Intégration d'un service de gestion de déconnexions dans les conteneurs des composants

Nabil Kouici — Denis Conan — Guy Bernard

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

RÉSUMÉ. *Ces dernières années ont été marquées par une forte évolution des équipements utilisés dans les environnements mobiles. L'accès aux applications réparties à partir de ces équipements soulève le problème de la disponibilité de ces applications en présence de déconnexions. La construction d'applications réparties converge de plus en plus vers l'utilisation des intergiciels orientés composants pour gérer la complexité des applications. Le modèle orienté composant offre une séparation entre les aspects fonctionnels et extrafonctionnels. Cette séparation est réalisée suivant le patron de conception composant/conteneur. Cependant, l'aspect déconnexion n'est que rarement considéré. Dans ce papier, nous présentons* D⊛MINT*, une architecture pour la gestion des déconnexions pour des applications à base de composants.* D⊛MINT *propose une structuration des conteneurs des composants et des services de l'intergiciel pour la gestion des déconnexions.*

ABSTRACT. *Last years have seen a strong evolution of small devices used in mobile environments. The access to the distributed applications raises the problem of service availability in the presence of disconnections. The design of distributed applications converges towards the use of the component-oriented middleware that manages applications' complexity. The component-oriented model offers a separation between functional and extra functional aspects. This separation is realized thanks to the component/container design pattern. However, disconnections management is rarely considered in component-oriented middleware. In this paper, we present* D⊛MINT*, a platform to cope with disconnections in mobile environments for component-based applications.* D⊛MINT *proposes structuring of container and middleware services for the disconnections management of component-based applications.*

MOTS-CLÉS : *Composant, conteneur, mobilité, déconnexion, adaptabilité.*

KEYWORDS: *Component, container, mobility, disconnection, adaptability.*

1. Introduction

Depuis le début des années 90, une forte évolution des terminaux mobiles tels que les assistants personnels numériques ou les téléphones portables est apparue. Cette évolution est en particulier une conséquence du développement des réseaux sans fil. Cette évolution a abouti à la définition d'une nouvelle technologie : l'informatique mobile. Selon 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 GRPS ou UMTS.

L'informatique mobile se caractérise par le nomadisme des utilisateurs. Or, les environnements d'exécution d'applications réparties qui se basent sur les intergiciels (*middleware*) actuels reposent sur des hypothèses (utilisateurs fixes, terminaux puissants, connexions aux réseaux de bonne qualité et peu coûteuses) incompatibles avec les caractéristiques de l'informatique mobile. La variation de la connectivité dans les environnements mobiles ne doit pas être vue comme une faute puisque cette variation est 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'un lien à faible débit, et enfin, le mode déconnecté où le mobile ne dispose plus de lien physique.

La construction d'applications réparties converge de plus en plus vers l'utilisation des intergiciels orientés composants. Le paradigme composant répond au problème de la complexité de gestion des applications. Il considère toutes les étapes du cycle de vie des applications et offre une séparation entre les aspects fonctionnels de l'application (le code métier) et les aspects extrafonctionnels (sécurité, transaction, cycle de vie…). Plusieurs modèles de composants existent actuellement, en particulier EJB [DEM 02] de SUN, CCM [Obj 02] de l'OMG, .Net [Mic 03] de Microsoft et Fractal [COU 03] de ObjectWeb.

Le travail présenté dans cet article s'inscrit dans la continuité de Domint [CON 02]. L'idée principale de Domint est de créer sur le terminal mobile des objets mandataires appelés « objets déconnectés ». La mobilité des applications se traduit par la mobilité des terminaux et non pas par la mobilité des entités qui forment l'application. La contribution principale des travaux présentés dans cet article est la proposition d'une architecture orientée composants nommée D⊛MINT pour la gestion des déconnexions. Nous proposons une structuration du conteneur pour intégrer un service de gestion des déconnexions.

Cet article est structuré comme suit. La section 2 identifie nos motivations et objectifs. Ensuite, dans la section 3, nous présentons l'architecture logique d'un composant et l'interaction de ce dernier avec le conteneur. La section 4 décrit l'architecture de D⊛MINT. La section 5 donne l'état d'avancement de notre implantation. La section 6 établit le lien avec les travaux existants. Enfin, la section 7 conclut l'article et donne quelques perspectives.

