Gestion des déconnexions pour applications réparties à base de composants en environnements mobiles

Nabil Kouici

Institut National des Télécommunications

Université d'Évry Val d'Essonne

Thèse de doctorat de l'Institut National des Télécommunications dans le cadre de l'école doctorale SITEVRY en co-accréditation avec l'Université d'Évry Val d'Essonne

spécialité informatique

Par :

NABIL KOUICI

Thèse présentée pour l'obtention du grade de Docteur de l'Institut National des Télécommunications

Gestion des déconnexions pour applications réparties à base de composants en environnements mobiles

Soutenue le 16 novembre 2005 devant le jury composé de :

Rapporteurs :

M. Andrzej Duda	Professeur à l'INPG-ENSIMAG
M. Michel Riveill	Professeur à l'ESSI & à l'Université de Nice - Sophia Antipolis

Examinateurs :

M. Philippe Merle	Chargé de Recherche à l'INRIA Lille
M. Bruno Traverson	Ingénieur-chercheur à EDF R&D
M. Guy Bernard	Professeur à l'INT (Directeur de thèse)
M. Denis Conan	Maître de conférence à l'INT (Encadrant)

Thèse numéro : xxxxxxxxxxxx

Résumé

Ces dernières années ont été marquées par une forte évolution des équipements et des réseaux utilisés dans les environnements mobiles. Cette évolution a abouti à la définition d'une nouvelle thématique : l'informatique mobile. L'informatique mobile offre aux utilisateurs la capacité de pouvoir se déplacer tout en restant connecté aux applications réparties et d'être indépendant de la localisation géographique. Toutefois, l'accès aux applications réparties dans ces environnements soulève le problème de la disponibilité des services en présence des déconnexions. Ces déconnexions peuvent être volontaires ou involontaires.

Les principaux intergiciels qui existent aujourd'hui sont inadéquats pour les environnements mobiles où les ressources (bande passante, batterie, mémoire...) peuvent spontanément varier considérablement voire disparaître. Ils sont destinés aux environnements traditionnels relativement statiques dans lesquels les ressources sont disponibles et pratiquement stables. Par ailleurs, 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 meilleure séparation entre les préoccupations fonctionnelles et extrafonctionnelles. Cette séparation est réalisée suivant le paradigme composant/conteneur.

Cette thèse s'intéresse à la gestion des déconnexions pour applications réparties à base de composants dans les environnements mobiles. La solution consiste à maintenir une connexion logique en utilisant le concept d'opération déconnectée. Cependant, la plupart des solutions existantes sont souvent des réponses « ad hoc ». En effet, ces solutions ne proposent pas une séparation entre les préoccupations fonctionnelles de l'application et la gestion des déconnexions. Cette contrainte limite les possibilités de maintenance, de réutilisation et de reconfiguration. Ces solutions ne proposent pas non plus de modèle de conception d'applications réparties devant fonctionner en présence des déconnexions. Enfin, le modèle orienté composant est peu investi dans la gestion des déconnexions, cette dernière limitation étant due à la nouveauté de ce modèle.

Dans cette thèse, nous présentons MADA, une approche de conception d'applications réparties pour le fonctionnement en présence des déconnexions qui suit l'approche MDA. Dans cette approche, la gestion des déconnexions est abordée dès la modélisation de l'application. Ensuite, nous présentons un service intergiciel pour la gestion du cache du terminal mobile. Nous validons la solution proposée à l'aide d'un prototype réalisé en Java, pour application à base de composants CORBA, dans le cadre du canevas logiciel DOMINT. Nous proposons aussi d'intégrer la gestion des déconnexions dans les conteneurs des composants. En nous basant sur le modèle du conteneur extensible (ECM) de OpenCCM, nous proposons une spécification et une réalisation Java/CCM de notre conteneur.

Mots-clés : Intergiciel, informatique mobile, déconnexion, composant, adaptabilité.

Abstract

Last years have been marked by a rapid evolution in computer networks and machines used in distributed environments. This evolution has opened up new opportunities for mobile computing. Mobile computing allows a mobile user to access various kinds of information at any time and in any place. However, mobile computing raises the problem of data availability in the presence of disconnections. We distinguish two kinds of disconnections : voluntary disconnections and involuntary disconnections.

Traditional middleware are mainly connection-oriented programming environments in which a client must maintain a connection to a server. These middleware are inadequate for mobile computing where the resources are unstable (bandwidth, battery, memory...). In addition, the development of distributed applications converges more and more towards the use of component-oriented middleware that better addresses the application complexity by separating functional and extra-functional concerns using the component/container paradigm.

The objective of this work is the disconnection management of component-based applications in mobiles environments. The solution consists in maintaining a logical connection between a client and its servers using the concept of disconnected operation. However, the majority of the existing solutions present an « ad hoc » solutions. Indeed, these solutions do not propose a separation between functional concerns and disconnection management. These solutions do not propose a disconnection-aware approach to design distributed applications that have to work in the presence of disconnections. Moreover, the component-oriented paradigm is rarely invested in disconnection management, this last limitation being due to the newness of this model.

In this PhD Thesis, we present MADA, a mobile application development approach. In this approach, disconnection management is taken into account when modelling application at the architectural level. Then, we present a middleware service for the software cache management of mobile terminal. We validate the solution using a prototype implemented in Java, for CORBA component-based application, within the DOMINT platform. We also integrate the disconnection management in the containers. Finally, we propose a specification and a Java/CCM implementation of our container using the extensible container model (ECM) of OpenCCM.

Keywords : Middleware, mobile computing, disconnection, component, adaptability.

Table des matières

Table des figures

Liste des tableaux

Introduction

Les récentes avancées dans le domaine des communications sans fil aussi bien du point de vue réseau (GSM, GPRS, UMTS, Wi-Fi...) que matériel (assistants personnels numériques, PC portables, téléphones portables...) ont rendu possibles de nouvelles applications dans lesquelles l'utilisateur peut avoir accès à l'information à n'importe quel moment et depuis d'importe quel endroit. Ainsi, un nouveau paradigme est apparu connu sous le nom d'*informatique mobile*. L'informatique mobile se caractérise par le nomadisme des utilisateurs qui introduit le facteur de variabilité des capacités matérielles : mémoire, bande passante, batterie... Cette variabilité ne doit pas être vue comme une faute puisqu'elle est une conséquence de la mobilité des utilisateurs. Malheureusement, elle peut induire des déconnexions réseau. Nous considérons deux types de déconnexions : les déconnexions volontaires et les déconnexions involontaires. Les premières, décidées par l'utilisateur depuis son terminal mobile, sont justifiées par les bénéfices attendus sur le coût financier des communications, l'énergie, la disponibilité du service applicatif, et la minimisation des désagréments induits par des déconnexions inopinées. Les secondes sont le résultat de coupures intempestives des connexions physiques du réseau, par exemple, lors du passage de l'utilisateur dans une zone d'ombre radio.

Par ailleurs, le développement des intergiciels et des applications réparties pour les environnements mobiles est générateur de nouveaux défis. D'une part, de nombreuses problématiques similaires à celles rencontrées en environnements classiques nécessitent des traitements différents, d'autre part, des problématiques spécifiques à ces environnements doivent être traitées.

Motivations

La problématique étudiée dans cette thèse découle du problème exposé ci-dessus, et plus spécialement des déconnexions réseau. Plus précisément, notre travail se situe dans la problématique de la gestion des déconnexions des applications réparties à base de composants en environnements mobiles.

La gestion des déconnexions passe par une adaptation aux changements de niveau de disponibilité des ressources. Cette adaptation peut être entièrement de la responsabilité de l'application (stratégie « laissez-faire »), de la responsabilité du système ou de l'intergiciel (stratégie « transparence »), ou encore, effectuée par la collaboration entre l'application et le système (stratégie « collaboration ») [144]. De nombreux travaux montrent que les approches « laissez-faire » et

« transparence » ne sont pas adéquates [72]. Aussi, dans nos travaux, nous adoptons la stratégie « collaboration ». De plus, cette adaptation peut être de trois types. 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. 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.

Par ailleurs, la construction d'applications réparties converge de plus en plus vers l'utilisation d'intergiciels orientés composants tels que CCM, EJB et .Net. 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 préoccupations fonctionnelles et extrafonctionnelles. Cette séparation est réalisée suivant le patron de conception composant/conteneur : le composant encapsule les préoccupations fonctionnelles et le conteneur gère les préoccupations extrafonctionnelles [170]. Les préoccupations extrafonctionnelles les plus utilisées sont la persistance, la gestion des transactions, la sécurité et la distribution [29]. La préoccupation de gestion des déconnexions, qui est l'objet de cette thèse, n'est que rarement considérée.

Contributions

Les contributions apportées par ce travail de recherche s'insèrent notamment dans deux projets européens : *ITEA Eureka OSMOSE* [128] et *RNTL franco-finlandais AMPROS* [1]. La solution que nous proposons se base sur le principe des opérations déconnectées. En présence des déconnexions, le client sur le terminal mobile utilise les composants préalablement déployés dans un cache logiciel local. Les opérations effectuées sur les composant locaux pendant les phases de déconnexions sont journalisées et des réconciliations lors des reconnexions sont opérées entre les composants locaux et les composants dans les sites distants. L'utilisation des opérations déconnectées pour le fonctionnement en présence des déconnexions fait apparaître quatre thématiques : la gestion du cache du terminal mobile, la détection des déconnexions, la commutation entre les composants locaux et les composants distants et la gestion de la cohérence des différents composants. Le travail que nous présentons dans cette thèse aborde la gestion du cache et de la commutation entre les composants locaux et les composants distants. Pour les autres thématiques, le lecteur peut se référer à [30] pour la gestion de la cohérence et à [158] pour la détection des déconnexions. Les principales contributions de cette thèse sont :

- **MADA**, une approche de conception d'applications pouvant fonctionner en présence des déconnexions. Le but de l'approche MADA n'est pas de proposer une nouvelle méthode de conception d'applications réparties, mais plutôt de prendre en considération le problème de la gestion du cache pour les déconnexions dès la conception de l'application ;
- un **service de gestion du cache** qui s'occupe de la gestion des composants déployés dans le cache du terminal mobile. Il fournit la mise en œuvre des stratégies de déploiement (défi-

nir les composants à déployer localement, quand et pour quelle durée) et de remplacement (définir les composants qui doivent être supprimées lorsqu'il n'existe plus assez d'espace mémoire dans le cache). La mise en œuvre de ce service est effectuée dans le cadre du canevas logiciel DOMINT [39] ;

– l'**intégration de la gestion des déconnexions dans les conteneurs des composants**, dans le but de séparer les préoccupations fonctionnelles et extrafonctionnelles de l'application. La mise en œuvre de cette intégration se base sur le modèle du conteneur extensible (en anglais, ECM pour *Extensible Container Model*) [165] de la plateforme OpenCCM [127].

Nous pensons montrer dans ce travail que la gestion des déconnexions doit être prise en compte dans tout le cycle de vie de l'application. De plus, nous montrons aussi que le principe de séparation des préoccupations convient aussi au problème de la gestion des déconnexions.

Plan du document

Cette thèse est découpée en deux parties et 6 chapitres. Dans la première partie, nous présentons le contexte de nos travaux ainsi que la problématique de gestion des déconnexions :

– le chapitre 1 constitue une introduction à l'informatique mobile. Dans la première partie de ce chapitre, nous décrivons les caractéristiques de l'informatique mobile tout en prêtant une attention particulière au problème de déconnexion. La deuxième partie de ce chapitre est consacrée aux technologies liées à l'informatique mobile : les réseaux sans fil et les terminaux mobiles ;

– le chapitre 2 présente dans sa première partie la notion d'intergiciel et les besoins qui ont conduit à son apparition. Cette description est développée par l'étude des deux types d'intergiciels qui reflètent les paradigmes de communication synchrone et asynchrone. La deuxième partie de ce chapitre aborde les différentes techniques de séparation des préoccupations fonctionnelles et extrafonctionnelles. Nous mettons en valeur cinq techniques : modélisation de l'architecture, paradigme composant/conteneur, réflexivité, programmation orientée aspects et mécanisme d'interception ;

– le chapitre 3 fournit l'état de l'art sur les travaux relatifs à la gestion du cache pour les déconnexions. Après avoir présenté les différentes problématiques liées à la gestion des déconnexions, nous étudions dans ce chapitre les solutions proposées dans la littérature par rapport à la gestion du cache du terminal mobile pour la gestion des déconnexions. Cette étude nous permet d'appréhender l'objet même de notre travail et de comprendre quels sont les critères qui peuvent caractériser une solution.

