distribuées : un outil pour la conception d'applications Java parallèles. *Technique et science informatiques*, 22(3) :289–314, 2003.

[GBJ+01] D. Gannon, P. H. Beckman, E. Johnson, T. Green, and M. Levine. HPC++ and the HPC++Lib Toolkit. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 73–108, 2001.

[GdRB91] G.Kiczales, J. d. Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[GKC+02] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0 : Design and Implementation of Component based Web Services. Technical Report TR562, Department of Computer Science, Indiana University, Bloomington, June 2002.

[GRI06] GRID5000. http ://www.grid5000.fr. http ://www.grid5000.fr, last visited 2006.

[GvLM99] J. A. Insley I. Foster J. Bresnahan C. Kesselman M. Thiebaux M. L. Rivers S. Wang B. Tieman G. von Laszewski, M.-H. Su and I. McNulty. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[GW97] A. S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1) :39–45, January 1997.

[GWK98] Liz Ungar Guijun Wang and Dan Klawitter. A framework supporting component assembly for distributed systems. In *Proceedings of the Second Enterprise Distributed Object Computing*, pages 136–146, San Diego, CA, November 1998.

[HKM+03] P. Hovland, K. Keahey, L. C. McInnes, B. Norris, L. F. Diachin, and P. Raghavan. A quality of service approach for high-performance numerical components. In *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference, Toulouse, France*, June 2003. (also available as Argonne preprint ANL/MCS-P1028-0203).

[HMP97] B. Haumacher, T. Moschny, and M. Philippsen. *JavaParty : A distributed companion to Java*. http ://www.ipd.uka.de/JavaParty/, 1997.

[Hog91] J. Hogg. Islands : Aliasing protection in object-oriented languages. In *OOPSLA*, pages 271–285, 1991.

[HST99] P. Homburg, M.V. Steen, and A.S. Tanenbaum. Globe : A wide-area distributed system. *IEEE Concurrency*, pages 70–78, Jan-Mar 1999.

[IBM01] IBM. Hyperspace home page. http ://www.research.ibm.com/hyperspace/, 2001.

[IF05] A. Savva D. Berry A. Djaoui A. Grimshaw B. Horn F. Maciel F. Siebenlist R. Subramaniam J. Treadwell J. Von Reich. I. Foster, H. Kishimoto. The open grid services architecture. Technical report, Global Grid Forum, 2005.

[J.B96] J.Bosch. Proceedings tools europe'96. In *Language Support for Design Patterns*, 1996.

[JGP99] C.Gransart J.M Geib and P.Merle. *CORBA : des concepts à la pratique.* Dunod, 1999.

[JO01] J.Bézivin and O.Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the Conference on Autonomous Software Engineering*, 2001.

[KA99] R. Scheifler J.Waldo K.Arnold, B.O'Sullivan and A.Woolrath. *The Jini Specification.* Addison-Wesley, 1999.

[Kar96] J. F. Karpovich. Support for Object Placement in Wide Area Heterogeneous Distributed Systems. Technical Report CS-96-03, University of Virginia Department of Computer Science, Janvier 1996.

[K.B95] K.Brockschmidt. *Inside OLE.* Microsoft Press, 1995.

[KBG+01] S. Krishnan, R. Bramley, D. Gannon, M. Govindaraju, R. Indurkar, A. Slominski, B. Temko, R. Alkire, T. Drews, E. Webb, and J. Alameda. The XCAT Science Portal. In *Proceedings of SuperComputing Conference, Denver, Colorado*, November 10–16 2001.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic92] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proc. of IMSA'92 Workshop on Reflection and Meta-Level Architecture*, 1992.

[K.W02] K.Whitehead. *Component-Based Development-Principles and Planning for Business Systems.* Addison-Wesley, 2002.

[Lab04] Lawrence Livermore National Laboratory. The babel project home page. http ://www.llnl.gov/casc/components/babel.html, Jan 2004.

[LG96] M. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE Conference on High Performance Distributed Computing HPDC5*, pages 551–561, Syracuse NY, 1996.

[LGC05] M. J. Lewis, M. Govindaraju, and K. Chiu. Exploring the Design Space for CCA Framework Interoperability Approaches. In *Workshop on Component Models and Frameworks in High Performance Computing*, Atlanta, GA, June 2005.

[LGK06] M. J. Lewis, M. Govindaraju, and K.Chiu. Design and Implementation Issues for Distributed CCA Framwork Interoperability. *Concurrency and Computation : Practice and Experience*, 2006.

[L.L97] L.Lamport. Composition : A way to make proofs harder. Technical report, Compaq Systems Research Center, 1997.

[LRN03] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, Los Alamitos, California, USA, April 2003. IEEE Computer Society.