2. Motivations

Dans une application répartie mobile, la distribution des différentes entités de l'application peut se faire soit dans des terminaux fixes, soit dans des terminaux fixes et des terminaux mobiles [TER 95]. Donc, un terminal mobile peut être un client pour des serveurs de l'application et peut être un serveur pour d'autres terminaux (mobiles ou fixes). Ce dernier cas était rarement étudié dans des applications mobiles pour deux raisons : la capacité limitée des terminaux mobiles (mémoire, CPU, batterie...), et la difficulté de mise en œuvre de ces applications avec les modèles de programmation traditionnels. Avec la généralisation du paradigme composant, plusieurs facilités sont apparues dans le but de généraliser l'utilisation des terminaux mobiles. En particulier, le paradigme composant offre la possibilité d'avoir des entités (composants) qui offrent et utilisent des services. Donc, le composant joue le rôle de client et de serveur.

Les intergiciels orientés objets tels que CORBA et COM fournissent un ensemble de services qui facilitent la gestion et le développement des applications. Ces services représentent la partie extrafonctionnelle de l'application. Cependant, le code nécessaire à leur manipulation est fortement couplé avec le code métier de l'application. En conséquence, l'intégration de ces services reste une chose très compliquée à mettre en œuvre par le développeur de l'application. De ce fait, la réutilisation du code métier s'avère difficile. Or, la réutilisation du code permettrait de raccourcir le développement d'une application. Par conséquent, la séparation des aspects fonctionnels et extrafonctionnels s'avère nécessaire [KIC 97]. Dans les intergiciels orientés composants, le composant encapsule le code métier et le conteneur prend en charge la gestion des services extrafonctionnels d'une manière transparente aux composants. Donc, une réutilisation du composant dans différents contextes est facilitée.

Dans les intergiciels orientés composants tels que CCM, EJB et .Net, la gestion des aspects extrafonctionnels est soumise à de nombreuses limitations. Alors que les aspects fonctionnels peuvent être modifiés en toute liberté, les aspects extrafonctionnels sont généralement limités en ce qui concerne leur nombre, leur type et leur interaction. De ce fait, le choix des services à utiliser est impossible, puisqu'ils sont figés lors du développement du modèle. De plus, ces composants ne peuvent pas s'adapter à leur contexte en modifiant les services extrafonctionnels au cours de l'exécution. Les aspects extrafonctionnels les plus utilisés sont la persistance, la gestion des transactions, la sécurité et la distribution. L'aspect de gestion de déconnexions, qui est l'objet de cet article, n'est que rarement considéré. Dans cet article, nous montrons que le principe de séparation des aspects fonctionnels et extrafonctionnels convient aussi au problème de gestion de déconnexions. Ce principe permet l'adaptation et la reconfiguration de l'aspect gestion de déconnexions.

3. Structure et architecture d'un composant

Le patron de conception composant/conteneur [VOL 01] concerne non seulement le développement du code métier de l'application, mais aussi l'administration et la

gestion des aspects extrafonctionnels. Dans cette section, nous présentons la structure logique d'un composant indépendamment des modèles de composants existants.

Un composant est défini comme un morceau de logiciel assez petit pour que l'on puisse le créer et le maintenir, et assez grand pour que l'on puisse l'installer et le réutiliser. De plus, il est doté d'interfaces standards pour pouvoir interopérer avec d'autres composants. Il encapsule un état et exhibe un comportement à travers ses interfaces. La construction d'applications se fait par composition. Cette composition correspond à l'assemblage des composants. Il existe deux types de modèles de composants. Les modèles « plats » tels que CCM, EJB et .Net assemblent des composants simples, non composés, appelés composants primitifs. Les modèles « hiérarchiques » tels que Fractal assemblent des composants constitués de sous-composants, respectivement appelés composants composites et sous-composants. Dans la suite de cet article, nous présentons notre architecture dans le contexte des modèles plats.