La deuxième partie de cette thèse présente nos contributions :

– le chapitre 4 est consacré à la présentation de MADA, une approche de conception d'applications pouvant fonctionner en présence des déconnexions. Nous décrivons notamment le profil de l'application qui sert à décrire les entités de l'application pour la gestion du cache, le profil UML pour la gestion des déconnexions qui permet d'étendre le processus de développement d'une application CCM pour prendre en compte la gestion des déconnexions, et enfin, le graphe de dépendances qui fait apparaître explicitement les différentes interactions

entres les entités de l'application afin de les prendre en compte dans la gestion du cache du terminal mobile ;

– le chapitre 5 présente l'architecture et la mise en œuvre du service de gestion du cache ainsi que le modèle de conteneur du composant que nous proposons dans le cadre du canevas logiciel DOMINT. À travers ce chapitre, nous illustrons l'utilisation de différents éléments de MADA dans le service de gestion du cache et le modèle de conteneur du composant ;
– le chapitre 6 présente des résultats expérimentaux permettant de valider la solution proposée.

Première partie

État de l’art

Chapitre 1

Introduction à l'informatique mobile

Dans une étude sur la mobilité éditée en décembre 2002 par *IDC* (http://www.idc.fr), plus de 4,5 millions de personnes en France sont mobiles au sens qu'ils travaillent sur plus 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 sans fil de l'entreprise, comme elle peut être étendue, avec les réseaux sans fil à grande échelle. Par ailleurs, les réseaux informatiques ont évolué des simples réseaux locaux filaires à des réseaux sans fil interconnectant des équipements mobiles tels que les assistants personnels numériques (*Personal Digital Assistant*, PDA) 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 répartie quelque soit son emplacement.

Ce chapitre a pour objectif d'introduire l'informatique mobile et les technologies utilisées dans les environnements mobiles. Nous donnons dans ce chapitre les caractéristiques de base de l'informatique mobile tout en prêtant une attention particulière au problème des déconnexions. Étant donné la richesse et la diversité des technologies liées à l'informatique mobile, nous présentons les concepts plutôt que les détails techniques. Ainsi, après avoir défini les concepts de base de l'informatique mobile (cf. section 1.1), nous présentons les différentes formes de la mobilité dans la section 1.2 et les technologies liées à l'informatique mobile dans la section 1.3. Enfin, la section 1.4 conclut sur les différents points abordés dans ce chapitre.

1.1 Informatique mobile

Depuis le début des années 90, la forte évolution réalisée dans les réseaux sans fil et les terminaux mobiles suscite un intérêt croissant pour l'informatique [49]. En outre, l'être humain se caractérise par son nomadisme : il semble avoir été le premier à avoir quitté sa terre d'origine et peuplé progressivement les différents continents. Nous ne voulons pas rentrer dans l'histoire de l'humanité, mais juste souligner que l'être humain est par nature nomade. Aujourd'hui, cette ambition de nomadisme n'est pas uniquement liée à un désir ardent de promotion sociale mais aussi à une vision professionnelle [100]. En effet, dans son entourage professionnel, l'être humain utilise

quotidiennement différentes sortes d'ordinateurs, au travail, à la maison ou même lorsqu'il voyage [145, 116, 157]. Ainsi, un nouveau paradigme est apparu, connu sous le nom d'informatique mobile. L'informatique mobile offre un mécanisme de communication flexible entre les utilisateurs et un accès à l'ensemble des services normalement disponibles dans un environnement classique (fixe), à travers un réseau, indépendamment de la localisation physique (géographique) et des déplacements de l'utilisateur.

Nous décrivons dans les prochaines sections les différentes étapes de constitution de l'informatique mobile, ses caractéristiques et les problèmes qu'elle introduit.

1.1.1 Introduction à l'informatique mobile

La figure 1.1 illustre l'évolution de l'informatique par rapport à l'indépendance à la localisation et à la flexibilité d'utilisation de ressources [81, 145]. Les premières machines informatiques étaient équipées d'un système d'exploitation mono-utilisateur et obligeaient l'utilisateur à être physiquement présent à côté de l'ordinateur. La deuxième étape de l'évolution est l'introduction du système d'exploitation multitâche qui permet de partager les ressources de l'unité de calcul centrale entre plusieurs programmes. Cependant, bien que l'utilisateur ne soit pas obligé d'exécuter ces tâches séquentiellement, sa présence physique est toujours exigée. L'étape suivante est l'apparition du système d'exploitation multiutilisateur. À cette époque, la machine est située dans une salle informatique, les utilisateurs y étant connectés au moyen d'un terminal ou d'un émulateur de terminal. Ceci permet de partager les ressources de l'ordinateur et d'être physiquement un peu plus éloigné de l'ordinateur. Ensuite, avec l'évolution des réseaux informatiques et des postes de travail, plusieurs limitations ont été résolues. Les utilisateurs disposent de leurs propres ressources, comme les postes de travail personnels dédiés, et de ressources partagées, comme les serveurs et les imprimantes qui sont accessibles à travers le réseau. Les systèmes répartis ont donc été l'étape suivante de l'évolution en offrant un éloignement géographique de grande envergure à travers des réseaux informatiques terrestres.

L'étape suivante de l'évolution avait comme objectif d'offrir à l'utilisateur plus d'indépendance envers son emplacement géographique. Les utilisateurs peuvent accéder aux ressources du système n'importe quand et n'importe où. Ainsi, l'informatique mobile vient répondre à ces objectifs. Enfin, l'étape la plus récente de l'évolution est l'informatique ambiante (en anglais, *pervasive computing*). Tout en incluant les domaines de recherche de l'informatique mobile, l'informatique ambiante ajoute d'autres domaines de recherche tels que la gestion efficace du contexte comme la localisation géographique [145].

Les environnements d'exécution des applications réparties classiques se basent sur plusieurs hypothèses [72]. Dans ces environnements, les ordinateurs sont connectés une fois pour toute à un réseau filaire et se caractérisent aussi par leurs puissances en termes de capacité de traitement et de stockage. En outre, les réseaux filaires offrent une bande passante stable, de bonne qualité et peu coûteuse voire même gratuite. Ces hypothèses sont différentes de celles de l'informatique mobile mettant en œuvre des réseaux sans fil et des terminaux mobiles. Dans un réseau sans fil, les utilisateurs sont mobiles et peuvent accéder à leurs applications depuis n'importe quelle localisation géographique. Cependant, la communication sans fil est moins fiable que dans

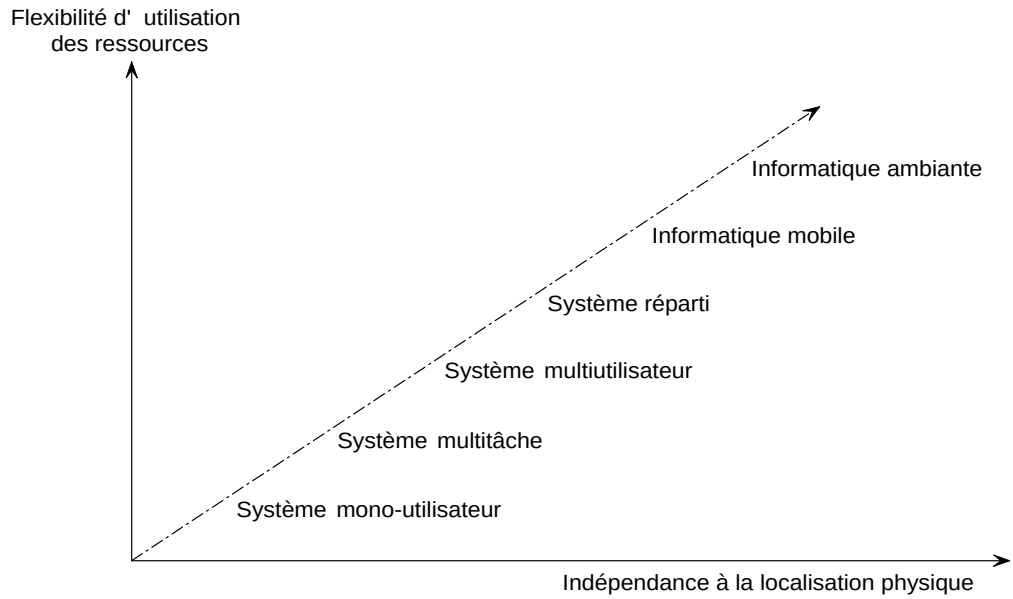


FIG. 1.1 – Évolution de l'informatique

les réseaux filaires. Le signal propagé sur le réseau sans fil subit des perturbations dues à l'environnement. Dans les cas extrêmes, ces perturbations conduisent à des déconnexions du terminal mobile. Nous décrivons en détail les réseaux sans fil dans la section 1.3.1. En ce qui concerne les terminaux mobiles, ils disposent de ressources limitées tant en puissance de traitement et de stockage qu'en autonomie d'énergie. Ainsi, pour être utilisé dans un environnement mobile, le terminal mobile doit être léger, conservateur d'énergie et facile à utiliser. Les terminaux mobiles se différencient selon leur taille physique, leur autonomie électrique, leur taille d'écran, leur capacité de stockage et leur puissance de calcul. Nous illustrons les différents compromis qui existent entre ces caractéristiques dans la section 1.3.2.

1.1.2 Problèmes de l'informatique mobile

Dans cette section, nous abordons les problèmes liés à l'informatique mobile, et plus spécifiquement ceux liés aux réseaux sans fil et aux terminaux mobiles. Nous essayons de caractériser l'impact de ces problèmes sur les applications réparties, les algorithmes et les protocoles d'un point de vue réseau et système. Nous nous limitons aux problèmes majeurs de l'informatique mobile en démontrant que tous ces problèmes convergent vers le problème de déconnexion qui fait l'objet de cette thèse.

L'informatique mobile est particulièrement touchée par la limitation et la variation de la bande passante. Ces deux caractéristiques introduisent des délais de transmission des données sur le réseau. Ces délais de transmission dégradent les performances des applications ayant, par exemple, des contraintes de temps telles que les applications multimédias interactives de type visioconférence ou les jeux en ligne multiutilisateurs. Dans les environnements mobiles, la variation

de la bande passante peut être, par exemple, le résultat d'un changement de type de réseau lors du passage d'un réseau sans fil avec un débit de 10 Mb/s vers un réseau sans fil avec un débit de 2 Mb/s. Cette variation peut aussi être le résultat de la dégradation de signal dû à des obstacles (bâtiments, tunnels. . .).

Dans le cas extrême, la variation de la bande passante peut provoquer des déconnexions réseau du terminal mobile [49]. Ce type de déconnexion est appelé « déconnexion involontaire ». L'utilisateur du terminal mobile peut aussi être gêné par la variation de la bande passante et peut à tout moment décider de se déconnecter, par exemple, pour diminuer les coûts financier des connexions sans fil ou pour minimiser les désagréments induits par des déconnexions inopinées. Ce type de déconnexion est appelé « déconnexion volontaire ». Par ailleurs, la durée de la déconnexion peut varier d'une déconnexion à l'autre. Par définition, les déconnexions courtes sont des déconnexions non perceptibles par les utilisateurs. Ces déconnexions sont traitées par le système de gestion du réseau. Par exemple, en téléphonie mobile, le transfert intercellulaire est effectué sans interruption de la communication. Cependant, le temps de basculement n'est pas nul. Ce temps correspond au temps entre la déconnexion d'une cellule et la connexion dans une autre cellule. En outre, les déconnexions courtes sont toujours involontaires. En revanche, les déconnexions longues sont des déconnexions perceptibles par l'utilisateur et qui sont dues à la dégradation de la bande passante. Le tableau 1.1 présente les différents types de déconnexions qui existent. Dans cette thèse, nous traitons des déconnexions longues volontaires et involontaires. Dans la suite de cette thèse, pour des raisons de simplification, le mot déconnexion désigne une déconnexion longue.

Déconnexion	Volontaire	Involontaire
Courte		X
Longue	X	X

TAB. 1.1 – Types de déconnexion

Les applications réparties risquent de ne plus fonctionner correctement en présence de déconnexions. Malheureusement, les déconnexions sont des évènements fréquents dans les environnements mobiles et ne doivent pas être traitées comme des défaillances [136] et les applications réparties doivent fonctionner en présence de déconnexions le plus normalement possible. Cela nécessite des mécanismes qui doivent apporter la valeur ajoutée qui différencie les applications et les systèmes pour les environnements mobiles par rapport à ceux conçus pour les environnements filaires. Le travail que nous présentons dans cette thèse propose de traiter cette problématique.

