MADA, une approche pour le développement d'applications mobiles

Nabil Kouici, Nawel Sabri, Denis Conan et Guy Bernard

GET / INT, CNRS UMR SAMOVAR
9 rue Charles Fourier, 91011 Évry, France
{Nabil.Kouici, Nawel.Sabri, Denis.Conan, Guy.Bernard}@int-evry.fr

RÉSUMÉ

Malgré une forte évolution des équipements utilisés dans les environnements mobiles, les intergiciels utilisés restent inadéquats dans le cadre d'environnements où les ressources peuvent varier considérablement voir disparaître spontanément. La grande majorité des travaux qui traitent le problème des déconnexions ne proposent pas de modèle de conception d'applications devant fonctionner en présence des déconnexions. Dans cet article, nous présentons MADA, une approche de conception d'applications qui fonctionnent en présence des déconnexions et qui suit l'approche MDA (*Model Driven Architecture*) [23] : l'architecture logicielle est modélisée en utilisant le patron de conception « Façade » [10] et avec 4+1 vues [17], et ensuite projetée sur une plateforme, ici la plateforme CCM (*CORBA Component Model*) [25], via un profil UML (*Unified Modelling Language*).

Mots clefs

Mobilité, déconnexion, adaptation, composant

ABSTRACT

Despite the rapid evolution of computers in mobile environments, current middleware are inadequate in mobile environments where resources can vary a lot and even disappear. Most of research works dealing with disconnection management do not provide a design model for applications. In this article, we present MADA, an approach to design distributed applications for disconnections thats follows the *Model Driven Architecture* approach [23]: the software architecture is modelled using the design pattern "Façade" [10] and with 4+1 views [17], and then projected over a platform, here the CCM (*CORBA Component Model*) platform [25], with a UML (*Unified Modelling Language*) profile.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Mobilité & Ubiquité 2004 June 1-3, 2004, Nice, France
Copyright 2004 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Categories and Subject Descriptors

C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems; D.4.7 [**OPERATING SYSTEMS**]: Organization and Design

General Terms

Design

Keywords

Mobility, disconnection, adaptation, component

1. INTRODUCTION

Dans une étude sur la mobilité parue en décembre 2002 par IDC (http://www.idc.fr), plus de 4,5 millions de personnes en France sont mobiles au sens de travailler en dehors d'un lieu physique unique au sein de leur entreprise, et plus de 6 millions le sont à l'extérieur. Cette mobilité peut être locale, avec les réseaux Wi-Fi ou Bluetooth, comme elle peut être étendue, avec les réseaux GPRS ou UMTS. Ceci est dû à l'évolution des terminaux mobiles tels que les assistants personnels numériques ou les téléphones portables. Cette évolution a abouti au développement de l'informatique mobile dans laquelle l'utilisateur peut continuer à utiliser les services fournis par une infrastructure distribuée sans tenir compte de son emplacement.

Les principaux intergiciels qui existent aujourd'hui sont destinés à simplifier le développement d'applications distribuées, en particulier les plateformes CORBA [25] de l'OMG, EJB [9] de SUN, .NET [18] de Microsoft et Fractal [20] d'ObjectWeb. Malheureusement, ils sont inadéquats pour les environnements mobiles où les ressources (bande passante, batterie, mémoire...) peuvent varier considérablement voire disparaître spontanément. Ils sont destinés aux environnements traditionnels relativement statiques où les ressources sont disponibles et pratiquement stables.

Les déconnexions dans les environnements mobiles ne doivent pas être vues comme des défaillances puisqu'elles sont une conséquence de la mobilité des utilisateurs. Nous définissons trois modes de connectivité : le mode connecté qui correspond à une connexion sans fil normale, le mode partiellement connecté où le mobile ne dispose que d'une communication à faible débit, et enfin, le mode déconnecté où le mobile ne dispose plus de lien physique. Les déconnexions peuvent être volontaires car décidées par l'utilisateur depuis son terminal mobile (par exemple, pour économiser

l'énergie de la batterie), comme elles peuvent être involontaires (par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio).

L'adaptation du fonctionnement de l'application aux déconnexions peut être entièrement de la responsabilité de l'application (stratégie « laissez-faire »), de la responsabilité du système ou de l'intergiciel (stratégie « transparence »), et enfin, effectuée par collaboration entre l'application et le système (stratégie « collaboration ») [27]. De nombreux travaux synthétisés dans [13] montrent que les approches « laissez-faire » et « transparence » ne sont pas adéquates. Aussi, dans nos travaux, nous adoptons la stratégie « collaboration ». Cependant, les travaux cités ci-dessus ont traité le problème des déconnexions d'une manière *ad hoc* et aucun de ces travaux ne propose de modèle de conception d'applications devant fonctionner en présence des déconnexions.

L'objectif de cet article est de présenter une approche de conception d'applications qui fonctionnent en présence des déconnexions nommée MADA (*Mobile Application Development Approach*). Nous présentons aussi l'intégration de MADA dans l'approche MDA [23], la plateforme ciblée étant D⊛MINT [8, 15, 16] construite pour les composants CORBA [25].

La suite de cet article est divisée comme suit. La section 2 introduit nos motivations et objectifs. Ensuite, dans la section 3, nous présentons globalement l'approche MADA avant de développer la phase de conception de l'architecture logicielle dans la section 4, puis la conception détaillée pour la projection dans le monde CCM dans la section 5. La section 6 établit le lien avec les travaux existants. Enfin, la section 7 conclut l'article et donne quelques perspectives.