L'architecture logique d'un composant est illustrée dans la figure 1-a. Cette architecture respecte le patron de conception décrit dans [VOL 01]. La structure logique d'un composant est caractérisée par deux parties. La première partie est fonctionnelle, elle représente le code métier du composant. Généralement, cette partie est appelée le *contenu* ou composant. La deuxième partie est extrafonctionnelle, gérée par le conteneur qui s'occupe de la gestion du cycle de vie des composants et des connexions inter-composants. La communication entre le conteneur et le composant se fait à travers des interfaces. Des interfaces internes offertes par le conteneur et utilisées par le développeur du composant aident à l'implantation du comportement du composant. Des interfaces de rappel offertes par le composant et utilisées par le conteneur sont disponibles pour la configuration des aspects extrafonctionnels. Le conteneur contrôle toutes les interactions des composants avec le monde extérieur. Ce contrôle est réalisé à travers des entités appelées les *objets de contrôle*. Dans la plupart des modèles de composants, le conteneur offre au minimum deux objets de contrôle, un *intercepteur des appels entrants* et un *intercepteur des appels sortants*. L'intercepteur des appels entrants est un objet de contrôle externe (exporté par le conteneur) ; il est visible de l'extérieur du composant et offre les mêmes interfaces que le composant. L'intercepteur des appels sortants est un objet de contrôle interne ; il n'est visible que de l'intérieur du conteneur (local). Le conteneur peut contenir d'autres objets de contrôle. Chaque objet de contrôle représente un accès vers un service extrafonctionnel de l'intergiciel ou bien réalise un service que le conteneur utilise dans la gestion des composants. Ces services peuvent être interrogés lorsque le composant reçoit ou émet des invocations.

La notion d'objet de contrôle est largement utilisée dans les différents modèles. Dans EJB, le conteneur comporte des objets d'interception qui interceptent les invocations sur les composants et gèrent les aspects extrafonctionnels au moyen de traitements avant et après ces invocations. Dans CCM, en plus des objets d'interception, le conteneur offre deux autres objets de contrôle[1] : un premier pour l'introspection et

1. CCM utilise l'expression « objets d'interposition » pour les objets de contrôle.

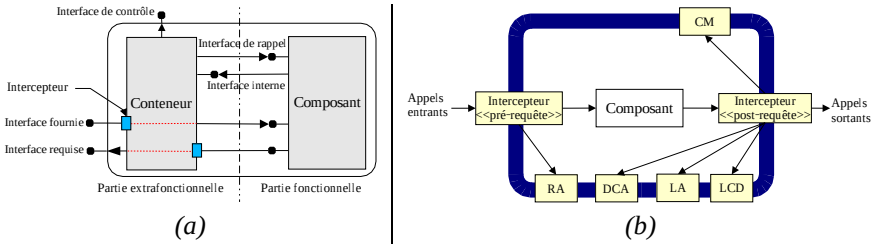

Figure 1. (a) *Structure logique d'un composant.* (b) *Architecture du conteneur pour la gestion des déconnexions.*

un second pour la gestion des ports. Le contrôleur d'introspection fournit des informations sur le type de composant, sur ses interfaces fournies et requises ainsi que sur l'état de ses connexions avec d'autres composants. Le contrôleur de gestion de ports est utilisé pour établir ou détruire des connexions entre composants au déploiement ou durant l'exécution. Dans Fractal, le conteneur est appelé *contrôleur*, il offre des objets de contrôle pour l'interception des appels entrants et sortants. Fractal définit une architecture flexible du conteneur qui permet de définir d'autres objets de contrôle suivant les besoins d'utilisation. Dans Julia[2][Obj 03], les objets de contrôle peuvent être optimisés et désoptimisés dynamiquement.

4. Architecture de D⊛MINT

Dans cette section, nous présentons d'abord l'architecture du conteneur que nous proposons pour la gestion des déconnexions (Cf. section 4.1). Ensuite, nous décrivons les services de l'intergiciel offerts par D⊛MINT (Cf. section 4.2). Enfin, nous montrons le fonctionnement de notre architecture avec les principales interactions (Cf. section 4.3).

4.1. *Architecture du conteneur*

Pour la gestion des déconnexions, nous proposons une architecture des conteneurs pour des applications mobiles où les hôtes peuvent être des clients et des serveurs pour d'autres hôtes (Cf. section 2). L'architecture de notre conteneur est illustrée dans la figure 1-b. Cette architecture est basée sur le modèle présenté dans la section 3.

La continuité de service est assurée par la création des composants déconnectés dans le terminal mobile. La solution consiste à utiliser ces composants déconnectés en mode partiellement connecté ou déconnecté, à journaliser les opérations exécutées localement et à réconcilier les composants déconnectés et les composants distants lors

2. Julia est une implantation de référence du modèle Java typé de Fractal.

des reconnexions. Dans ce papier, nous ne traitons pas le problème de la gestion de cohérence entre les différentes répliques du composant, ce problème est étudié dans [CHA 03].