Par ailleurs, les terminaux mobiles deviennent de plus en plus puissants en terme de ressources offertes (cf. section 1.3.2). Cependant, ces ressources présentent des performances qui sont loin d'atteindre les performances des terminaux fixes. En fait, pour qu'un terminal soit portable, il doit être léger et de petite taille. Ces caractéristiques limitent la capacité de stockage, de traitement et de visualisation. Dans de telles conditions, l'utilisateur peut être obligé de se déconnecter volontairement pour économiser la batterie dont le rechargement n'est pas toujours possible au moment du déplacement.

1.2 Typologie de la mobilité

La mobilité peut être de trois types : mobilité du terminal, mobilité de l'utilisateur et mobilité de la session [135]. Nous schématisons ces trois types dans la figure 1.2. Nous retrouvons les déconnexions volontaires et involontaires dans chaque type de mobilité. Cependant, dans cette thèse, nous nous limitons aux déconnexions pour la mobilité du terminal. Par souci de complétude, nous décrivons ces trois types de mobilité dans les sections suivantes.

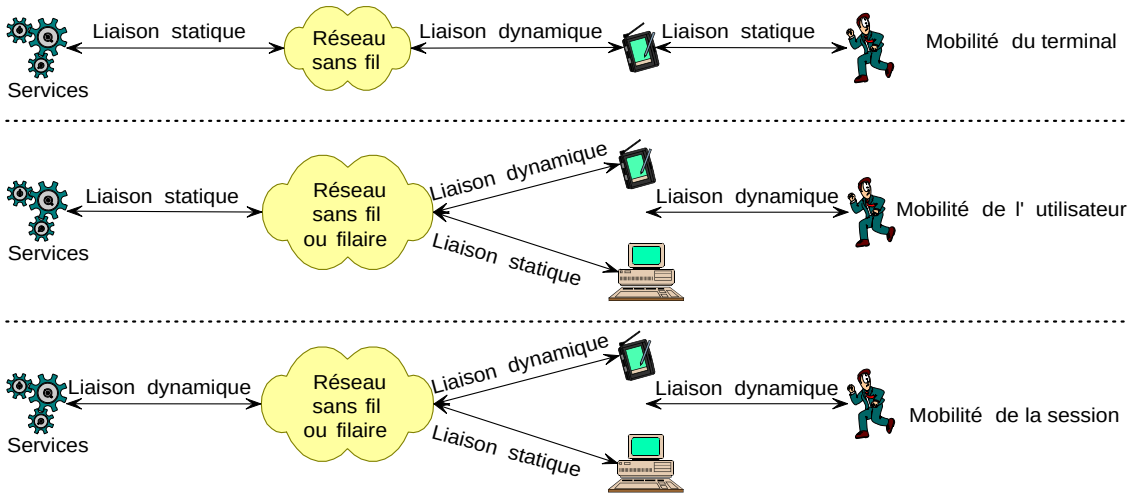


FIG. 1.2 – Typologie de la mobilité

1.2.1 Mobilité du terminal

Aussi désignée par *mobilité des unités*, elle définit la capacité d'un terminal à changer sa localisation physique, en permettant à ce terminal d'accéder aux services à partir de n'importe quelle localisation. Cette mobilité est en effet le résultat du déplacement de l'utilisateur du terminal. L'utilisateur et le terminal forment alors une seule entité logique que nous appelons *unité mobile*. Ainsi, la relation qui existe entre le terminal et l'utilisateur est statique. Le réseau doit localiser et identifier l'unité mobile en déplacement tout en permettant l'accès aux services répartis de n'importe quel endroit. La relation entre le terminal mobile et le réseau est donc dynamique, tandis que la relation entre le réseau et les services est statique.

La mobilité du terminal peut être classée suivant la disponibilité des services en deux catégories [70] : discrète ou continue. Dans la mobilité discrète du terminal, le déplacement de l'unité mobile est contrôlée seulement quand l'utilisateur n'est pas en communication. En d'autres termes, l'environnement mobile doit offrir la continuité de service à partir d'endroits fixes (maison, bureau. . .) mais pas lorsque l'unité mobile se déplace d'un endroit à un autre. Par contre, la mobilité continue du terminal exige le maintien de la communication même lorsque l'unité mobile se déplace.

La mobilité continue des terminaux offre un avantage aux utilisateurs dans la mesure où ils peuvent continuer à travailler tout en se déplaçant. Cependant, cette mobilité exige des capacités réseau importantes [161]. En effet, les systèmes de gestion des réseaux sans fil doivent prendre en compte la mobilité des unités intra-réseau et inter-réseaux (cf. section 1.3.1). Les recherches qui existent sur la mobilité du terminal se focalisent particulièrement sur les aspects réseau [43] et les aspects système [126]. Le travail que nous présentons dans cette thèse aborde le problème des déconnexions d'un point de vue système.

1.2.2 Mobilité de l'utilisateur

La mobilité de l'utilisateur définit la capacité d'un utilisateur mobile à accéder aux services à partir de différents terminaux (fixes ou mobiles) de différentes localisations géographiques. Donc, contrairement à la mobilité du terminal, l'utilisateur et le terminal ne forment plus une entité unique, mais la relation entre eux est dynamique [24]. En outre, la relation entre le terminal et le réseau peut être statique (dans le cas où le terminal est fixe), comme elle peut être dynamique (dans le cas où le terminal est mobile). Ainsi, la mobilité de l'utilisateur peut inclure aussi la mobilité du terminal.

Dans ce type de mobilité, les utilisateurs mobiles doivent s'identifier au niveau de chaque terminal afin d'utiliser les services. L'utilisateur mobile doit être muni d'un numéro ou d'un code d'identification personnel plutôt que d'un terminal, ce qui offre plus de flexibilité. Par exemple, dans le projet CESURE [130], les données personnelles de l'utilisateur mobile sont stockées dans une carte à microprocesseur.

La mobilité de l'utilisateur a des répercussions sur le réseau de communication (filaire ou sans fil). En effet, ces réseaux doivent fournir les services suivants [103] :

– identification globale des utilisateurs en utilisant un schéma d'identification unique et sécurisé. Plusieurs travaux sont en cours de standardisation, en particulier le protocole *SIP* (*Session Initiation Protocol*) [66] ;
– adaptation de la présentation des services aux caractéristiques des terminaux pour permettre à l'utilisateur de mieux utiliser les services. Si le terminal n'offre pas les caractéristiques requises par le service, ce dernier peut être offert avec une qualité restreinte. Par exemple, pour afficher les pages Web, les téléphones portables de deuxième génération (cf. section 1.3.2) utilisent le protocole *WAP* (*Wireless Application Protocol*) [171].

1.2.3 Mobilité de la session

La mobilité session offre la possibilité aux utilisateurs mobiles de suspendre une session de travail ouverte sur un terminal et de la réactiver sur un autre terminal [176]. De plus, la suspension/réactivation de la session peut être initiée par l'utilisateur ou faite automatiquement. La session est définie comme une période dans l'espace-temps qui commence par l'activation du terminal ou le lancement d'un programme, qui se poursuit par une authentification et se termine par l'arrêt de ce qui a été démarré. Comme exemple de ce type de mobilité, le *Teleporting* [13]

est un système qui permet aux utilisateurs d'assurer le rapatriement de leurs applications sur n'importe quelle station du réseau. Dans ce système, seuls les affichages sont transférés, les exécutions restant distantes.

La mobilité de la session est généralement couplée soit avec la mobilité du terminal, soit avec la mobilité de l'utilisateur [104], ceci pour s'assurer que la session active n'est pas perturbée ou altérée quand l'unité mobile ou l'utilisateur se déplace [54]. Par ailleurs, la mobilité de la session est intimement liée au concept de *réseau intelligent* [135]. Les réseaux intelligents facilitent l'introduction et la modification de nouveaux services dans les réseaux : le profil de l'utilisateur est sauvegardé dans une base de données à laquelle l'utilisateur peut accéder pour contrôler les services qu'il utilise [7].

1.3 Technologies liées à l'informatique mobile

Cette section fait un état de l'existant en matière de technologies des réseaux et des terminaux. La section 1.3.1 est consacrée à une description rapide des réseaux sans fil. Nous détaillons les différents types de réseaux sans fil ainsi que les différentes technologies de communication. La section 1.3.2 décrit et compare les différents types de terminaux mobiles.

1.3.1 Réseaux sans fil avec infrastructure et *ad hoc*

Un réseau sans fil est, comme son nom l'indique, un réseau dans lequel au moins deux terminaux peuvent communiquer sans liaison filaire. Ces communications sont basées sur une liaison utilisant des ondes radioélectriques (radio, infrarouges ou laser). Nous distinguons plusieurs catégories de réseaux sans fil selon deux aspects : l'architecture du réseau et la technologie utilisée.

Les réseaux sans fil peuvent fonctionner de deux façons différentes [156] : avec infrastructure ou *ad hoc*. Le tableau 1.2 récapitule les différences entre les réseaux sans fil avec infrastructure et les réseaux sans fil *ad hoc*.

	Réseau sans fi l avec infrastructure	Réseau sans fi l *ad hoc*
Communication	Via un point d'accès	Par voisinage
Cellule	Fixe, gérée par la station de base	Auto-confi gurable, dépend de la position des terminaux mobiles
Routage intra-cellule	Via un point d'accès	Par voisinage avec algorithme de routage
Routage inter-cellule	Via plusieurs points d'accès	Nécessite une station relais

TAB. 1.2 – Réseau sans fil avec infrastructure et réseau sans fil *ad hoc*

La figure 1.3 présente l'architecture d'un réseau sans fil avec infrastructure appelé aussi réseau sans fil à point d'accès. Il fait appel à des bornes de concentration appelées des points d'accès, qui gèrent l'ensemble des communications dans une même zone géographique. La portée des communications définit une zone appelée cellule. Les points d'accès sont connectés entre

eux par une liaison filaire ou hertzienne. Les terminaux utilisés peuvent se déplacer au sein de la cellule et changer de cellule, ce qui s'appelle le *handover* (par exemple, le `Terminal 3` dans la figure 1.3). Ce changement de cellule est assuré par un protocole (*handoff protocol*) [163] qui assure automatiquement le transfert de prise en charge d'un point d'accès à un autre tout en conservant la connexion. Ce protocole gère les déconnexions courtes associées au changement de la cellule.

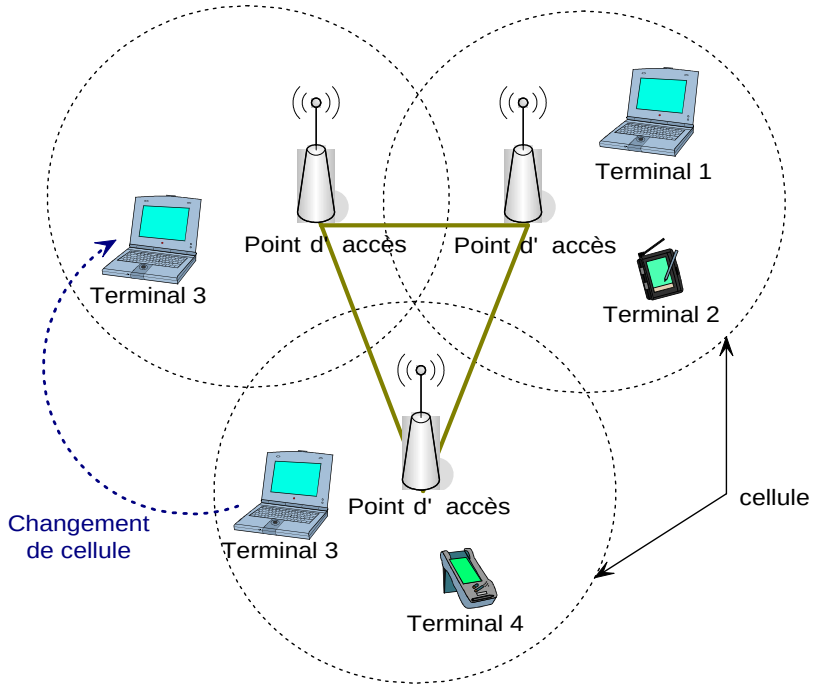


FIG. 1.3 – Architecture d'un réseau sans fil avec infrastructure

La figure 1.4 présente l'architecture d'un réseau *ad hoc* [37]. À l'inverse des réseaux sans fil avec infrastructure, les réseaux sans fil *ad hoc* ne nécessitent pas d'infrastructure de communication fixe. Un réseau *ad hoc* est un réseau sans fil auto-configurable. En effet, lorsque deux terminaux mobiles ou plus se retrouvent dans le même secteur géographique (cellule), ils doivent se reconnaître pour pouvoir s'échanger des données. Le réseau doit se configurer automatiquement pour faire la liaison entre ces terminaux. Il existe deux types de fonctionnement dans les réseaux sans fil *ad hoc*. Dans le premier, le plus simple, les nœuds peuvent échanger des données uniquement lorsqu'ils sont à portée de réception l'un par rapport à l'autre. Dans le deuxième type, lorsque deux machines ne peuvent pas se joindre directement, chaque nœud du réseau peut servir de routeur pour d'autres nœuds. Par exemple, dans la figure 1.4, le `Terminal 1` et le `Terminal 4` peuvent échanger des informations en utilisant le `Terminal 3` comme routeur pour établir la liaison.