[Mey99] B. Meyer. On to components. *IEEE Computer*, 32(1) :193–140, January 1999.

[Mey00] B. Meyer. Contracts for components. *Software Development Magazine*, octobre 2000.

[Mic98] Microsoft. Distributed component object model. Technical report, Microsoft, 1998.

[MM99] B. Meyer and C. Mingins. Component-based development : From buzz to spark. *IEEE Computer*, 32(7) :35–37, july 1999.

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide (Draft Version 0.2)*. `{http://www.omg.org/docs/ab/03-01-03.pdf}`, 2003.

[Mor94] J. P Morrison. *Flow-Based Programming : A New Approach to Application Development*. van Nostrand Reinhold, New York, NY, 1994.

[MS99a] M. Migliardi and V. Sunderam. Heterogeneous distributed virtual machines in the harness metacomputing framework. In *Proc. of the Heterogeneous Computing Workshop of IPPS/SPDP 1999*, pages 60–73, San Juan de Puerto Rico, April 1999.

[MS99b] M.Migliardi and V. Sunderam. The Harness metacomputing framework. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[NPH99] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 153–159, San Francisco, CA, Juin 1999.

[OFAT06] R. Olejnik, V. Fiolet, I. Alshabani, and B. Toursel. Desktop grid platform for data mining applications. In *International Symposium on Parallel and Distributed Computing, ISPDC-06*, Timisoara, Romania, July 2006.

[OMG] *Object Management Group* . http ://www.corba.org.

[OMG99] OMG. Corba component model joint revised submission. Technical report, Object Management Group, 1999.

[OMG00] OMG. Meta object facility (mof), version 1.3. Technical report, Object Management Group, 2000.

[OMG01a] OMG. Corba 3.0 new components chapters. Technical report, OMG, 2001.

[OMG01b] OMG. Model driven architecture. Technical report, OMG, 2001.

[PPR04] C. Pérez, T. Priol, and A. Ribes. PaCO++ : A Parallel Object Model for High-Performance Distributed Systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.

[PZ97] M. Philippsen and M. Zenger. Transparent remote objetcts in java. *Concurrency : Practice and Experience*, 11(9) :1125–1242, 1997.

[RC00] Pete Dean Rich Detry Ernest Friedman-Hill Victor Holmes Carl Melius David Miller John Mitchiner Patrick Moore Ly Sauer Lee Taylor Bob Whiteside Robert Clay, Rob Armstrong. Sandia's advanced software interoperability architecture and strategy. Technical Report v1.0a, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, April 2000.

[RD00] G. Roussel and E. Duris. *Java et internet : concepts et programmation*. Paris : Vuibert Informatique, 2000.

[Red97] Frank E. III Redmond. *DCOM : Microsoft Distributed Component Object Model*. Hungry Minds, Inc, 1997.

[Rey97] John Reynders. The pooma framework - a templated class library for parallel scientific computing. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[Rom99] Ed Roman. *Mastering Enterprise JavaBeans*. Wiley Computer Publishing, 1999.

[RP00] C. Renée and T. Priol. MPI Code Encapsulating Using Parallel CORBA Object. *Cluster Computing*, 3(4) :255–263, 2000.

[RPS03] R. Reussner, I. Poernomo, and H. Schmidt. Contracts and quality attributes for software components. In *Proceeding of the 8th Int'l Workshop on Component-Oriented Programming*, 2003.

[RRA+02] Lavanya Ramakrishnan, Helen Nell Rehn, Jay Alameda, Rachana Ananthakrishnan, Madhusudhan Govindaraju, Aleksander Slominski, Kay Connelly, Von Welch, Dennis Gannon, Randall Bramley, and Shawn Hampton. An authorization framework for a grid based common component architecture. In *Proceedings of the 3rd International Workshop on Grid Computing, Baltimore, Maryland*, pages 169–180. Springer Press, November 18 2002.

[Sch95] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10) :65–74, 1995.

[SGGB01] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software :Beyond Object-Oriented Programming*. Component Software. Adison-Wesley, second edition edition, 2002.

[Sha01] Bill Shannon. Java 2 platform, enterprise edition specification. Technical report, Sun Microsystems, 2001.

[SKR01] Jeff PAinter Scott Kohn, Gary Kumfert and Cal Ribbens. Divorcing laguage dependencies from a scientific software library. In Socity for Industrial and Applied Mathematics, editors, *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Compouting*, 2001.

[SMF02] Kate Keahey Sue Mniszewski and Pat Fasel. Paws : Collective interactions and data transfers. 2002.

[Sun] Java 2 enterprise edition home page.

[Sun97] Javabeans api specification, 1997.