Comme dessiné dans la figure 1-b, nous définissons cinq objets de contrôle dans le conteneur. Un détecteur de connectivité local (*Local Connectivity Detector*, LCD), un contrôleur d'accès au service de gestion des composants déconnectés (*Disconnected Component Access*, DCA), un contrôleur d'accès au service de journalisation (*Log Access*, LA), un gestionnaire de connecteurs (*Connector Manager*, CM) et un contrôleur d'accès au service de réconciliation (*Reconciliation Access*, RA).

Toutes les requêtes sortantes du composant sont interceptées par le contrôleur des appels sortants (intercepteur « post-requête » dans la figure 1-b) qui interagit avec les autres objets de contrôle pour gérer les déconnexions. Chaque contrôleur est lié à des services de D⊛MINT, respectivement le service de détection de connectivité, le service de journalisation, le service de gestion des composants déconnectés et le service de réconciliation. Nous décrivons ces services dans la section 4.2. En plus de l'accès au service de détection de connectivité, LCD offre un moyen local au conteneur pour gérer le mode de connectivité entre le composant hébergé et les composants connectés avec ce dernier.

4.2. *Rôle des sevices de* D⊛MINT

D⊛MINT offre quatre services pour la gestion des déconnexions. Ces services sont présentés dans la figure 2. Dans notre travail, nous considérons que les services de D⊛MINT sont des services constitués d'objets ajoutés à l'intergiciel sous-jacent.

Le service de gestion des composants déconnectés centralise la gestion de tous les composants déconnectés dans le cache du terminal mobile. Nous avons choisi de partager le cache pour toutes les applications sur le terminal mobile afin d'éviter d'avoir plusieurs instances d'un composant déconnecté dans le terminal mobile. Ce service comporte un objet (*Proxies Manager*, PM) fournissant des services aux conteneurs pour la création et la recherche des composants déconnectés. Le service de gestion des composants déconnectés est accessible par les conteneurs des composants sur le terminal mobile ainsi que par les conteneurs des composants déconnectés (nous développons ce dernier point dans la section 4.3).

Le service de journalisation est responsable de la sauvegarde des opérations effectuées par les composants déconnectés. Il est accessible par les conteneurs à travers le gestionnaire des opérations locales (*Local Operations Manager*, LOM). Les opérations effectuées sur les composants déconnectés sont transmises automatiquement par le conteneur au LOM, et ce dernier crée une entrée dans le journal des opérations pour chaque requête. Une autre fonctionnalité du service de journalisation est d'opérer le transfert des opérations journalisées vers les composants distants. Ce transfert est réalisé par le gestionnaire des mises à jours (*Update Manager*, UM).

Le service de détection de connectivité surveille le niveau de disponibilité des ressources locales pour les communications et exprime la connectivité selon les modes « connecté », « partiellement connecté » ou « déconnecté ». L'apparition d'une déconnexion n'est pas liée qu'à une chute de la bande passante du réseau, mais aussi à d'autres ressources du terminal mobile. Par exemple, la baisse de puissance de la batterie du terminal mobile cause une déconnexion. Le détecteur de connectivité utilise un mécanisme d'hystérésis pour contrôler les variations de niveau de disponibilité des ressources [CON 02].

Le service de réconciliation maintient la cohérence entre le composant distant et les différents composants déconnectés. Il comporte un gestionnaire de diffusions (*Broadcast Manager*, BM), un gestionnaire d'estampillage (*Timestamp Manager*, TM) et un gestionnaire de files d'attente (*Queue Manager*, QM). Le fonctionnement de ce service est décrit dans [CHA 03].

4.3. D⊛MINT *en exécution*

Dans cette section, nous décrivons les interactions entre les différentes entités de D⊛MINT pour la gestion des déconnexions. Nous entendons par entités le conteneur présenté dans la section 4.1 et les différents services présentés dans la section 4.2. Ces interactions sont décrites dans la figure 2. Dans ce papier, nous ne décrivons que les interactions ajoutées pour le contexte des composants ; les autres interactions ont déjà été décrites dans [CON 02].