Comme mentionné précédemment dans le début de cette section, il existe plusieurs technologies de communication utilisées dans les réseaux sans fil. Ces technologies se distinguent par la fréquence d'émission utilisée ainsi que par le débit et la portée des transmissions [156] :

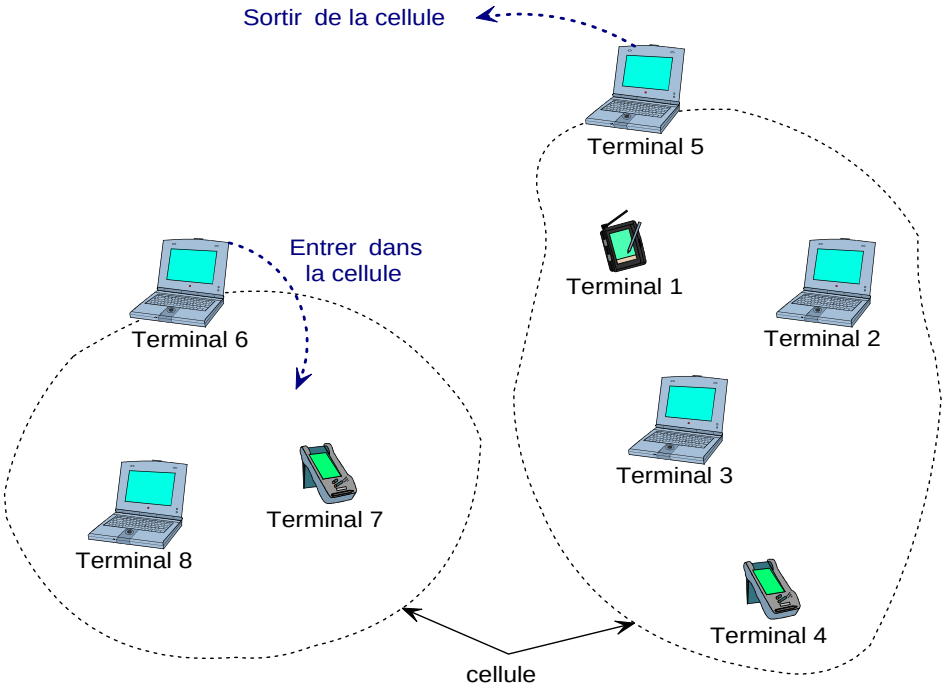


FIG. 1.4 – Architecture d'un réseau sans fil *ad hoc*

1. réseaux personnels sans fil (WPAN, *Wireless Personal Area Network*) : d'une faible portée, ils ont la vocation de relier sur une courte distance, de l'ordre de quelques dizaines de mètres, des équipements (imprimante, appareils domestiques...) soit entre eux, soit avec d'autres réseaux tels que Internet ;
2. réseaux locaux sans fil (WLAN, *Wireless Local Area Network*) : ils offrent des fonctionnalités similaires à celles des WPAN mais qui couvrent une zone plus étendue (université, hôtel, hall d'aéroport...) ;
3. réseaux métropolitains sans fil (WMAN, *Wireless Metropolitan Area Network*) : la portée est de 4 à 10 kilomètres ;
4. réseaux étendus sans fil (WWAN, *Wireless Wide Area Network*) : ce sont les plus répandus. Ils sont utilisés principalement dans la téléphonie mobile.

Nous décrivons dans la suite de cette section les principales technologies de communication qui existent. Nous donnons un récapitulatif de cette étude dans le tableau 1.3.

Bluetooth Lancée par Ericsson en 1994, la technologie Bluetooth propose un débit théorique allant de 57.6 Kb/s à 1 Mb/s pour une portée maximale d'une trentaine de mètres (réseau de type WPAN) utilisant une fréquence de 2.4 GHz [19]. Cette technologie est connue aussi comme la norme `IEEE 802.15.1`. Elle possède l'avantage d'être très peu gourmande en énergie, ce qui la rend particulièrement adaptée à une utilisation au sein de petits périphériques.

HomeRF (*Home Radio Frequence*) La technologie HomeRF est proposée par le *HomeRF Working Group*, qui regroupe des acteurs de l'industrie : Compaq, HP, IBM, Intel et Microsoft. Son débit théorique peut atteindre les 11 Mb/s pour une portée de 50 à 100 mètres (réseau de type WPAN) sur une fréquence de 2.4 Ghz. HomeRF est notamment utilisée pour faire communiquer les téléphones sans fil domestiques avec leur base fixe.

Wi-Fi (*Wireless Fidelity*) Connue aussi comme la norme `IEEE 802.11b`, la technologie Wi-Fi peut être utilisée avec infrastructure ou *ad hoc*. Dans la pratique, Wi-Fi permet de relier tout type de périphérique à un débit qui peut dépasser les 11 Mb/s sur un rayon de plusieurs dizaines de mètres à plusieurs centaines de mètres en environnement ouvert (sans obstacle). Les réseaux Wi-Fi sont de type WLAN et opèrent sur une fréquence de 2.4 GHz.

HiperLAN (*High Performance Radio LAN*) La technologie HiperLAN émet dans la bande des 5 GHz et permet d'atteindre un débit de 54 Mb/s sur une zone d'une centaine de mètres (réseau de type WLAN) comme Wi-Fi et HomeRF. La communication peut se faire directement de station à station (*ad hoc*) ou par l'intermédiaire d'une station centrale (point d'accès).

LMDS (*Local Multipoint Distribution Services*) La technologie LMDS est basée sur une infrastructure de boucle locale utilisant des liens radio point à multipoints sur des distances inférieures à 10 Km (réseau de type WMAN) pour une fréquence allant de 27 à 31 Ghz. LMDS offre une très grande bande passante, identique à celle offerte par la fibre optique qui peut dépasser le 1 Gb/s.

GSM (*Global System for Mobile Communication*) Norme de téléphonie cellulaire numérique européenne développée par l'ETSI (Institut Européen des Normes de Télécommunication), c'est la norme utilisée pour la téléphonie mobile première génération. GSM utilise une fréquence de 900 Mhz et atteint un débit de 9.6 Kb/s. Il existe aussi des versions dérivées du GSM atteignant des fréquences de 1800 ou 1900 Mhz.

GPRS (*Global Packet Radio Services*) La technologie GPRS représente une nouvelle génération pour le standard GSM. C'est la norme de la téléphonie mobile deuxième génération. Elle offre la possibilité de prendre en charge les applications multimédias dans le cadre de la mobilité. GPRS utilise pour le transfert des données la commutation de paquets en utilisant le protocole IP pour le formatage des données. Ce mode de communication permet d'allouer des ressources aux seuls utilisateurs qui ont des données à émettre ou à recevoir. Le GPRS offre un débit pouvant atteindre 170 Kb/s.

UMTS (*Universal Mobile Telecommunications System*) C'est la nouvelle norme de transmission pour les téléphones mobiles de troisième génération. Théoriquement, UMTS peut atteindre des capacités de transmission de 2Mb/s avec la possibilité de faire à la fois de la commutation de circuit (utilisée dans le GSM) et de la commutation de paquets (utilisée dans le GPRS). UMTS opére dans la bande de fréquences de 2GHz.

TETRA (*Terrestrial Trunked Radio*) La technologie TETRA désigne une norme européenne de radiotéléphonie bidirectionnelle destinée à la transmission de la voix et des données. Cette technologie diffère des systèmes publics de téléphonie mobiles (GSM, GPRS et UMTS) surtout par la rapidité d'établissement de la communication, les appels de groupe, les appels prioritaires ou d'urgence, le cryptage de bout en bout et la possibilité de relier directement deux stations mobiles sans passer par une station de base. TETRA utilise plusieurs gammes de fréquences suivant la nature des applications (médicale, militaire. . .) pour une portée qui peut attendre les 14 Km en environnement ouvert. Par exemple, la gamme 380–400MHz est spécialement réservée aux services de secours et de sécurité en Europe [5]. TETRA peut atteindre le débit de 36 Kb/s.

Technologie	Topologie	Réseau	Bande passante	Fréquence
Bluetooth	*Ad hoc*	WPAN	57.6 Kb/s à 1 Mb/s	2.4 GHz
Home RF	Infrastructure	WPAN	11 Mb/s	2.4 GHz
Wi-Fi	*Ad hoc* ou infrastructure	WLAN	11 Mb/s	2.4 GHz
HiperLAN	*Ad hoc* ou infrastructure	WLAN	54 Mb/s	5 GHz
LMDS	Infrastructure	WMAN	1 Gb/s	27 à 31 GHz
GSM	Infrastructure	WWAN	9.6 Kb/s	900, 1800, 1900 Mhz
GPRS	Infrastructure	WWAN	170 Kb/s	200 KHz
UMTS	Infrastructure	WWAN	2 Mb/s	2 GHz
TETRA	*Ad hoc* ou infrastructure	WMAN	36 Kb/s	Plusieurs gammes

TAB. 1.3 – Technologies de communication dans les réseaux sans fil

1.3.2 Terminaux mobiles

Les premiers terminaux mobiles ont été introduits avec la technologie *CT* (*Cordless Telephone*) aux États-Unis et en Europe dans les années 70 pour remplacer les téléphones avec fil. Ces terminaux étaient en réalité le début d'une technologie qui touche non seulement la téléphonie mais aussi les ordinateurs personnels. Nous décrivons dans cette section les différentes catégories de terminaux mobiles qui existent.

Les terminaux mobiles se caractérisent par les ressources offertes, l'encombrement, l'autonomie et les extensions possibles. Ces caractéristiques permettent de choisir les terminaux mobiles à utiliser. Par exemple, un terminal mobile volumineux offre plus de ressources, mais il consomme plus d'énergie qui induit une faible autonomie. À l'inverse, un terminal mobile plus petit et plus léger offre moins de ressources mais permet d'avoir une bonne autonomie. Les extensions représentent un facteur non négligeable de choix d'un terminal mobile : par exemple, lecteur de codes-barres, Wi-Fi, Bluetooth, modem GSM/GPRS. . . Ainsi, plusieurs configurations et architectures peuvent contribuer à améliorer la fiabilité des terminaux mobiles dans le but de satisfaire les exigences des utilisateurs finaux. Sur le marché actuel, les terminaux mobiles se classent en trois catégories : les ordinateurs portables, les ordinateurs de poche ou assistants personnels numériques et les téléphones mobiles.

Les ordinateurs portables offrent des ressources similaires à celles d'une station fixe. La puissance du processeur sur les ordinateurs portables les plus récents peut atteindre les 3 GHz avec

une mémoire entre 128 Mo et 1 Go. Pour se connecter aux réseaux, la plupart des ordinateurs portables disposent de cartes réseaux filaire ou sans fil, intégrées ou amovibles. La technologie *Mobile Intel Centrino* [67] a été conçue pour augmenter l'autonomie des portables (jusqu'à 6h), baisser la température des processeurs et de la carte réseau sans fil. Cependant, le poids, le volume et l'autonomie demeurent les inconvénients majeurs des ordinateurs portables. De plus, les ordinateurs portables sont en général plus chers que les terminaux fixes.