[SUN98a] *Sun Products - Remote Method Invocation JDK1.2.* http ://-java.sun.com/products/ jdk/1.2/docs/guide/rmi/index.html, 1998.

[Sun98b] Sun. *Java core reflection.* http ://-java.sun.com/products/jdk/1.1/docs/guide/reflection/, 1998.

[Sun98c] Sun. *Java remote method invocation specification.* ftp ://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf, 1998.

[Sun00] Java 2 sdk standard edition documentation, 2000.

[Szy00a] C. Szyperski. Components and architecture. *Software Development Magazine*, octobre 2000.

[Szy00b] C. Szyperski. Components and contracts. Software Development Magazine, May 2000.

[TC01] George T.Heineman and Wiolliam T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[TS02] A. S. Tanenbaum and M. V. Steen. *Distributed Systems : Principles and Paradigms*. Prentice Hall, 2002.

[UML97] UML. Uml summary. Technical report, Rational Software Corporation, 1997.

[VB99] Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA*, pages 82–96, 1999.

[VD99] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. *Lecture Notes in Computer Science*, 1616 :250–? ?, 1999.

[VGS+99] J. Villacis, M. Govindaraju, D. Stern, A. Whitaker, F. Breg, P. Deuskar, B. Temko, D. Gannon, and R. Bramley. Cat : A high performance distributed component architecture toolkit for the grid, 1999.

[Vin97] Steve Vinoski. Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), February 1997.

[WPS99] G. Y. Fu X. Z. Tang H. R. Strauss W. Park, E. V. Belova and L. E. Sugiyama. Plasma simulation studies using multilevel physics models. *Physics of Plasmas*, 6 :1796–1803, 1999.

[WUK99] G. Wang, L. Ungar, and D. Klawitter. Component assembly for oo distributed systems. *IEEE Computer*, 32(7) :71–78, July 1999.

Un framework pour les composants logiciels distribués et parallèles

Résumé. Dans cette thèse nous nous sommes intéressés aux applications distribuées/parallèles à base de composants. Notre contribution se situe dans la construction d'un framework de composants (CCADAJ) au dessus de la plateforme DG-ADAJ (Desktop Grid - Adaptive Distributed Applications in Java). La plateforme DG-ADAJ propose des mécanismes d'observation, de placement et de migration d'objets pour les applications distribuées/parallèles. Le framework CCADAJ est basé sur le modèle de composants CCA (Common Component Architecture). CCA permet une simplicité d'écriture, une efficacité d'exécution et la dynamicité dans la construction des applications parallèles. Notre démarche consiste à étendre le modèle CCA pour prendre en compte la composition hiérarchique, intégrer les services de framework dans la plateforme DG-ADAJ et construire certaines structures spécifiques à la construction des applications distribuées/parallèles. Nous avons introduit une structure nommée super composant pour la composition hiérarchique. De plus, nous avons construit des structures spécifiques à la construction des applications distribuées/parallèles comme le composant distant, les composants parallèles qui sont basés sur le super composant, le distributeur, le collecteur et le pipeline. L'introduction de la construction des applications distribuées/parallèles par le biais d'un framework aide le développeur à mieux structurer l'application et à réutiliser des composants existants. Mais l'utilisation du framework entraîne un surcoût à l'exécution qui est largement compensé par les facilités de conception apportées par l'utilisation des composants. Le surcoût d'exécution est évalué dans ce travail.

Mots-clés: traitement distribué, composants logiciels, CCA, Common Component Architecture, ADAJ, Desktop Grid, parallélisme, composants parallèles

A framework for distributed and parallel software components

Abstract. In this thesis we are interested in component based distributed and parallel applications. Our work concerns the construction of a component framework (CCADAJ) above the DG-ADAJ (Desktop Grid - Adaptive Distributed Applications in Java) platform. The DG-ADAJ platform provides object observation and placement and migration mechanisms. The framework CCADAJ is based on the CCA (Common Component Architecture) model. CCA promises application writing facility, execution efficiency and dynamic construction of parallel applications. The construction of CCADAJ consists of extending the CCA model to take into account the hierarchical composition, integrating the different services of the framework into the DG-ADAJ platform and building some special structures for distributed/parallel application construction. We introduce the super component model to be able to do hierarchical composition. Furthermore, we build some structures that are specific to distributed and parallel application development. Theses structures are the remote component, the parallel components which are based on the super component, the distributor, the collector and the pipeline components. Using a component framework for the component based construction of distributed/parallel applications helps the developer to build well structured applications and to reuse in better way existing components. But the execution of such applications has an overhead due to framework use. This overhead that we have evaluated is minimized relatively to advantages brought by component based development.

Keywords: distributed computing, software components, CCA, Common Component Architecture, ADAJ, Desktop Grid, parallelism, parallel copmonents