2. MOTIVATION

La disponibilité des services applicatifs dans les environnements mobiles soulève de nombreux problèmes, dont la réplication des données et du code sur le terminal mobile. Pour une meilleure adaptation fonctionnelle, les entités instanciées sur le terminal mobile ne sont pas exactement les mêmes que celles accédées à distance ; nous les appelons des entités déconnectées [8]. Elles sont utilisées pour assurer la continuité de service lors de déconnexions. Les opérations effectuées pendant les phases de déconnexion sont journalisées localement pour être envoyées lors de la reconnexion aux serveurs pour la réconciliation [6].

Comme tout aspect extrafonctionnel, la gestion des déconnexions doit offrir le maximum de transparence aux utilisateurs. Celle-ci passe par une adaptation aux changements de niveau de disponibilité des ressources (mémoire, bande passante, batterie...). D'une manière générale, l'adaptation est comprise comme étant « la capacité d'un système à s'ajuster aux changements de son environnement ou de sa structure interne » [3]. Cette adaptation peut être de trois types. Premièrement, l'adaptation « statique » intervient avant l'exécution, par exemple, en modifiant le code ou le déploiement de l'application. Donc, une connaissance *a priori* de l'environnement d'exécution de l'application est nécessaire et ne prend pas en compte le choix de l'utilisateur dans les politiques à utiliser pour la gestion des déconnexions. Deuxièmement, l'adaptation « dynamique » intervient au moment de l'exécution de l'application ; elle est réalisée par une intervention extérieure qui est la plupart du temps humaine. Donc, l'utilisateur doit connaître la structure de son application. Enfin, « l'auto-adaptation » s'effectue durant l'exécution ; elle est initiée par l'application elle-même ou par le système. Nous proposons d'effectuer l'adaptation en mixant l'adaptation statique et l'auto-adaptation. L'adaptation statique est utilisée dans le processus de développement de l'application. Nous définissons l'architecture logicielle et le fonctionnement de l'application suivant le mode de connectivité disponible. Pendant l'exécution, l'auto-adaptation consiste en la commutation transparente entre les différentes configurations de l'application définies dans l'adaptation statique. Cette commutation est pilotée par l'intergiciel qui, lui-aussi, doit s'adapter suivant les besoins de l'application.

Dans l'approche MDA [23], l'architecture logicielle de l'application est modélisée pour devenir le point de départ de la spécification de la solution. Nous choisissons le modèle 4+1 vues de l'architecture [17] car il utilise plusieurs vues concourantes. Cette utilisation de vues permet d'adresser séparément les intérêts des divers groupes d'intervenants (architectes, utilisateurs, développeurs, chefs de projet...), et de mieux séparer les aspects fonctionnels et extrafonctionnels. En outre, comme nous en montrons l'intérêt dans la section 3.2, le modèle 4+1 vues est « piloté » par les cas d'utilisation ou scénarios. Nous surchargeons les cas d'utilisation initiaux de l'application par des cas d'utilisation qui définissent le fonctionnement en présence des déconnexions.

La construction d'applications réparties converge de plus en plus vers l'utilisation d'intergiciels orientés composants. Le paradigme composant répond mieux au problème de la complexité de gestion des applications. Il couvre toutes les étapes du cycle de vie des applications et offre une séparation entre les aspects fonctionnels et extrafonctionnels. Cette séparation est réalisée suivant le patron de conception composant/conteneur [28] : le composant encapsule les aspects fonctionnels et le conteneur gère les aspects extrafonctionnels. Cependant, de nombreux architectes ont déjà mis en évidence que quel que soit l'intergiciel utilisé pour implanter des applications à base de composants, la logique métier reste la même. C'est suite à cette constatation que l'OMG propose MDA : définir une représentation abstraite de l'architecture logicielle indépendante[1] de toute architecture technique[2], tout en lui associant une multitude de services métiers. Dans notre travail, MDA nous permet de modéliser les différents aspects extrafonctionnels pour les prendre en compte dans l'architecture de l'application. Nous proposons dans cet article un profil UML pour la gestion des déconnexions pour les applications à base de composants CCM.

3. PRÉSENTATION GÉNÉRALE DE MADA

Les concepts que nous présentons dans cet article sont suffisamment généraux pour être utilisés avec d'autres modèles d'architecture que le modèle 4+1 vues, dans la mesure où ils séparent les aspects fonctionnels et extrafonctionnels, et utilisent un processus centré sur l'architecture logicielle et piloté par des cas d'utilisation. Mais avant d'introduire la conception dans MADA en section 3.4, nous décrivons d'abord l'application exemple utilisée dans cet article dans la section 3.1. Ensuite, nous présentons le modèle 4+1 dans

[1] Appelée un PIM (*Platform Independent Model*).
[2] Appelée un PSM (*Platform Specific Model*).

la section 3.2 et le patron de conception façade en section 3.3.

3.1 Application exemple

L'approche MADA est illustrée par une application qui modélise un service d'urgence médical lors d'une catastrophe (accident d'avion ou de train, tremblement de terre...). Cette application est développée dans le cadre du projet AMPROS [1]. Elle comporte plusieurs acteurs répartis sur le terrain et dans des structures fixes. Les patients sont couverts avec des couvertures appareillées avec des calculateurs embarqués enregistrant périodiquement des informations médicales (température, rythme cardiaque, rythme respiratoire...). Ces informations sont transmises via un réseau sans fil aux sauveteurs (secouristes, pompiers, ambulanciers, médecins...) équipés d'assistants personnels numériques. Ces sauveteurs sont éparpillés sur le lieu de l'accident et viennent au secours des patients. Le scénario inclut des ambulances « du futur » équipées de matériels médicaux pour des interventions plus complexes. Un centre de commandement détaché est déployé proche du lieu de l'accident afin de gérer les ambulances et les sauveteurs. Ce centre est déployé par un centre de commandement régional qui communique avec les hôpitaux.