Les ordinateurs de poche ou assistants personnels numériques sont de plus en plus populaires et leur utilisation ne cesse de se simplifier. Du fait de leur taille, ces terminaux peuvent être tenus dans une seule main ou être mis dans une poche, ce qui augmente leur portabilité. Le clavier est remplacé par un stylet et un écran tactile voire même l'utilisation d'un système de reconnaissance vocale tel que *VoCon-3200* [149] ou de reconnaissance d'écriture [3]. Pour se connecter à un réseau, les ordinateurs de poche disposent généralement d'une connexion Wi-Fi, Bluetooth ou GSM (par exemple, le `iPaq h6340` de HP [59]). Concernant les ressources, les ordinateurs de poche disposent de ressources très limitées comparées à celles des ordinateurs portables.

En ce qui concerne les téléphones mobiles, au cours de leur évolution, ils ont connu une miniaturisation importante pour offrir une meilleure portabilité. L'autonomie atteint aujourd'hui plusieurs jours de veille et plusieurs heures d'utilisation. D'un point de vue fonctionnalité, l'évolution des téléphones mobiles est conditionnée par la technologie de communication disponible. La première génération des téléphones portables utilise la technologie GSM pour la transmission de la voix sous forme digitale, mais également de données (SMS, *Short Message Service*). Les téléphones portables de la première génération sont très limités en termes de ressources et de fonctionnalités. La deuxième génération utilise la technologie GPRS qui permet l'envoi de messages multimédias (MMS, *Multimedia Messaging Services*) et de télécharger des jeux sur Internet en utilisant la technologie WAP. La troisième génération, introduite récemment, utilise la technologie UMTS. Elle permet la transmission de voix et de données à une vitesse beaucoup plus grande, ce qui permet de proposer des services tels qu'un accès de bonne qualité à Internet. Les téléphones portables de la troisième génération ne diffèrent pas de ceux de la deuxième génération par la technologie matérielle mis à part le réseau, mais par les logiciels utilisés tels que les systèmes d'exploitation.

1.4 Synthèse

Ce chapitre a été axé sur le concept de l'informatique mobile et l'utilisation des technologies de communication sans fil. L'évolution rapide qu'a connue ce domaine, a permis l'apparition de nouveaux systèmes de communication qui offrent plus d'avantages par rapport aux systèmes classiques. Les nouveaux systèmes n'astreignent plus l'usager à une localisation fixe, mais lui permet une libre mobilité.

Les environnements mobiles sont caractérisés par la variabilité de la bande passante réseau et des restrictions sur les ressources utilisées, surtout si tous les usagers du système sont mobiles. Ces limitations transforment certains problèmes, ayant des solutions évidentes dans l'envi-

ronnement classique, en des problèmes complexes et difficiles à résoudre. Parmi ces problèmes figure le problème des déconnexions qui est le sujet de cette thèse. Les déconnexions peuvent être volontaires ou involontaires, de durée courte ou longue. Dans cette thèse, nous traitons les déconnexions longues volontaires et involontaires. En outre, la mobilité peut être de trois types : mobilité du terminal, mobilité de l'utilisateur et mobilité de la session. Dans cette thèse, nous nous limitons aux déconnexions pour la mobilité du terminal.

La compréhension parfaite du problème des déconnexions nécessite la compréhension des notions de base de la technologie sans fil. Ainsi, nous avons donné dans ce chapitre une brève description des différents réseaux et des technologies de communication sans fil.

Chapitre 2

Intergiciel et séparation des préoccupations

La construction d'applications réparties converge de plus en plus vers l'utilisation d'intergiciels (en anglais, *middleware*). Un intergiciel est une couche logicielle qui se situe entre le système d'exploitation et l'application. Il fournit des solutions réutilisables aux problèmes récurrents des applications réparties tels que la répartition, l'hétérogénéité et la portabilité. Par ailleurs, les méthodologies de conception d'applications réparties considèrent de plus en plus la structuration du code de l'application en préoccupations fonctionnelles et préoccupations extrafonctionnelles. Les préoccupations fonctionnelles sont la réalisation de la logique métier d'un système. Les préoccupations extrafonctionnelles sont indépendantes de la logique métier et idéalement peuvent être ajoutées ou modifiées sans changer la logique métier de l'application.

Ce chapitre situe brièvement le modèle ainsi que le contexte des applications réparties que nous considérons dans nos travaux. Dans un premier temps et après avoir présenté en section 2.1 les principes de base des intergiciels, nous décrivons dans la section 2.2 les deux types d'intergiciels qui reflètent les paradigmes de communication synchrone et asynchrone. Ensuite, nous présentons dans la section 2.3 les travaux de recherche effectués autour de la séparation des préoccupations. Nous synthétisons ce chapitre dans la section 2.4 où nous identifions notamment les limitations des intergiciels vis-à-vis du problème de la gestion des déconnexions.

2.1 Principes des intergiciels

Dans cette section, nous introduisons le fonctionnement des systèmes répartis à base d'intergiciel (cf. section 2.1.1). La structure logicielle d'un intergiciel comprend un ensemble de services appelés services d'intergiciel. Nous présentons les interactions entre ces services dans la section 2.1.2.

2.1.1 Systèmes répartis à base d'intergiciel

Avec l'émergence des réseaux informatiques à large échelle, la construction d'applications repose de plus en plus sur des systèmes répartis. Ces systèmes contribuent à l'amélioration des performances des applications en exploitant les ressources des ordinateurs participants. De plus, les systèmes répartis apportent aux applications un certain niveau de résistance aux défaillances en autorisant la réplication des données et des services. Cependant, la répartition introduit de nouvelles problématiques. D'une part, les développeurs doivent faire face aux problèmes d'hétérogénéité des architectures matérielles et logicielles. D'autre part, les systèmes répartis doivent prendre en compte le changement dynamique des ressources (bande passante, capacité de stockage, puissance de traitement...). La solution à ces problèmes réside dans l'utilisation d'un système réparti à base d'intergiciel.

Bernstein définit l'intergiciel comme « un logiciel présent sur un nœud d'un système réparti, qui offre aux composants applicatifs présents sur ce nœud les abstractions d'un modèle de répartition » [15]. La figure 2.1 schématise la structuration en couches d'un système réparti à base d'intergiciel. Ce dernier se caractérise par un modèle de programmation qui offre une abstraction à l'hétérogénéité et à la distribution des machines sur le réseau tout en utilisant un ensemble de services d'intergiciel. Un service d'intergiciel est défini par un ensemble d'interfaces de programmation (en anglais, API pour *Application Program Interface*) qui reflètent ses interfaces et le protocole qu'il supporte. Chaque service peut avoir plusieurs implantations qui se conforment à ses interfaces et à la spécification du protocole. De plus, un service intergiciel peut être local ou distribué. Dans le premier cas, le service ne peut être utilisé que par les applications ou les services sous-jacents (sur la même machine). L'exemple le plus souvent cité pour faire référence à un service local est le service de journalisation. Concernant les services distribués, ils peuvent être consultés à distance (par exemple, une base de données ou le service de nommage[1]) ou permettent à d'autres services ou applications d'être accessibles à distance (par exemple, le service de communication).

2.1.2 Interactions entre services d'intergiciel

Un service peut être défini par un ensemble d'interfaces, chacune présente un aspect particulier du service. L'accès aux fonctionnalités d'un service se fait à travers ces interfaces. Chaque interface doit définir deux vues : la vue usage et la vue contrat [170]. La vue usage permet de définir les opérations et la structure des données pour l'utilisation du service. La vue contrat définit un contrat entre le fournisseur du service et le consommateur du service. En conséquence, l'utilisation d'un service met en évidence deux interfaces : l'interface serveur fournie par le fournisseur du service et l'interface client fournie par le consommateur du service (cf. figure 2.2). De ces deux points de vue, Bieber et Carpenter définissent un service comme un comportement défini par contrat, qui peut être implanté et fourni par un composant pour être utilisé par un autre composant, sur la base exclusive du contrat [17]. Suivant le type de service (cf. section 2.1.1), le consommateur du service peut être un autre service intergiciel ou une application cliente.

[1] Un service de nommage fournit une association entre un nom et un composant logiciel dans un système réparti.

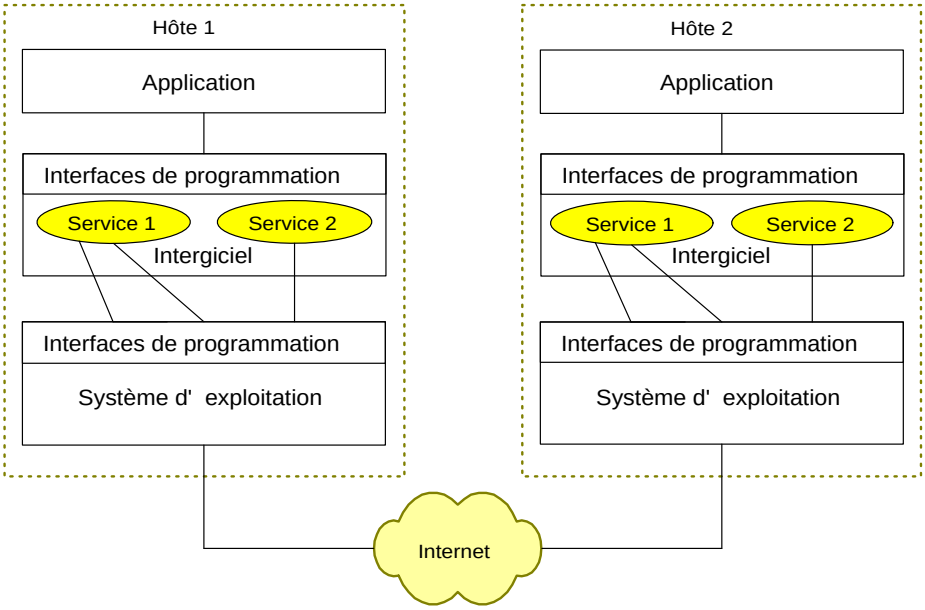


FIG. 2.1 – Couche intergiciel

[16] distingue quatre niveaux de contrat entre une interface client et une interface serveur. Le premier niveau correspond à la description syntaxique des opérations (noms d'opérations, paramètres d'entrées, paramètres de sorties et les exceptions possibles). La vérification de la conformité de ce niveau de contrat peut être effectuée statiquement au moment de la compilation. La description de ce niveau de contrat est effectuée dans un langage de définition d'interface (en anglais, IDL pour *Interface Definition Language*). Actuellement, il n'existe pas un IDL universel, mais chaque intergiciel définit son propre IDL. Par exemple, CORBA *Common Object Request Broker Architecture* [120] de l'OMG (*Object Management Groupe*) propose un IDL indépendant des langages de programmation qui peut être projeté vers plusieurs langages de programmation. Microsoft définit deux langages de description d'interfaces : le SCL (*Service Contract Language*) pour .Net [160], et le MIDL (*Microsoft Interface Definition Language*) pour COM+ [152]. Ces deux langages sont très similaires à l'IDL de CORBA. Nous examinons avec un peu plus de détails CORBA, .Net et COM+ dans les prochaines sections.

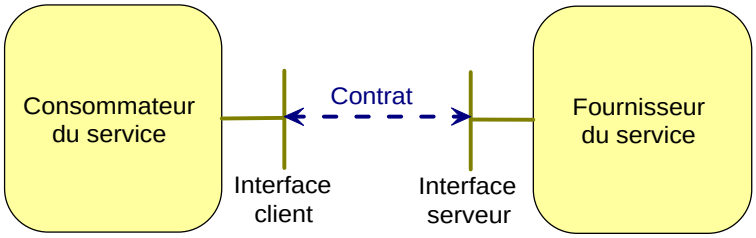


FIG. 2.2 – Relation entre service et interface

Le deuxième niveau de contrat décrit le comportement dynamique des opérations d'une interface. La description de ce niveau de contrat doit offrir un moyen pour vérifier la sémantique de chaque opération de l'interface. Par exemple, la vérification de la sémantique d'une opération peut être effectuée en utilisant des préconditions et des postconditions. Dans UML (*Unified Modeling Language*) [164], la description de la sémantique des opérations se fait via le langage OCL (*Object Constraint Language*) [172]. Cependant, le deuxième niveau de contrat suppose que les services sont atomiques. Cette hypothèse n'est pas toujours vérifiée. En effet, l'exécution d'un service peut donner lieu à des interactions entre plusieurs interfaces de plusieurs services.