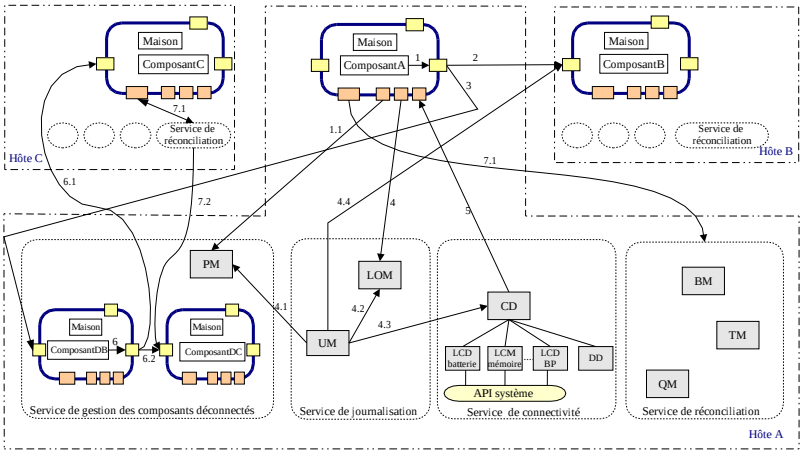

Figure 2. *Architecture de* D⊛MINT

Chaque hôte utilisé par l'application comporte une instance des composantes de D⊛MINT. Elles sont accessibles via le service de nommage de l'intergiciel. D⊛MINT définit un profil qui permet de les configurer. Ce profil définit la taille maximale du

cache pour les composants déconnectés et les différentes valeurs de l'hystérésis du détecteur de connectivité. Les objets de contrôle du conteneur sont créés au moment de l'instanciation du conteneur. Nous pouvons imaginer de créer ces objets au moment de la première invocation du composant, mais cette solution peut donner un temps de latence non négligeable lors de la première requête. En outre, le gain d'espace mémoire ne dure que de l'instanciation du composant à sa première utilisation. À l'instanciation du composant, DCA importe la liste des mandataires qui existent dans le cache et LCD abonne le composant au détecteur de connectivité.

Dans [KOU 03], 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 en mode déconnecté. Dans la suite de ce papier, nous supposons que la première requête sortante d'un composant survient dans le mode connecté et que D⊛MINT doit créer un composant déconnecté pour les composants déconnectables. Ce papier ne discute pas du cas où le composant invoqué est non déconnectable.

Dans la figure 2, lorsque le composant émet une requête, l'intercepteur des appels sortants intercepte la requête (1), contrôle la déconnectabilité de ce composant et vérifie via LCD si le composant déconnecté existe. Dans le cas où le composant déconnecté n'existe pas dans le cache, LCD demande à PM de créer un composant déconnecté pour le composant distant (1.1). PM crée d'abord une maison pour la création du composant déconnecté. Ensuite, l'intercepteur des appels sortants transmet la requête sortante au composant distant (2). Tant que le niveau de connectivité fournit par LCD est bon (mode connecté), l'intercepteur des appels sortants laisse la requête passée vers le composant distant. Par conséquent, dans le mode connecté, comme dans Coda [SAT 96], le composant accède directement au composant distant et le composant déconnecté associé n'est pas à jour.

Dans les modes déconnecté et partiellement connecté, l'intercepteur des appels sortants redirige la requête vers le composant déconnecté (3). Mais avant que l'intercepteur ne redirige cette requête, il connecte le composant avec le composant déconnecté en utilisant CM. Ensuite, LA retransmet la même requête au LOM et ce dernier sauvegarde cette requête dans le journal des opérations (4). Périodiquement, UM récupère la liste des composants déconnectés depuis PM (4.1) et la liste des opérations exécutées pour chaque composant déconnecté depuis LOM (4.2). Ensuite, UM teste la connectivité (4.3) et envoie les requêtes journalisées vers le composant distant si la connectivité le permet (4.4).

Dans la figure 2, nous décrivons quatre scénarios d'interaction possibles entre les composants. L'interaction peut intervenir entre un composant et un composant distant (`composantA` avec `composantB`), entre un composant et un composant déconnecté (`composantA` avec `composantDB`), entre un composant déconnecté et un composant distant (`composantDB` avec `composantC`) et entre deux composants déconnectés dans le même cache (`composantDB` avec `composantDC`). L'architecture du conteneur présentée dans la section 4.1 est la même pour tous les composants