3.2 Architecture modélisée en 4+1 vues

Le modèle 4+1 vues permet d'organiser la description d'une architecture logicielle en plusieurs vues concourantes, chacune adressant un point de vue différent. L'architecture logicielle modélise la structure de l'application à un niveau d'abstraction élevé.

La figure 1 schématise le modèle 4+1 vues. Ce modèle est composé de cinq vues. La vue « logique » décrit les aspects statiques et dynamiques d'un système en termes de classes, d'objets et de composants, de connexions et de collaborations. Elle se concentre sur l'abstraction et l'encapsulation. La vue « processus » capte les aspects de concurrence et de synchronisation de la conception, et les décompose en flots d'exécution (processus, file d'exécution...). Elle se rapporte aux objets actifs et aux interactions. La vue « développement » représente l'organisation statique des modules (exécutable, code sources, paquetages...) dans l'environnement de développement. La vue « physique » décrit les différentes ressources matérielles et l'implantation logicielle dans ces ressources. Donc, elle se rapporte aux nœuds du matériel et aux instances de composants. La dernière vue, la vue « cas d'utilisation » se concentre sur la cohérence en présentant des scénarios d'utilisation qui mettent en œuvre les éléments des quatre premières vues. Les scénarios sont une abstraction des exigences fonctionnelles ou services. Cette dernière vue valide en quelque sorte les autres vues.

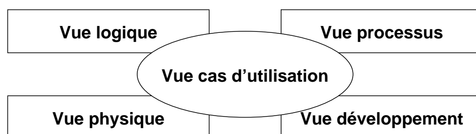

Figure 1: Le modèle 4+1 vues de l'architecture

3.3 Patron de conception façade

Le patron de conception « façade » [10] permet de fournir un accès unifié à un groupe de services (ou interfaces) d'un

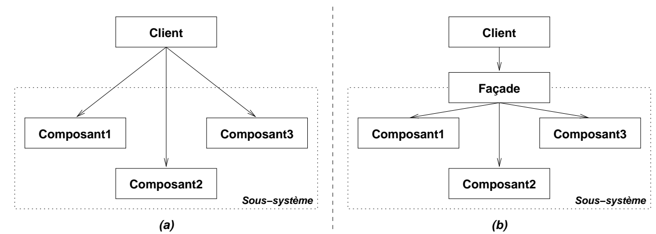

Figure 2: Application avec et sans façade

sous-système. Les façades réduisent également le nombre de composants présentés au client. Ce patron de conception rend donc le sous-système plus facile à utiliser et à remplacer.

Supposons que la figure 2-a représente une application répartie sans façade, et la figure 2-b représente la même application avec façade. Dans le cadre de l'application avec façade, c'est la façade que le client adresse ; dans la suite, c'est elle aussi qui choisira le mode de fonctionnement : par exemple, travail en local pendant les déconnexions. En outre, les opérations déduites des cas d'utilisation de l'architecture viennent naturellement constituer l'ensemble des opérations de l'interface de la façade, d'où la facilité de conception d'une interface graphique présentant les choix (prédéfinis) des opérations disponibles pendant les déconnexions. Enfin, l'ajout de nouvelles fonctionnalités ou services dans le sous-système à comme effet la surcharge de la façade puisque rien ne nous empêche d'ajouter des méthodes à la façade ou de modifier ces méthodes.

3.4 Présentation générale de MADA

Dans [15], nous avons présenté un patron de conception pour le partitionnement des composants de l'application en composants déconnectables et composants non déconnectables suivant une méta-donnée appelée *déconnectabilité*. Un composant déconnectable est un composant qui peut avoir des composants déconnectés utilisés pendant le mode déconnecté. Nous avons aussi proposé la méta-donnée *nécessité* qui donne un « poids » aux composants de l'application quant à leur présence dans le terminal mobile. Un composant nécessaire est un composant dont la présence dans le terminal mobile est obligatoire pour le fonctionnement en mode déconnecté. Un composant non nécessaire est un composant dont l'absence dans le terminal mobile ne peut pas empêcher le fonctionnement global de l'application en mode déconnecté. L'application doit donc être construite en conséquence.

Nous présentons dans cet article une approche complète centrée sur l'architecture logicielle et pilotée par les cas d'utilisation (c.-à-d., partant des fonctionnalités fournies aux utilisateurs finaux), s'intégrant dans l'approche MDA. L'approche MADA suit l'approche MDA comme l'indiquent les figures 3-a et 3-b, respectivement. Dans la figure 3-b, l'architecte de l'application répartie construit une modélisation UML de l'architecture logicielle (dans la figure, le PIM), tout en respectant le profil UML EDOC [22] qui cible les applications construites à base de composants. Ce modèle est indépendant de la plateforme d'implantation et d'exécution. Il est transformé en un modèle ayant pour cible le monde CCM et respectant le profil UML CCM [24]. Ne