Le troisième niveau de contrat considère les interactions dynamiques entre les opérations des interfaces, en spécifiant les contraintes de synchronisation dans l'exécution de ces opérations. Le troisième niveau de contrat permet donc de garantir aux consommateurs du service une exécution correcte du service.

Enfin, le quatrième niveau de contrat définit la qualité de service (en anglais, QoS pour *Quality of Service*). La QoS est généralement liée à des préoccupations extrafonctionnelles telles que la performance, la sécurité, la disponibilité... Le quatrième niveau de contrat inclut aussi la possibilité de négocier la QoS. Cette négociation peut être le sujet d'échanges entre le fournisseur du service et le consommateur du service pendant l'exécution. Elle peut aussi être initiée par un tiers, suite à des changements dans l'environnement d'exécution. Par exemple, dans l'application exemple utilisée dans le projet CARISM [27], dans un service de météorologie et dans le contrat de la QoS de ce service, l'envoi de la carte météo d'une ville peut se transformer en l'envoi d'un simple texte descriptif dans le cas où la bande passante est faible.

2.2 Types d'intergiciels

Un modèle de référence pour la classification des intergiciels a été proposé dans [107]. Dans ce modèle, les intergiciels peuvent être classés suivant : la charge requise, le paradigme de communication ou la représentation du contexte. La charge requise définit les ressources requises par l'intergiciel pour permettre aux applications de s'exécuter. L'intergiciel peut donc être lourd (*heavyweight*) ou léger (*lightweight*). Le paradigme de communication inclut la communication synchrone et la communication asynchrone. Enfin, le contexte d'exécution peut être soit exposé aux applications (*awareness*), soit maintenu caché par l'intergiciel (*transparency*).

Dans notre étude, nous nous basons sur le paradigme de communication pour la classification des intergiciels. Ainsi, dans cette section, nous présentons deux catégories d'intergiciels : les intergiciels synchrones fondés sur la notion de bus logiciel de communication et les intergiciels asynchrones connus aussi sous le nom des MOM (*Message Oriented Middleware*) et qui se basent sur le principe d'envoi, de réception et de notification de messages. Le travail que nous défendons dans cette thèse se base sur les intergiciels synchrones. Ainsi, nous présentons ce type d'intergiciel avec plus de détails dans la section 2.2.1 et nous nous limitons à une brève description des intergiciels asynchrones dans la section 2.2.2. Une étude approfondie des intergiciels asynchrones a été donnée dans [98]. Enfin, dans la section 2.2.3, nous identifions les limitations des intergiciels, en particulier, pour les environnements mobiles.

2.2.1 Intergiciels synchrones

Dans le paradigme de communication synchrone, l'émetteur et le récepteur doivent être prêts à communiquer avant qu'une invocation ne puisse être effectuée. L'exemple le plus souvent cité pour définir le paradigme de communication synchrone est la communication téléphonique. La réalisation de référence de ce paradigme de communication est l'appel de procédure à distance (en anglais, RPC pour *Remote Procedure Call*) [18]. Les RPC se basent sur une architecture client/serveur. Le client appelle une fonction sur un serveur distant et suspend ses activités jusqu'au retour des résultats. Les paramètres sont passés comme lors d'un appel à une procédure locale. Le serveur reçoit l'appel de procédure, lit les paramètres, exécute la procédure et renvoie le résultat au client.

Le principe des RPC est exprimé dans le modèle de programmation orienté objets en introduisant la notion de bus logiciel. Le rôle du bus logiciel est d'enregistrer la localisation des objets, de recevoir, d'accepter et de transmettre les requêtes des composants logiciels. L'architecture d'un bus logiciel est basée sur le patron de conception *Courtier* [23]. Ce patron de conception apporte une solution à la construction des systèmes répartis à base d'intergiciel.

La figure 2.3 illustre le diagramme UML des classes du courtier. Ce dernier fait intervenir six types de participants : clients, serveurs, courtiers, ponts, mandataires des clients (en anglais, *client proxies*) et mandataires des serveurs (en anglais, *server proxies*). Les serveurs implantent des objets qui exposent leurs fonctionnalités à travers des interfaces qui respectent un contrat (cf. section 2.1.2). Les clients sont des objets qui accèdent aux services offerts par les serveurs. Le courtier offre les fonctionnalités minimales et nécessaires aux interactions (requêtes/réponses) entre les clients et les serveurs. Par conséquent, le courtier doit avoir un moyen de localiser le destinataire d'une requête grâce à un identifiant unique. Il offre pour cela une interface de programmation aux clients et serveurs qui inclut des opérations pour enregistrer des serveurs et invoquer des méthodes sur le serveur. Les ponts sont des entités optionnelles permettant l'interopérabilité. Les courtiers peuvent communiquer indépendamment des protocoles de communication (TCP/IP, IPX/SPX...) et des systèmes d'exploitation (Windows, Unix...) utilisés.

Les clients et les serveurs de l'application interagissent au moyen de mandataires, lesquels composent les fonctionnalités offertes par le courtier. Le mandataire du côté client fournit principalement la transparence à la localisation des serveurs. Le mandataire du côté serveur gère principalement le cycle de vie de l'objet : activation, publication de la référence...

La figure 2.4 présente le diagramme UML de séquences d'un client envoyant une requête vers un serveur via le bus logiciel. Ce dernier fait apparaître deux courtiers afin de mettre en valeur l'utilisation des fonctions du courtier lorsque le client et le serveur sont dans des nœuds différents. Quand un courtier reçoit une requête destinée à un serveur maintenu par le courtier local, le courtier passe la requête directement au serveur. Si le serveur est hébergé sur un courtier distant, alors le courtier local recherche une route vers le courtier distant et transmet la requête. Suivant les besoins de l'application et les besoins métiers ciblés, des services additionnels tels qu'un service de nommage et un support d'emballage[2] peuvent être utilisés.

[2] Nous entendons par emballage (en anglais, *marshalling*) la conversion de données de façon invariante de la sémantique vers un format indépendant.

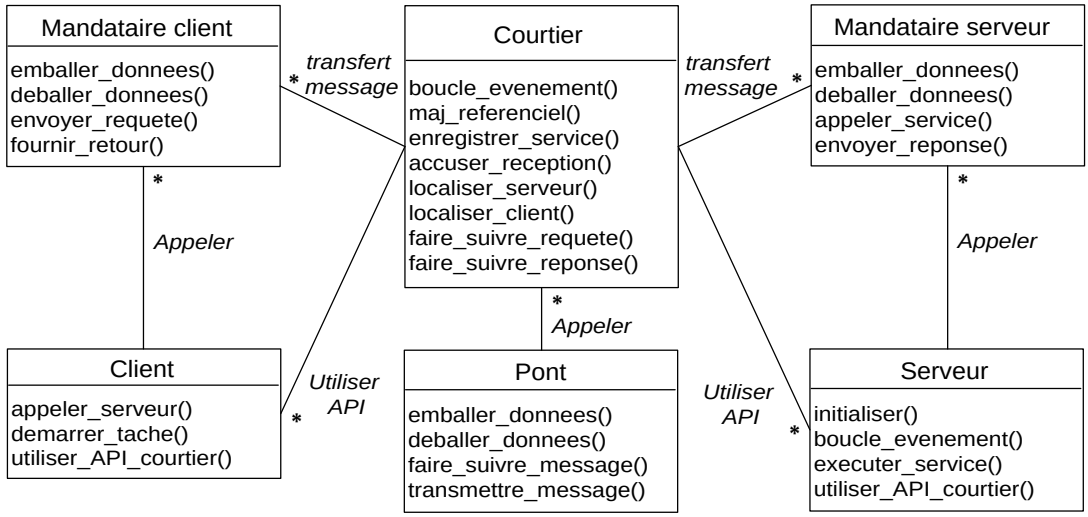


FIG. 2.3 – Diagramme des classes du patron Courtier

Dans la suite de cette section, nous présentons brièvement les principaux intergiciels en illustrant l'utilisation du patron courtier dans chaque intergiciel. Nous commençons par la présentation de CORBA de l'OMG qui sert de base à notre travail. Nous présentons ensuite COM/DCOM de Microsoft et Java RMI de Sun Microsystems.

OMG CORBA

L'OMG [124] est un consortium fondé en 1989, dont le rôle est la standardisation de cadres architecturaux pour intergiciel réparti orienté objet, puis composant. La première spécification publiée par l'OMG est OMA (*Object Management Architecture*) [121]. L'OMA définit le modèle objet utilisé dans toutes les technologies de l'OMG offrant l'interopérabilité d'objets répartis dans des environnements hétérogènes. CORBA représente la deuxième étape des spécifications de l'OMG [120]. Il définit l'architecture du courtier appelé ORB (*Object Request Broker*). Dans sa version 1.1, CORBA définit le langage de définition d'interface IDL (*Interface Definition Language*). Il permet de décrire le service fourni par un objet indépendamment de sa mise en œuvre. L'OMG fait de l'interopérabilité sa priorité dans la version 2 de CORBA définissant le protocole IIOP (*Internet Inter-ORB Protocol*). En plus de l'interopérabilité, de nombreux services intergiciels sont ajoutés. Ces services normalisés par l'OMG sont souvent réalisés sous forme de bibliothèques associées au code lors de la compilation, tels que le service de nommage que nous avons déjà évoqué, le service de courtage qui permet de classer et chercher les services en fonction d'attributs recherchés ou encore le service de cycle de vie qui prend en charge la création, le déplacement ou la destruction des objets qu'il gère. Les descriptions des services CORBA peuvent être trouvées dans [56].

La spécification CORBA a donnée lieu à plusieurs variantes comme une spécification pour la gestion des déconnexions courtes (cf. section 1.1.2) appelée Wireless CORBA [126] et une spécification destinée au temps réel appelée RT-CORBA [150]. Toutes ces spécifications offrent

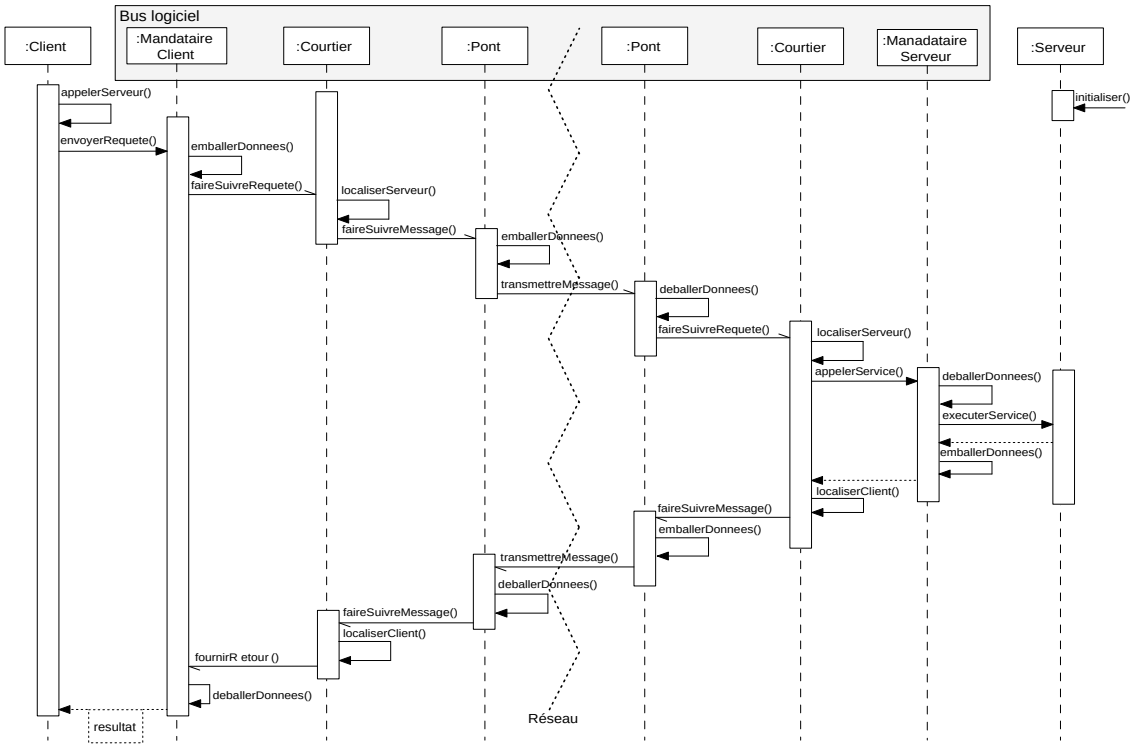


FIG. 2.4 – Diagramme de séquences du courtier

un cadre d'utilisation des plus larges, des applications réparties les plus simples jusqu'à la gestion de la qualité de service [167] ou à la personnalisation des services CORBA pour le domaine des télécommunications [142]. Les travaux actuels de l'OMG sur les intergiciel portent sur CCM, le modèle de composant CORBA (*CORBA Component Model*) [29]. CCM est la principale nouveauté de la version 3 de CORBA. Nous présentons CCM dans la section 2.3.2.