de l'application, y compris les composants déconnectés. Cependant, le conteneur des composants déconnectés n'a pas la possibilité de gérer la cohérence, puisque cette tâche est de la responsabilité du composant distant. En outre, les interactions entre le conteneur et les composantes de D⊛MINT sont les mêmes pour tous les composants. Par exemple, dans la figure 2, le composant déconnecté `composantDB` utilise une interface offerte par le composant `composantC`. Lorsque la connectivité entre le composant `composantDB` et le composant `composantC` est bonne, à la première invocation, le conteneur du composant `composantDB` demande au gestionnaire des composants déconnecté de créer un composant déconnecté pour le composant `composantC` dans le cache. Ensuite, l'intercepteur des appels sortants transmet la requête vers le composant `composantC` après avoir connecté les deux composants (6.1). Par contre, si la connectivité est nulle, l'intercepteur des appels sortants redirige la requête vers le composant déconnecté (`composantDC`) (6.2).

Comme décrit dans la section 3, l'intercepteur des appels entrants offre l'accès aux interfaces du composant. Par ailleurs, l'intercepteur des appels entrants offre une autre interface qui est utilisée par le gestionnaire des mises à jours (UM) (comme dans 4.4). Cette interface transmet les appels entrants vers RA et ce dernier communique avec le service de D⊛MINT associé (7). Le service de réconciliation utilise aussi RA pour faire la réconciliation entre le composant distant et les composants déconnectés associés.

5. Réalisation en cours

Cette section présente l'état actuel de notre réalisation. Tout d'abord, nous justifions le choix de l'utilisation de la plateforme OpenCCM [MAR 01]. Enfin, nous décrivons l'intégration des méta-données dans OpenCCM.

Le choix du modèle CCM est fondé sur le fait que CCM est multilanguage, multi-OS, multi-ORB, multivendeur, contrairement à EJB qui est purement Java, et COM et .NET qui sont purement Microsoft Windows. Un autre avantage de CCM est que le composant peut être segmenté et, pour chaque segment, CCM génère un squelette. Les segments sont activés indépendamment les uns des autres et possèdent un état. CCM définit aussi un nouveau langage, *Component Implementation Definition Language* (CIDL), pour décrire les interactions entre les parties fonctionnelles et extrafonctionnelles du composant. Le choix d'OpenCCM est justifié par le fait qu'il est le seul à offrir une implantation (partielle) libre de CCM en Java.

En plus de la méta-donnée *déconnectabilité* décrite dans la section 4.3, nous avons proposé la méta-donnée *nécessité* qui donne un « poids » aux entités de l'application quand à leur présence dans le terminal mobile [KOU 03]. Nous avons ajouté dans la syntaxe CIDL de nouvelles entrées grammaticales pour spécifier la déconnectabilité et la nécessité de chaque composant et de ses segments. Cette syntaxe est décrite dans la figure 3. Nous spécifions d'abord la déconnectabilité et la nécessité du composant (ligne 7), ensuite, celles des segments (ligne 12). Les valeurs affectées aux méta-

données (lignes 14 et 15) sont « *yes* » ou « *no* ». La chaîne de compilation d'OpenCCM a été modifiée pour prendre en compte cette nouvelle syntaxe dans la génération du code. Ainsi, chaque squelette du segment comporte le code nécessaire pour spécifier les méta-données. CCM définit la notion de segment principal qui contient toutes les interfaces et tous les attributs non affectés à d'autres segments et qui offre la possibilité de localiser les autres segments par l'interface *ExecutorLocator*. Donc, il est clair que la déconnectabilité et la nécessité du composant implique la déconnectabilité et la nécessité de son segment principal.

Par manque de place, nous ne pouvons pas argumenté la nécessité de modifier le CIDL. En un mot, les méta-données sont affectées dès l'architecture, la description CIDL devant contenir ces informations.

```
1  ...
2  <executor_ref>  ::=
3                      "manages"
4                      [<executor_body>] ";"
5  <executor_body> ::= "{" < body_def> "}"
6  < body_def>     ::=
7                      <disconnection_management>
8                      <segment_def>
9  <segment_def>   ::=
10                     <seg_per_decl> ";"
11                     <facet_decl> ";"
12                     <disconnection_management>
13 <disconnection_management> ::=
14                             "disconnectable" <value> ";"
15                             "necessity" <value> ";"
16 ...
```

Figure 3. *Syntaxe CIDL pour la définition des méta-données*

6. Travaux connexes

Beaucoup de travaux ont été effectués dans le domaine de l'intégration des services extrafonctionnels dans les conteneurs des composants, mais peu d'entre eux ont traités le service de gestion de déconnexions. Par exemple, [MEG 02] définit une architecture pour intégrer la gestion de la qualité de service (QoS) dans les conteneurs EJB, [ROU 03] propose une architecture orientée composant pour réaliser les politiques transactionnelles indépendamment des plates-formes pour composants.