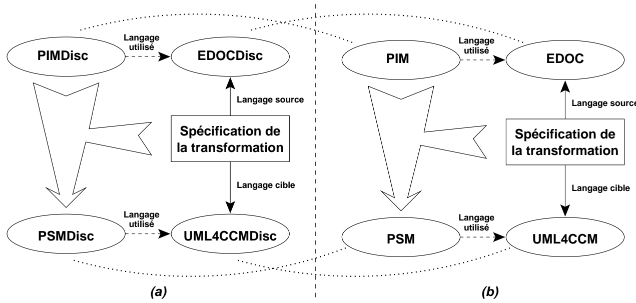

Figure 3: Vision MDA de MADA

désirant pas fournir un atelier de génie logiciel, nous considérons que cette transformation existe. Par analogie, dans la figure 3-a, l'architecture logicielle de l'application est surchargée de spécifications dédiées à la gestion de déconnexion (dans la figure, le PIMDisc) et suit le profil EDOCDisc, lui-même étendu à partir de EDOC. Enfin, la projection dans le monde CCM donne le modèle PSMDisc respectant le profil UML4CCMDisc. Par conséquent, dans la suite de l'article, l'approche MADA est présentée comme une surcharge de deux profils, le premier pour la spécification de l'architecture logicielle, le second pour le modèle cible de la transformation.

4. CONCEPTION AVEC 4+1 VUES

Les diverses vues du modèle 4+1 vues ne sont pas entièrement indépendantes. Des éléments d'une vue sont connectés aux autres éléments dans d'autres vues suivant certaines règles de conception. Par exemple, dans la vue logique, nous définissons les classes et les associations entres ces classes d'une manière abstraite et statique, et dans la vue de développement, la création des paquetages revient à mettre plusieurs classes dans un seul répertoire. Dans cette section, nous décrivons les différentes vues du modèle 4+1 vues de l'architecture dans le cadre de l'approche MADA.

4.1 Scénarios et cas d'utilisation

Les cas d'utilisation sont modélisés par un diagramme UML des cas d'utilisation. Ce diagramme montre les divers scénarios d'utilisation de l'application par des acteurs. À partir de ce diagramme et suivant le cahier des charges de l'application, les différents services exprimées dans les cas d'utilisation sont classés en services qui peuvent être effectués en mode déconnecté, appelés *déconnectables*, et en services qui ne peuvent pas être effectués en mode déconnecté. Ensuite, les services déconnectables vont être classés en services nécessaires et non nécessaires.

Dans le méta-modèle UML des cas d'utilisation, la relation `Extend` entre deux cas d'utilisation définit un héritage et permet de modéliser des variantes ou des comportements exceptionnels. La relation `Extend` comporte une condition booléenne à vérifier pour que le cas d'utilisation spécialisé soit choisi. Dans MADA, le comportement du système en mode déconnecté est modélisé par des cas d'utilisation qui héritent des cas d'utilisation de base. Chacun de ces cas d'utilisation est lié à une condition sur, par exemple, la bande passante du réseau sans fil et le niveau de la batterie. Pour modéliser les méta-données *déconnectabilité* et *nécessité*, le profil UML pour la gestion des déconnexions ajoute la relation `DiscExtend` et la valeur nommée `isNecessary` à la relation `DiscExtend`. `DiscExtend` hérite de la relation `Extend` et indique que le cas d'utilisation est déconnectable : le comportement du système en mode déconnecté est remplacé par le cas d'utilisation spécialisé. Le booléen `isNecessary` indique que le cas d'utilisation déconnectable est aussi nécessaire.

Dans notre application exemple, un sauveteur qui demande une expertise médicale, s'il est en mode connecté, discute avec un expert via une application collaborative, et s'il est déconnecté, accède à un composant local « système expert ».

4.2 Vue logique

Dans la vue logique, nous appliquons le patron de conception façade. Les cas d'utilisation déconnectables impliquent des opérations stéréotypées « déconnectables » : introduction du méta-élément `DiscOperation`. Par déduction, l'interface englobante est déclarée « déconnectable », ainsi que la façade : introduction des méta-éléments `DiscInterface` et `DisComponent`. Le diagramme de collaborations correspondant à cette opération met en œuvre des composants, qui peuvent devenir déconnectables, et qui fournissent des interfaces déconnectables incluant des opérations déconnectables. Bien sûr, l'objectif est de circonscrire la propagation de la déconnectabilité. Clairement, la première solution est de faire en sorte que le composant déconnectable « client » fonctionne sans le composant « serveur », quitte à fournir une forme dégradée du service. En conséquence, l'exécution pendant les déconnexions n'est pas « équivalente » à l'exécution sans déconnexions. Cela est acceptable pourvu que l'utilisateur final ait connaissance du mode de connectivité via un icône ou une interface graphique. En ce qui concerne la nécessité, le même cheminement est appliqué.

Dans notre exemple, la fonctionnalité d'expertise est déconnectable, mais pas celle du travail collaboratif.

4.3 Vue physique

Dans une application répartie « classique » fonctionnant avec une bonne connectivité entre les différents composants, les composants proches de l'utilisateur peuvent s'exécuter sur des terminaux légers et n'être constitués que d'une interface graphique, les autres composants restant sur des nœuds du réseau fixe. Pour y ajouter la continuité de service lors de déconnexions, des mandataires des composants distants doivent être instanciés sur les terminaux mobiles, les opérations pendant les phases de déconnexion journalisées et des réconciliations lors des reconnexions opérées. La continuité de service pendant les déconnexions est donc assurée par le chargement de composants mandataires appelés composants déconnectés [8] (cf. figure 4), similaires en conception et en implantation aux composants distants restant sur le réseau fixe, mais spécifiquement construits pour faire face au fonctionnement en mode déconnecté (cf. section 4.2). Il est de la responsabilité du concepteur de l'application d'opérer un compromis entre une conception simple et une conception plus complexe qui s'adapte plus efficacement aux variations de connectivité. En outre, les composants sont partageables par toutes les applications s'exécutant sur le terminal mobile.