Microsoft COM/DCOM

Le modèle COM/DCOM est une contribution de Microsoft aux environnements à objets répartis [21]. COM (*Component Object Model*) vise à permettre l'interaction d'objets hétérogènes en termes de langages de programmation. Ces besoins ont été résolus par les technologies OLE (*Object Linking and Embedding*), basées sur les librairies dynamiques (DLL). OLE est utilisé pour gérer l'échange d'informations entre applications sous Windows. Dans sa première version, OLE cible la gestion de documents composés, par exemple, des fichiers contenant à la fois du texte et des dessins. C'est à partir de la version 2 de OLE qu'est introduit un modèle objet générique pour les applications qui s'exécutent sur une même machine. Rien dans la structure du modèle OLE 2 n'interdisait que les applications soient réparties. Ainsi, la réalisation de l'intergiciel de Microsoft nommé DCOM (*Distributed COM*) a pu voir le jour. Dans cet intergiciel, la technologie COM/DCOM représente la partie support de communication qu'on pourrait assimiler à l'ORB dans CORBA.

La technologie COM/DCOM reconnaît trois modes d'exécution de serveurs : les DLL, les objets locaux partagés par des applications s'exécutant localement et les objets distants. Le modèle COM/DCOM définit un schéma d'accès, basé sur des interfaces, commun aux trois types de serveurs. Dans ce cadre, COM est l'implantation supportant les communications locales, DCOM offre le support d'accès aux serveurs distants. Ainsi dans COM, les clients accèdent aux objets par l'intermédiaire de contrats qui sont réalisés par les interfaces offertes par l'objet. De plus, le dialogue client/serveur se fait suivant trois politiques. Dans le cas où le client et le serveur appartiennent au même espace d'adressage, COM charge le code du serveur et donne au client un pointeur offrant l'accès direct au serveur. Si le serveur fait partie d'un espace d'adressage autre que celui du client mais local à la machine, le dialogue se fait par des appels de méthodes locaux LRPC (*Lightweight Remote Procedure Call*). Enfin, si le serveur fait partie d'un espace d'adressage distant, DCOM utilise des RPC orientés objet (ORPC).

Sun Java RMI

Java RMI (*Remote Method Invocation*) est l'intergiciel pour objets répartis de Sun Microsystems pour la plateforme J2EE [69]. Il offre un moyen de communication entre deux objets Java localisés dans deux machines virtuelles distinctes. Java RMI implante le modèle client/serveur dans lequel le client est soit une *applet*, soit une application Java et le serveur est un objet appartenant à une application Java. Java RMI utilise un serveur de noms (*registry*), basé sur le standard JNDI (*Java Naming and Directory Interface*) [74]. Le nommage utilisé par ce serveur repose sur le système des URL (*Uniform Resource Locator*). Java RMI ne supporte pas l'hétérogénéité du langage, puisque seuls des objets Java peuvent interagir. Bien évidemment, il permet l'interopérabilité entre des objets Java s'exécutant sur des machines virtuelles de fournisseurs différents.

Le modèle d'exécution dans Java RMI respecte le patron du courtier. Toute interface distante étend l'interface `Remote`, ce qui permet de spécifier directement dans le langage les méthodes accessibles à distance sans utiliser d'IDL. La mise en œuvre du serveur doit hériter de `RemoteObject` et implanter cette interface. Un générateur de code est chargé de créer à la volée des mandataires client et serveur pour traiter les communications. Les objets Java qui constituent les paramètres des invocations de méthodes distantes sont passés par valeur, à l'aide d'un mécanisme de sérialisation. Quand la référence d'un objet distant est passée en paramètre, un mécanisme basé sur HTTP permet au récepteur de récupérer l'objet référencé sur le site d'origine (simulant ainsi un passage par référence).

À partir du JDK 1.3, Sun a introduit dans RMI la possibilité d'utiliser IIOP comme format d'emballage (fonctionnalité appelée RMI/IIOP) pour répondre aux besoins d'interopérabilité avec CORBA. L'utilisation du mode RMI/IIOP repose sur les librairies d'un ORB CORBA et rend disponible la plupart des fonctionnalités CORBA à des applications RMI. Des outils assurent aussi la conversion d'interfaces Java vers l'IDL CORBA. Java RMI étend le mécanisme de ramasse-miettes aux invocations distribuées et le développeur n'a pas besoin de soucier de la destruction des objets.

Le modèle de composant de Java repose sur la notion de *bean*. La spécification des composants repose sur deux modèles : les *beans* et les EJB (*Entreprise Java Beans*) [113]. Nous décrivons le modèle de composant de J2EE dans la section 2.3.2.

2.2.2 Intergiciels asynchrones

Par définition, la communication asynchrone ne nécessite pas une présence permanente d'un lien de communication entre deux entités communicantes. Le système réparti est donc constitué à partir d'entités réparties faiblement couplées. Les intergiciels asynchrones (en anglais, MOM pour *Message Oriented Middleware*) se basent sur ce paradigme de communication. Ils offrent aux applications la possibilité de communiquer par échange de messages. Les intergiciels asynchrones reposent sur un modèle de programmation basé sur la génération, l'envoi, l'observation et la notification des messages[3]. Ainsi, les interactions entre les composants du système se font à base de messages ou d'évènements. Ce modèle distingue deux types d'entités : l'émetteur qui produit des messages et le récepteur qui consomme des messages (cf. figure 2.5).

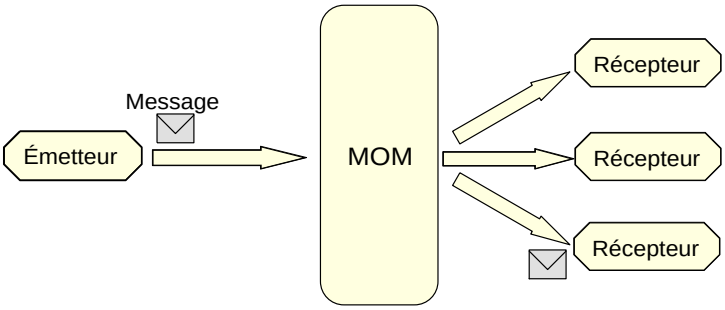


FIG. 2.5 – Intergiciel orienté message

Contrairement aux intergiciels synchrones où l'architecture est de type point à point, il existe trois types d'architecture d'intergiciel asynchrone : l'architecture centralisée, l'architecture répartie en point à point ou partiellement maillée, et l'architecture répartie en bus. Dans l'architecture centralisée, le MOM est constitué d'un site principal qui distribue tous les messages. Cette centralisation rend l'implantation de ce type d'architecture très simple. Cependant, cette architecture engendre des problèmes de fiabilité et de passage à l'échelle. L'architecture partiellement maillée traite les problèmes de la première architecture. L'intergiciel est réparti sur plusieurs sites. Dans ce cas, l'émetteur envoie le message au site auquel il est enregistré et ce dernier soit transmet le message directement au destinataire s'il est directement accessible, soit transmet le message à un site intermédiaire qui fait suivre le message.

Par ailleurs, les modèles usuels de communication sont la communication par envoi de messages (en anglais, *message passing*), par file de messages (en anglais, *message queuing*), évènementiel, par espace partagé (en anglais, *shared spaces*) et par abonnement (en anglais, *publish/subscribe*) [44]. La communication par envoi de messages est la forme de base des com-

[3] La spécifi cation sémantique de l'évènement par rapport au message ne présente pas d'intérêt pour la communication, les deux termes dans le texte sont interchangeables.

munications asynchrones. Dans ce modèle, le producteur et le consommateur du message sont couplés dans le temps et dans l'espace. Ils doivent tous les deux être en activité en même temps et le destinataire d'un message doit être connu par l'expéditeur. Ce modèle peut être assimilé à la communication synchrone non-bloquante. La communication par file de messages ajoute la notion de fiabilité au modèle précédent par l'utilisation d'une file de messages au niveau de l'intergiciel. La file de messages joue le rôle de messager entre l'émetteur et le récepteur. En outre, elle assure l'intégrité des messages dans le cas où les récepteurs sont injoignables. La communication avec file de messages est très utilisée dans les intergiciels asynchrones actuels, tels que MSMQ de Microsoft [138] intégré dans la plupart des plateformes Windows et MQSeries d'IBM [114].

Le modèle évènementiel consiste à préparer le destinataire à exécuter une tâche si un évènement particulier se produit. Ce type de communication est utilisé par exemple dans les interfaces graphiques de Java (AWT) ou dans les bases de données avec la notion de déclencheur (en anglais, *trigger*). Le modèle de communication par espace partagé est généralement assimilé à la notion de *tuple* du langage Linda [55]. Un tuple est un vecteur de données typées. Un espace de tuples se compose d'une collection de tuples dans une mémoire partagée accessible à tous les participants. La communication entre les participants se fait par l'insertion, la lecture et la suppression des tuples. Ce modèle de communication fournit un découplage dans l'espace et dans le temps avec en plus l'anonymat entre les producteurs et les consommateurs. Cependant, il ne fournit pas le découplage dans la synchronisation puisque le producteur pose ses tuples en mode synchrone. Cet inconvénient est résolu dans les nouveaux systèmes à base de tuple tels que JavaSpaces [71] et TSpaces [99].

Enfin, le modèle de communication par abonnement, le plus récent des modèles de communication utilisés dans les intergiciels asynchrones, se base sur le patron de conception observateur [23]. Dans ce type de communication, les émetteurs publient des évènements, les récepteurs s'abonnent à certaines catégories d'évènements, puis consomment les évènements correspondants. L'abonnement à un canal d'évènements se fait suivant trois principaux critères : l'abonnement basé sur le sujet de l'évènement (en anglais, *topic-based publish/subscribe*), l'abonnement basé sur le contenu de l'évènement (en anglais, *content-based publish/subscribe*) et l'abonnement basé sur le type de l'évènement (en anglais, *type-based publish/subscribe*) [44].

2.2.3 Limitations des intergiciels

Le développement d'applications réparties s'avère très simplifié lorsque celles-ci sont construites au-dessus d'un intergiciel. En effet, l'intergiciel gère les problèmes d'hétérogénéité et d'interopérabilité, de manière transparente pour le développeur d'applications. De plus, ils proposent en général un ensemble de services qui facilitent la construction des applications. Toutefois, les intergiciels sont exposés à plusieurs limitations. Bien que les intergiciels offrent l'interopérabilité aux applications sous-jacentes, l'interopérabilité entre les intergiciels tels que CORBA et COM/DCOM prend une dimension toute particulière. Dans ce cas de figure, l'interopérabilité ne se limite pas aux systèmes d'exploitation et aux langages de programmation, mais également aux modèles de répartition. Cette problématique est connue sous l'acronyme M2M (*Middleware To*

Middleware) [9]. L'interopérabilité entre intergiciels représente donc un nouvel axe de recherche pour lequel plusieurs solutions voient le jour. Citons parmi elles la solution MDA (*Model Driven Architecture*) de l'OMG (cf. section 2.3.1) et Jonathan [75], un intergiciel générique basé sur la notion de personnalité pour fournir une base commune pour la construction des ORB à partir d'un canevas logiciel modulaire et paramétrable.

Comme nous l'avons vu dans le chapitre 1, les besoins de l'informatique mobile sont considérables comparés à ceux de l'informatique traditionnelle. La plupart des solutions proposées pour faire face aux besoins de l'informatique mobile adoptent une approche radicale [107]. Dans ces approches, les caractéristiques des environnements mobiles sont généralement transparentes à l'intergiciel et c'est l'application qui prend en charge l'adaptation à ces caractéristiques.