Les travaux sur la gestion de déconnexions peuvent être classés en deux catégories : les approches basiques et les approches dédiées. Les approches basiques sont les approches qui ne se basent pas sur un paradigme ou un principe particulier. Un panorama est donnée pour ces approches dans [JIN 99] sur la gestion de déconnexions des applications client/serveur. La plupart des travaux décrits ne se situent pas dans le contexte composant et ne traitent pas la séparation des aspects. Ils discutent en particulier de la gestion des mandataires dans le terminal mobile. Les approches dédiées sont les approches qui utilisent un principe bien défini. Les principes les plus utilisés sont

la programmation par aspect (AOP) [KIC 97] et la réflexion [PAR 00]. La réflexion permet de donner au système une représentation de lui-même. Cette représentation lui permet d'agir sur lui-même pour permettre l'introspection et l'adaptation. L'introspection permet d'observer son implantation, alors que l'adaptation permet de modifier son comportement. Il existe deux types de réflexion sur les intergiciels : la réflexion structurale et la réflexion comportementale [CAP 02]. Dans D⊛MINT, nous avons utilisé les deux types de réflexion. La réflexion structurale est utilisée pour la création des composants déconnectés et leur connecteur. La réflexion comportementale est utilisée pour l'interception des appels entrants et sortants au niveau du conteneur.

L'architecture des conteneurs que nous avons présentée dans ce papier est inspirée de l'architecture des conteneurs ouverts [VAD 01], qui définit une chaîne complète pour la description, la production, le déploiement et l'administration des conteneurs. Le conteneur ouvert est caractérisé par trois couches : une couche d'interception, une couche de coordination et une couche de contrôle. Dans notre architecture, l'intercepteur joue le rôle de la couche d'interception et de coordination, et les autres objets de contrôle jouent le rôle de la couche de contrôle. Le conteneur ouvert tel que décrit dans [VAD 01] n'intercepte pas les appels sortants, mais nous pensons que l'utilisation de la même architecture sur les appels sortants ne posera pas de problème.

7. Conclusion et perspectives

Dans ce papier, nous avons présenté l'état actuel de D⊛MINT, une architecture de gestion des déconnexions pour des applications à base de composants dans les environnements mobiles. Nous avons décrit une architecture des conteneurs dédiés pour la gestion des déconnexions et les quatre services utilisés par les conteneurs : détection de connectivité, journalisation, gestion des composants déconnectés et réconciliation (ou gestion de la cohérence).

Les caractéristiques de l'architecture ainsi construite sont les suivantes. L'utilisation d'un intercepteur au niveau des conteneurs plutôt qu'un intercepteur portable de l'intergiciel CORBA limite le surcoût qui peut être généré par la détection de connectivité lors de chaque requête sortante. En outre, nous limitons le surcoût de la réflexion dans la création des composants déconnectés. Par ailleurs, le gestionnaire des composants déconnectés gère un cache de composants déconnectés global au terminal mobile. Il permet aussi de créer des connecteurs pour les communications entre composants déconnectés ainsi qu'avec les composants distants. D'autre part, l'intercepteur des appels entrants redirige toutes les requêtes entrantes vers le gestionnaire de cohérence qui gère la réconciliation. Enfin, bien que le prototype que nous réalisons se base sur le modèle de composants CORBA (CCM), les concepts proposés sont suffisamment généraux pour être utilisés dans d'autres modèles de composants, dans la mesure où ils respectent le modèle générique décrit dans la section 3.

Dans nos travaux futurs, nous allons terminer l'implantation de D⊛MINT, porter cette architecture sur des terminaux mobiles (PDA) et faire des tests de performances.

Par ailleurs, de nouvelles problématiques se dessinent, telles que l'invocation d'un composant qui ne comporte pas encore de mandataire en mode déconnecté. En outre, les composantes de D⊛MINT sont, d'une part, utilisées par tous les objets de contrôle du conteneur, et d'autre part, partagées par plusieurs composants du terminal. Ces composantes interagissent aussi entre elles. Comme il est montré dans [HéR 03], un moyen pour mieux gérer ce partage et ces interactions est de les *fractaliser*. Enfin, dans notre étude préliminaire, nous avons utilisé la granularité des segments définis dans le modèle CCM. Donc, un autre axe de travail est d'intégrer le principe de segmentation dans la plateforme D⊛MINT. Cela fournirait une meilleure disponibilité des services applicatifs en mode déconnecté.