Dans l'application exemple, le composant « système ex-

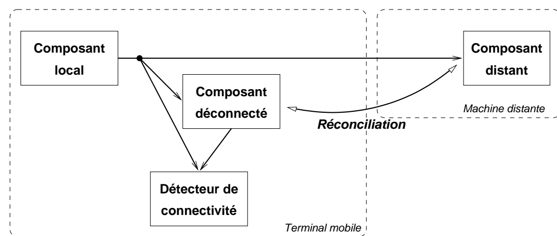

Figure 4: Composant déconnecté

pert » est déconnectable, et même nécessaire, et le composant « *chat* » pour le travail collaboratif ne l'est pas. Il est aisé d'imaginer que le système expert consulte et modifie des données, rassemblées sous la forme d'un dossier médical : donc, du code et des données sont à charger.

4.4 Vue processus

Comme indiqué dans les deux sections précédentes 4.2 et 4.3, même si le support de l'intergiciel sous-jacent doit être le plus large possible, MADA permet d'exprimer que l'utilisateur doit aussi adapté son comportement pendant le mode déconnecté. Ces adaptations sont spécifiées dans des diagrammes d'activités. L'interface graphique indique à l'utilisateur les changements de mode de connectivité. Dans notre cas, le diagramme d'activités représente la machine à état [3] de la procédure que l'utilisateur final doit suivre, s'il en existe une bien sûr. Les événements proviennent soit de la plateforme, par exemple le détecteur de connectivité de la figure 4, et doivent être pris en considération par l'utilisateur, soit de l'utilisateur lui-même, par exemple une déconnexion volontaire, et c'est la plateforme qui doit s'adapter.

Dans notre application exemple, le sauveteur accède via la façade à l'expertise et au travail collaboratif. Par défaut, il discute avec l'expert. Lorsqu'il est déconnecté, la demande de travail collaboratif n'est pas acceptée, un message l'avertit, et il est redirigé automatiquement vers le système expert.

4.5 Vue développement

Dans MADA, cette vue consiste d'abord au développement des composants d'origine et de leurs composants déconnectés. Ensuite, les paquetages comportent en plus du code des composants d'origine le code des composants déconnectés susceptibles d'être déployés sur les terminaux mobiles. Le composant d'origine et ses composants déconnectés doivent comporter des interfaces conçues pour la capture, le transfert et la mise à jour de l'état du composant.

Dans l'exemple, la façade et le composant déconnecté « système expert » sont empaquetés ensemble.

5. CONCEPTION DÉTAILLÉE AVEC CCM

Cette section présente un survol du modèle de composant CCM dans la section 5.1 ainsi que le profil UML pour les applications CCM supportant les déconnexions dans la section 5.3. Ce profil est le résultat de l'intégration des notions de déconnectabilité et de nécessité au profil UML pour CCM en cours d'adoption à l'OMG (cf. section 5.2).

[3] C'est un diagramme d'activités plutôt qu'un diagramme de transitions d'états car plusieurs classes UML sont concernées.

5.1 Composant CORBA

Le choix de CCM est justifié par plusieurs points. Premièrement, par rapport aux autres modèles tels que EJB, COM et .NET, CCM peut être vu comme l'union de ces modèles. Comme dans EJB, un composant CCM est créé et géré par une maison, et exécuté dans un conteneur. Comme dans COM, un composant CORBA offre et utilise des interfaces, et permet la navigation et l'introspection. Comme dans .NET, un composant CORBA peut être écrit dans plusieurs langages de programmation et peut être empaqueté pour être distribué. Deuxièmement, CCM est multi-langage, multiOS, multiORB, multivendeur, contrairement à EJB qui est purement JAVA, et COM et .NET qui sont purement Windows. Enfin, CCM dispose d'un profil UML qui permet de décrire des composants conformes au modèle de composants CCM. Troisièmement, le modèle de composant Fractal, plus complet que CCM, ne bénéficie pas encore d'un méta-modèle ni d'un profil UML permettant de l'intégrer aisément dans une approche MDA.

Un composant CCM est caractérisé par des ports classés en synchrones (facettes et réceptacles) et asynchrones (sources et puits d'événements), et des attributs pour la configuration (cf. figure 5). En outre, le composant CCM peut être segmenté, et pour chaque segment, CCM génère un squelette. Les segments sont activés indépendamment les uns des autres et possèdent un état. Enfin, chaque segment peut être identifié séparément dans le système.

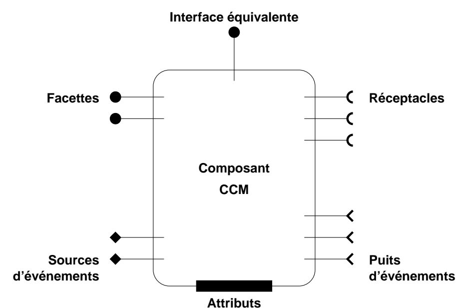

Figure 5: Architecture abstraite d'un composant CCM

5.2 Profil UML pour CCM

Un profil UML existe pour la modélisation des applications à base de composants CORBA [24]. Nous l'appelons *UML4CCM*. Ce profil est une spécialisation du profil pour CORBA [21] permettant de modéliser les applications à base d'objets CORBA. Le profil UML4CCM décrit tous les concepts nécessaires à la modélisation d'une application CCM, à savoir la modélisation IDL et la modélisation CIDL. Ces deux modélisations définissent deux vues complémentaires de l'application. *La vue « client »* spécifie dans le langage IDL (*Interface Definition Language*) les caractéristiques des composants de l'application en terme de ports (facettes, réceptacles...), de maison et d'attributs. Autrement dit, la vue client définit les composants tels qu'ils sont vus et utilisés par les composants clients. *La vue « implantation »* décrit dans le langage CIDL (*Component Implementation Description Language*) la structure interne des composants ainsi que la persistance, si le composant est persistant. Cette structure consiste essentiellement en l'ensemble des

segments de ce composant. Autrement dit, la vue implantation définit les composants tels qu'ils sont implantés.