8. Bibliographie

[CAP 02] CAPRA L., BLAIR G., MASCOLO C., EMMERICH W., GRACE P., « Exploiting Reflection in Mobile Computing Middleware », *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, n° 4, 2002.

[CHA 03] CHATEIGNER L., CHABRIDON S., BERNARD G., « Intergiciel pour l'informatique nomade : réplication optimiste et réconciliation », *Actes. Manifestation des jeunes chercheurs STIC, MAJECSTIC*, Marseille, France, Octobre 2003.

[CON 02] CONAN D., CHABRIDON S., VILLIN O., BERNARD G., « Disconnected Operations in Mobile Environments », *Proc. 2nd IPDPS Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, Ft. Lauderdale, USA, Avril 2002.

[COU 03] COUPAYE T., BRUNETON E., STEFANI J., « The Fractal Composition Model », rapport, Septembre 2003, ObjectWeb, France.

[DEM 02] DEMICHIEL L., « Enterprise JavaBeans Specifications, version 2.1, proposed final draft », Sun Microsystems, http ://java.sun.com/products/ejb/docs.html, août 2002.

[HéR 03] HÉRAULT C., LECOMTE S., « Adaptabilité des Services Techniques dans un Modèle à Composants », *Actes. 3ème Conférence Française sur les Systèmes d'Exploitation*, La colle sur Loup, France, Octobre 2003.

[JIN 99] JING J., HELAL A., ELMAGARMID A., « Client-Server Computing in Mobile Environments », *ACM Computing Surveys*, vol. 31, n° 2, Juin 1999.

[KIC 97] KICZALES G., LAMPING J., MENHDHEKAR M., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », *Proc. European Conference on Object-Oriented Programming*, vol. 1241, p. 220–242, Springer-Verlag, 1997.

[KOU 03] KOUICI N., CONAN D., BERNARD G., « Disconnected Metadata for Distributed Applications in Mobile Environments », *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, Juin 2003.

[MAR 01] MARVIE R., MERLE P., « CORBA Component Model : Discussion and Use with OpenCCM », rapport, 2001, Laboratoire d'Informatique Fondamentale de Lille, France.

[MEG 02] MEGUEL A., « Integration of QoS Facilities into Component Container Architectures », *Proc. Object oriented Real-Time Distributed Computing*, Washington, DC, USA, 2002, p. 394–401.

[Mic 03] MICROSOFT CORPORATION, « Microsoft Developer Network », http ://www.msdn.microsoft.com, 2003.

[Obj 02] OBJECT MANAGEMENT GROUP, « CORBA Components », OMG Document n° formal/02-06-65, Version 3.0, Juin 2002.

[Obj 03] OBJECTWEB OPEN SOURCE SOFTWARE COMMUNITY, « Fractal home page », http ://www.fractal.objectweb.org, 2003.

[PAR 00] PARLAVANTZAS N., COULSON G., CLARKE M., BLAIR G., « Towards a Reflective Component-based Middleware Architecture », *Workshop on Reflection and Metalevel Architectures*, Sophia Antipolis et Cannes, France, Juin 2000.

[ROU 03] ROUVOY R., MERLE P., « Abstraction of Transaction Demarcation in Component-Oriented Platforms », *Proc. ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, Juin 2003.

[SAT 96] SATYANARAYANAN M., « Mobile Information Access », *IEEE Personal Communications*, vol. 3, n° 1, 1996.

[TER 95] TERRY D., THEIMER M., PETERSEN K., DEMERS A., SPREITZER M., HAUSER C., « Managing Update Conflicts in Bayou : A Weakly connected Replicated Storage System », *Proc. 15th Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, 1995.

[VAD 01] VADET M., MERLE P., « Les conteneurs ouverts dans les plates-formes à composants », *Actes. Journée Thème Émergent Composants*, Besançon, France, Octobre 2001.

[VOL 01] VOLTER M., « Server-side Components - A Pattern Language », *Proc. 6th European Conference On Pattern Languages of Programs*, Irsee, Germany, 4-8 Juillet 2001.