La spécification de CCM formalise les différents concepts relatifs aux spécifications IDL et CIDL en définissant pour chacune un méta-modèle. Le méta-modèle de l'IDL représente les concepts de composant, de facette, de réceptacle, de puits et source d'événements..., ainsi que les relations qui existent entre eux. Le méta-modèle du CIDL regroupe essentiellement les concepts relatifs à l'implantation du composant et de ses segments. Chaque élément de modélisation du profil UML4CCM est la représentation graphique d'un concept spécifié dans le méta-modèle. Le profil UML4CCM consiste en un ensemble de stéréotypes, de règles et de valeurs nommées non détaillés ici par souci de concision, mais dont nous citons quelques éléments dans la section suivante.

5.3 Profil UML pour la déconnexion

Nous introduisons une nouvelle vue : la *vue mode déconnecté* dont la spécification étend le processus de développement d'une application CCM standard. Pour cela, nous étendons le profil UML4CCM. Conformément à la démarche de CCM, nous présentons successivement le méta-modèle du nouveau profil (*UML4CCMDisc*) puis le profil UML lui-même.

La vue mode déconnectée est une description des composants ici appelés des *composants déconnectés* (cf. figure 6). Un composant déconnecté est représenté par le méta-élément `CCMDisComponent`. Conceptuellement, c'est un composant CORBA étendu, d'où le lien de généralisation avec le méta-élément CCM `ComponentDef`. En fait, il surcharge le composant CORBA d'une sémantique spécifique pour le mode déconnecté. En outre, un composant déconnecté représente la version déconnectée d'un composant standard de la vue client, d'où le lien d'association avec le méta-élément `ComponentDef`.

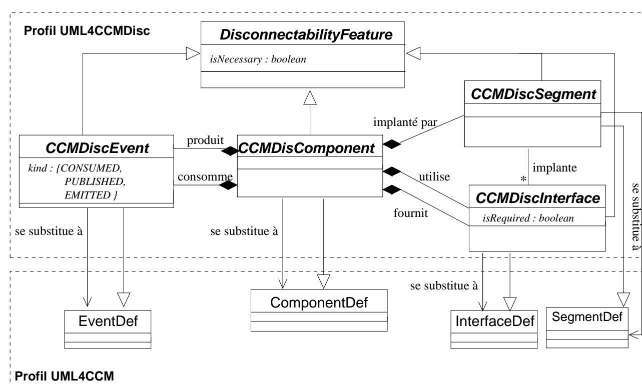

Figure 6: Méta-modèle du profil UML4CCMDisc

Un composant déconnecté fournit et requiert un ensemble d'interfaces déconnectées désignées par le méta-élément `CCMDiscInterface`. Une interface déconnectée est une interface CORBA normale, d'où le lien de généralisation avec le méta-élément `InterfaceDef`. L'attribut booléen `isRequired` indique que l'interface est fournie ou requise. Les interfaces d'un composant déconnecté correspondent à tout ou partie des interfaces fournies de son composant associé de la vue client. Une interface déconnectée doit être associée à l'interface du composant associé à laquelle elle se substitue en cas de déconnexion, d'où le lien d'association avec le méta-élément `InterfaceDef`.

Un composant déconnecté communique de manière asynchrone en produisant et consommant des événements. Ces événements sont désignés par le méta-élément `CCMDiscEvent`. Ce type d'événements dérive des événements définis dans le modèle CCM, d'où le lien de généralisation avec le méta-élément `EventDef`. Comme pour les interfaces déconnectées, les événements déconnectées doivent être associés aux événements du mode connecté auxquels ils se substituent, d'où le lien d'association avec `EventDef`. Afin de garder les mêmes catégories de ports asynchrones que dans CCM, l'attribut `kind` indique à travers ses trois valeurs si l'événement est consommé, diffusé (vers un groupe de consommateurs) ou émis (vers un seul consommateur).

Un composant déconnecté possède également une implantation exprimée par le méta-élément `CCMDiscSegment`. Comme pour les composants CCM, une implantation déconnectée peut être répartie sur plusieurs segments (d'où le lien de généralisation avec le méta-élément `SegmentDef`). Chaque segment réalise une ou plusieurs interfaces fournies, dont certaines sont des interfaces déconnectées, d'où le lien d'association avec le méta-élément `CCMDiscInterface`.

Nous modélisons la propriété de déconnectabilité par le méta-élément `DisconnectabilityFeature`. La propriété de déconnectabilité exprime aussi la propriété de nécessité à travers l'attribut booléen `isNecessary`. Il n'était pas nécessaire de définir une entité à part pour la nécessité parce qu'elle ne peut être vérifiée que si la déconnectabilité l'est aussi. Ces propriétés s'appliquent sur le composant et ses constituants.

Nous passons maintenant à la définition du profil UML4CCMDisc lui-même. Ce profil est constitué d'un ensemble de règles et de stéréotypes dont nous définissons la correspondance avec les éléments de conception UML et avec les éléments du méta-modèle du profil UML4CCMDisc. Le lien avec les éléments UML sert à définir la modélisation graphique de la vue mode déconnecté de l'application alors que le lien avec les éléments du méta-modèle sert à interpréter et vérifier la validité du modèle graphique.

Tel que présenté dans le tableau 1 , le profil UML4CCMDisc contient quatre stéréotypes représentant respectivement un composant déconnecté, ces interfaces, ces événements et son implantation. Ces quatre éléments correspondent à des classes UML stéréotypées que le concepteur de l'application pourra utiliser pour définir la vue mode déconnecté de son application. Chacun de ces stéréotypes possède une valeur nommée `isNecessary` qui exprime la nécessité. Les stéréotypes `CCMDiscInterface` et `CCMDiscEvent` possèdent respectivement les valeurs nommées `isRequired` et `kind` telles qu'elles ont été définies par le méta-modèle.

Enfin, des règles régissent la déconnectabilité et la nécessité des composants et de leurs éléments. Elles servent à vérifier la cohérence de la vue mode déconnecté. Un composant de la vue client est déconnectable si, et seulement si, un composant de la vue mode déconnecté lui est associé. Un composant déconnecté est nécessaire si, et seulement si, au moins une interface ou un événement sont nécessaires. Si une interface déconnectée fournie est nécessaire, le segment correspondant est nécessaire. Un composant nécessaire est déconnectable. Une interface (*resp.* un événement, un segment) nécessaire implique que le composant correspondant

Stéréotype	Méta-élément	Élément UML	Valeur nommée	Parent
CCMDiscomponent	CCMDiscomponent	Class	isNecessary	ComponentDef
CCMDiscInterface	CCMDiscInterface	Class	isNecessary, isRequired	InterfaceDef
CCMDiscEvent	CCMDiscEvent	Class	isNecessary, kind	EventDef
CCMDiscSegment	CCMDiscSegment	Class	isNecessary	SegmentDef

Table 1: Profil UML4CCMDisc

soit déconnectable. La spécification de la vue mode déconnectée sert à fournir les informations nécessaires à la configuration et/ou génération des parties de l'intergiciel qui collaborent pour la gestion des déconnexions [15].

6. TRAVAUX CONNEXES

Un panorama est donné dans [13] sur l'adaptation à la mobilité des applications client/serveur, qui se concentre de plus en plus sur la gestion et le déploiement[4] des entités déconnectées dans le terminal mobile. Coda [27] est un système de gestion de fichiers qui définit la notion de données implicites et de données explicites. Les données implicites représentent l'historique d'utilisation du client. Les données explicites prennent la forme d'une base de données construite par le client de l'application. Si l'utilisateur fait un mauvais choix pour ses données explicites suite à une mauvaise compréhension de l'application, cette dernière peut ne pas fonctionner en mode déconnecté. Odyssey [19] ajoute pour chaque objet distant de l'application deux méta-données : « fidélité » et « fenêtre de tolérance ». Ces méta-données sont bien adaptées pour la gestion de la faible connectivité, mais pas pour les déconnexions. Rover [14] définit les notions d'objet dynamique relogeable (*Relocatable Dynamic Object*) et d'appel de procédure à distance non-bloquant (*Queued Remote Procedure Call*). Rover traite tous les objets de l'application de la même façon et ne tient pas compte de la sémantique de l'application.

La decomposition de la description d'une architecture en plusieurs vues existe déjà dans le modèle RM-ODP [12]. RM-ODP définit une démarche de construction d'applications réparties basée sur la spécification de cinq points de vues architecturaux : entreprise, information, traitements, ingénierie et technologie. Cependant, contrairement à l'approche 4+1 vues, RM-ODP n'est pas une méthodologie de conception mais plutôt un ensemble de concepts. Ce modèle ne fournit pas d'outils, ni de règles d'usage ou de notations qui mettraient en œuvre ces concepts et guideraient les architectes dans la modélisation de leurs applications.

L'usage des méta-données a été aussi introduit dans les intergiciels réflexifs [2]. XMIDDLE [4] définit la méta-donnée « profil » qui décrit ce que l'intergiciel doit faire lorsqu'il se trouve dans un contexte particulier. Cette méta-donnée contient des informations sur les ressources externes (bande passante, batterie...) et ne prend pas en compte la sémantique de l'application. Comme dans CARISMA [5], dans D⊛MINT, la réalisation en cours associe à chaque application un fichier de description de comportements.

Dans le contexte CORBA, les projets Π^2 [26] et Alice [11] traitent le problème des courtes déconnexions provoquées par le changement de cellule dans *wireless* CORBA. Π^2 ne traite que le problème des déconnexions involontaires et il utilise deux mandataires, un sur le terminal mobile et un autre sur le terminal fixe. CASCADE [7] définit un service de gestion des mandataires générique et hiérarchique pour des objets CORBA. Il définit un service de désignation pour les serveurs de mandataires et les clients utilisent ce service pour trouver des serveurs où créer des mandataires. Dans CASCADE, si le client se déconnecte avec tous les serveurs comportant les mandataires, il ne peut pas continuer à travailler si sa machine n'est pas elle-même un serveur de mandataires.

[4]Ici, le mot déploiement signifie le téléchargement ou la création des mandataires sur le terminal mobile.

7. CONCLUSION

Dans cet article, nous avons présenté MADA, une approche d'adaptation des applications réparties en environnements mobiles pour la gestion des déconnexions. MADA suit l'approche MDA de l'OMG au sens où elle est orientée modèle UML, centrée sur l'architecture logicielle qui est exprimée selon deux formes : la première est indépendante de la plateforme, la seconde, dépendante de la plateforme, est obtenue par transformation de spécification. Nous ajoutons à la démarche de conception d'architecture logicielle des éléments dédiés à la gestion des déconnexions. Tout d'abord, l'architecture est décomposée en 4+1 vues. La vue « +1 » contenant les cas d'utilisation de l'application, pilote la conception des autres vues et a ainsi pour rôle d'assurer la cohérence globale. Ensuite, la stratégie choisie pour la gestion des déconnexions étant la collaboration entre l'application et l'intergiciel, nous montrons comment exprimer la gestion de la déconnexion dès l'architecture. En outre, le patron de conception façade permet de simplifier et de mieux contrôler la spécification du comportement de l'application lors d'une déconnexion. Enfin, pour la partie dépendante de la plateforme, dans notre cas CCM, nous étendons le profil UML de CCM avec les éléments introduits dans l'architecture logicielle. En fait, historiquement, les concepts de composants déconnectés, de déconnectabilité et de nécessité ont été mis en exergue avant les éléments décrits dans cet article [8, 15, 16].

MADA est un préambule à la plateforme D⊛MINT en cours de réalisation. Les perspectives de nos travaux sont donc les suivantes. Tout d'abord, nous allons terminer l'implantation de D⊛MINT, sans la partie transformation de spécification « à la » MDA. D'autre part, les méta-données que nous avons présentées dans cet article ne s'occupe pas du problème de la priorité entre les différentes entités de l'application et de politique remplacement de cache. Cette priorité doit être spécifiée dans une autre méta-donnée. En outre, les méta-données sont spécifiées « statiquement », il serait intéressant d'étudier la possibilité de les modifier pendant l'exécution. Enfin, la démarche est applicable à une autre organisation (en vues) de l'architecture et à un autre modèle de composant spécifié avec un méta-modèle décrit

dans un profil UML. Une dernière piste que nous n'avons pas encore explorée est l'approche à AOD/AOP (*Aspect Oriented Design/Aspect Oriented Programming*).

8. RÉFÉRENCES

[1] AMPROS Home Page. `http://www-inf.int-evry/AMPROS`, 2004.

[2] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, England, 1998.

[3] E. Bruneton. *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties.* PhD thesis, INPG, 2001.

[4] L. Capra, W. Emmerich, and C. Mascolo. Exploiting Reflection and Metadata to build Mobile Computing Middleware. In *Workshop on Middleware for Mobile Computing*, Heildelberg, Germany, Nov. 2001.

[5] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, Oct. 2003.

[6] L. Chateigner, S. Chabridon, and G. Bernard. Intergiciel pour l'informatique nomade : réplication optimiste et réconciliation. In *MAJECSTIC*, Oct. 2003.

[7] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing Caching Service for Distributed CORBA Objects. In J. Sventek and G. Coulson, editors, *Proc. ACM/IFIP Middleware*, volume 1795 of *Lecture Notes in Computer Science*, Apr. 2000.

[8] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Domint : une plate-forme pour la faible connectivité et la déconnexion d'objets CORBA en environnement mobile. Rapport de recherche, INT, Évry, France, May 2003.

[9] L. DeMichiel. *Enterprise JavaBeans Specifications, version 2.1, proposed final draft.* Sun Microsystems, http://java.sun.com/products/ejb/docs.html, Aug. 2002.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[11] M. Haahr, R. Cunningham, and V. Cahill. Supporting CORBA Applications in a Mobile Environment. In *Proc. 5th ACM SOSP*, pages 36–47, Seattle Washington, USA, Aug. 1999.

[12] ISO/IEC et ITU-T. Information Technology, Open Distributed Processsing: Reference Model. ISO/IEC specification 10746-1, 2, 3, 4, 1996–1998.

[13] J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):117–157, June 1999.

[14] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proc. 15thACM Symposium on Operating Systems Principles*, pages 156–171, Dec. 1995.

[15] N. Kouici, D. Conan, and G. Bernard. Disconnected Metadata for Distributed Applications in Mobile Environments. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2003.

[16] N. Kouici, D. Conan, and G. Bernard. Intégration d'un service de gestion de déconnexions dans les conteneurs des composants. In *Actes des Journées Composants*, Lille, France, Mar. 2004.

[17] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

[18] Microsoft Corporation. MSDN. http://www.msdn.microsoft.com, 2003.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16thACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, Oct. 1997.

[20] ObjectWeb Consortium. Fractal Home Page. http://fractal.objectweb.org, 2004.

[21] OMG. UML Profile for CORBA Specification, Version 1.0. Omg document, Object Management Group, Apr. 2002.

[22] OMG. UML Profile for Enterprise Distributed Object Computing Specification. ptc/02-02-05, Object Management Group, Feb. 2002.

[23] OMG. MDA Guide Version 1.0. omg/2003-05-01, Object Management Group, May 2003.

[24] OMG. UML Profile for CORBA Components RFP. Omg document, Object Management Group, May 2003.

[25] OMG. CORBA Components. OMG Document formal/02-06-65, Version 3.0, Object Management Group, Juin 2002.

[26] R. Ruggaber, J. Seitz, and M. Knapp. π^2 - A Generic Proxy Platform for Wireless Access and Mobility. In *Proc. 19th ACM PODC*, Portland, Oregon, July 2000.

[27] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc. 15thACM Symposium on Operating Systems Principles*, Philadelphia, USA, May 1996.

[28] C. Szyperski, D. Gruntz, and S. Murer. *Component Software, Beyond Object-Oriented Programming.* ACM-Press, Addison-Wesley, 2002.