2.3 Séparation des préoccupations

La séparation des préoccupations est considérée comme l'une des approches les plus prometteuses du génie logiciel. Selon cette approche, une application contient deux parties distinctes : les préoccupations fonctionnelles et les préoccupations extrafonctionnelles (aussi appelées non fonctionnelles dans la littérature). Les préoccupations fonctionnelles sont celles qui réalisent le service de base rendu par l'application. Par exemple, dans une application d'achat électronique par Internet, une des fonctionnalités est « acheter un billet d'avion ». Les préoccupations extrafonctionnelles représentent la partie qui adapte les préoccupations fonctionnelles à un environnement et/ou à une utilisation particulière. Dans l'exemple précédent, cela peut être d'assurer l'exécution d'une opération d'une manière transactionnelle.

Cependant, le code traitant les préoccupations extrafonctionnelles est généralement complexe et parfois disséminé dans tout le code source de l'application, ce qui rend la lisibilité et la réutilisation des parties fonctionnelle et extrafonctionnelle très compliquées. La séparation des préoccupations est donc primordiale pour la simplification, la maintenance et la réutilisation des préoccupations fonctionnelles d'une application.

Dans cette section, nous présentons les différentes techniques de séparation des préoccupations. Nous avons choisi d'ordonner ces techniques suivant le cycle de développement (modélisation, programmation et maintenance). La section 2.3.1 donne une brève description de MDA dont l'objectif est de séparer les préoccupations dès la phase de modélisation de l'application. La section 2.3.2 présente le paradigme composant/conteneur qui sépare les préoccupations au moment de la conception de l'application. Ensuite, les sections 2.3.3 et 2.3.4 présentent deux techniques, respectivement réflexivité et programmation orientée aspects, qui opèrent au niveau programmation. Enfin, nous présentons dans la section 2.3.5 une technique de séparation des préoccupations basée sur le mécanisme d'interception qui opère au niveau exécution.

2.3.1 Architecture basée sur les modèles

Les intergiciels tels que CORBA, EJB et .Net proposent des abstractions pour définir des applications indépendamment des systèmes ou des langages de programmation. Toutefois, il

n'existe pas d'intergiciel universel qui définit un niveau d'abstraction supérieur. L'utilisation de l'ingénierie dirigée par les modèles[4] permet d'atteindre ce niveau d'abstraction. De plus, il offre la possibilité de séparer les préoccupations dès la modélisation. Nous présentons dans cette section, MDA, une vision de l'OMG pour la modélisation de l'architecture [108]. MDA est le modèle que nous utilisons dans notre approche de conception d'application orientée déconnexion.

La figure 2.6 tirée de [111] représente le méta-modèle de MDA. La spécification d'un système dans MDA se base sur deux principaux modèles : le modèle indépendant des plateformes (en anglais, PIM pour *Platform Independant Model*) et le modèle dépendant d'une plateforme (en anglais, PSM pour *Platform Specific Model*). Les PIM fournissent les spécifications de la structure et du comportement d'un système en faisant abstraction des détails techniques et des considérations technologiques. Les PSM décrivent la structure et certaines[5] préoccupations fonctionnelles d'un système de manière spécifique à la plateforme visée. Les PSM servent essentiellement de base à la génération du code vers la plateforme visée. De plus, l'intégration des préoccupations extrafonctionnelles se fait dans les PSM. Chaque modèle (PIM ou PSM) est basé sur un méta-modèle exprimé en un ou plusieurs langages tels que UML et XMI [175].

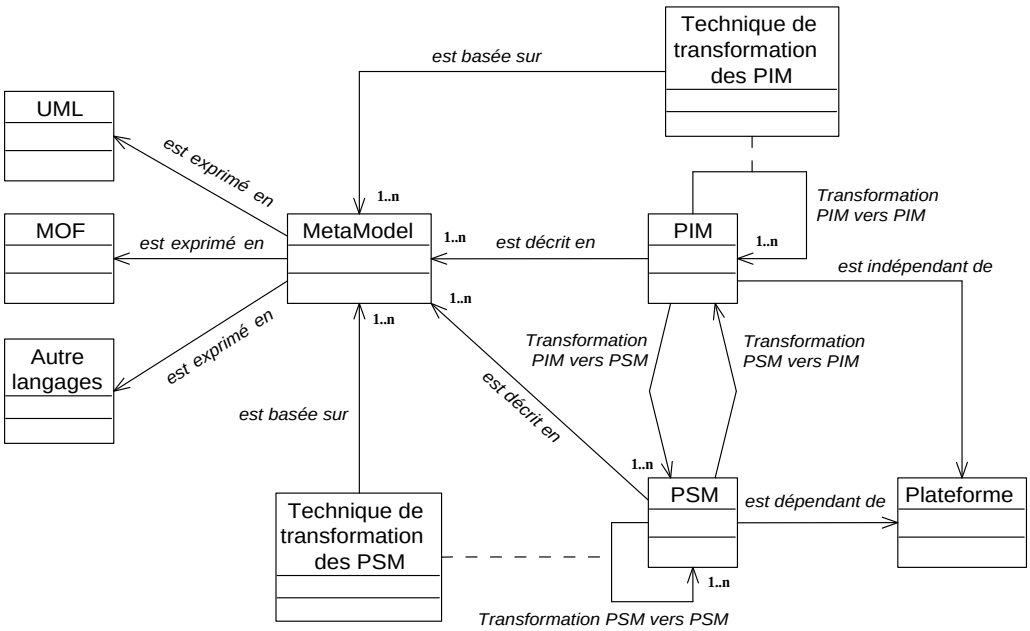


FIG. 2.6 – Méta-modèle de MDA

Une fois établi, le PIM est projeté vers une plateforme spécifique pour définir un PSM. C'est la notion de transformation dans MDA. Une transformation regroupe un ensemble de règles et de techniques pour transformer un modèle en un autre. MDA définit quatre types de transformation (cf. figure 2.6). Premièrement, la transformation PIM vers PIM permet d'étendre ou de spécialiser un modèle sans ajout d'information dépendant de la plateforme. Ce type de transformation

[4]Un modèle représente tout ou partie d'une fonctionnalité, de la structure ou encore du comportement d'un système. La spécifi cation d'un modèle se fait généralement au moyen d'une combinaison entre des langages textuels tels que XML et OCL, et des langages graphiques tels que UML.

[5]Celles qui sont réalisées par l'intergiciel cible.

est en général liée à l'enrichissement de modèles. Deuxièmement, la transformation PIM vers PSM est utilisée dès lors qu'un PIM est suffisamment enrichi pour être projeté vers la plateforme cible. Troisièmement, la transformation PSM vers PIM sert à abstraire les modèles d'implantation existants dans une technologie particulière vers un modèle indépendant de la plateforme. Cette transformation permet aussi d'intégrer des applications existantes afin de pouvoir les utiliser dans une modélisation MDA. Enfin, la transformation PSM vers PSM autorise le raffinement des PSM. Cette dernière transformation est aussi utilisée pour l'intégration des préoccupations extrafonctionnelles spécifiques à la plateforme visée.

Les travaux de l'OMG sur MDA se découpent en quatre activités principales qui se réfèrent à des standards déjà adoptés par l'industrie ou en cours de normalisation. L'activité de base de MDA est la modélisation de la logique métier de l'application en se basant sur les technologies UML, MOF [112] et CWM [38] spécifiées par l'OMG. La deuxième activité consiste à la spécialisation des modèles métiers dans des technologies intergiciels telles que CORBA, EJB, .NET et les services web. Cette spécialisation se base sur la notion de profil UML. Un profil UML est une forme de patron introduisant des stéréotypes précisant le sens d'une classe ou d'une relation dans les diagrammes UML. Différents profils UML sont standardisés comme le profil UML pour CORBA [123] et le profil UML pour EJB [154], ou en cours de standardisation comme le profil EDOC [41] et le profil UML pour CCM [125]. L'approche de développement d'applications orientées déconnexions que nous présentons dans cette thèse se base sur la notion de profil UML, en particulier, le profil UML pour CCM (cf. chapitre 4).

La troisième activité concerne les services d'intergiciel. L'OMG a repris pour les services d'intergiciel de MDA ceux qui avaient été définis pour CORBA : concurrence, évènement, notification, nommage, persistance. . . . Le défi actuel de l'OMG est de décrire ces services pour qu'ils puissent être utilisés dans un modèle PIM. La gestion de déconnexions longues dans les environnements mobiles que nous traitons dans cette thèse n'est pas encore abordée par l'OMG. Enfin, la dernière activité de l'OMG concerne les services métiers. Elle se base sur les profils UML et permet de proposer des canevas logiciels spécifiques à un domaine d'application : espace, télécommunication, mécanique, médecine. . . .

2.3.2 Paradigme composant/conteneur

Il n'existe pas aujourd'hui une définition unique de composant logiciel. Toutefois, plusieurs définitions ou caractéristiques sont acceptées. Szyperski définit un composant comme « une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites sur son contexte. Un composant peut être déployé indépendamment et il est sujet à des compositions par des parties tiers » [155]. L'architecture logique d'un composant illustrée dans la figure 2.7 est caractérisée par deux parties [170]. 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, par exemple, 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 préoccupations extrafonctionnelles.

Le conteneur contrôle toutes les interactions du composant avec le monde extérieur. Ce contrôle est réalisé à travers des entités appelées objets de contrôle. Dans la plupart des modèles de composants, le conteneur offre au minimum deux objets de contrôle, un intercepteur pré-invocation et un intercepteur post-invocation. L'intercepteur pré-invocation 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 post-invocation est un objet de contrôle interne (local) ; il n'est visible que de l'intérieur du conteneur. Le conteneur peut contenir d'autres objets de contrôle. Chaque objet de contrôle peut représenter 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.

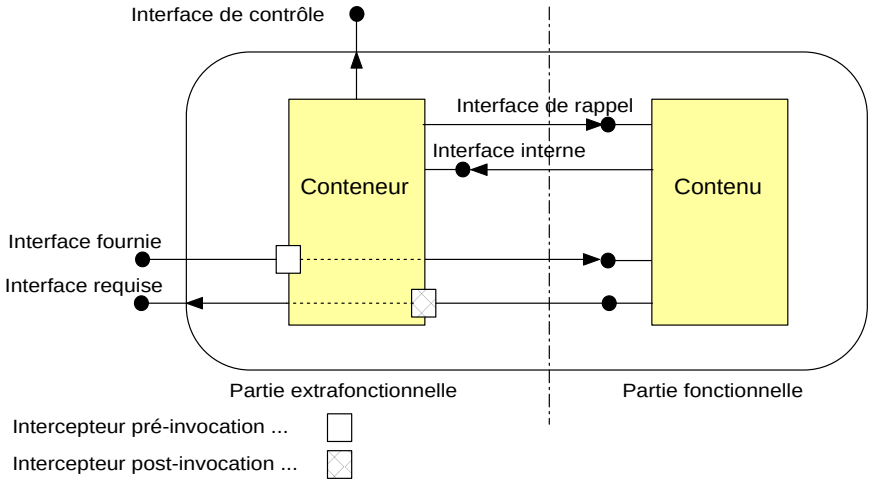


FIG. 2.7 – Structure logique d'un composant

Dans la suite de cette section, nous présentons brièvement les principaux modèles de composants logiciels utilisés dans la recherche et l'industrie à savoir J2EE, CCM et Fractal. Nous nous focalisons principalement sur les interactions entre le composant et son conteneur.

J2EE

J2EE [69] regroupe un ensemble de technologies de Sun pour le développement d'applications s'appuyant sur le langage Java. La plateforme J2EE est essentiellement un environnement fournissant trois types de composant : *JavaBeans* pour les composants des interfaces client, *Java Server Pages* (JSP) pour les composants des serveurs Internet et *Entreprise JavaBeans* (EJB) pour les composants des serveurs d'applications. Dans la suite de cette section, nous présentons la description des composants EJB.

Les EJB représentent un canevas de composants serveurs pour concevoir des applications réparties trois tiers. Le modèle de composant EJB distingue trois types de composants : session, entité et à message. Au sein des applications, les EJB de type session (composant à état temporaire) et entité (composant à état persistant) sont accessibles à travers une communication synchrone basée sur Java RMI (cf. section 2.2.1). Les EJB à messages (composant sans état) permettent d'intégrer un service d'événements dans les EJB en utilisant l'API JMS [73].

EJB distingue trois catégories d'interfaces : les interfaces de rappel, les interfaces internes et les interfaces externes (cf. figure 2.8). Certaines interfaces dépendent de la catégorie du composant. Un EJB présente deux interfaces à ses clients : une interface maison (*home*) et une interface distante. L'interface maison est utilisée pour la création, la recherche dans le cas des composants entités et la suppression de l'instance de composant. Elle offre des méthodes pour la gestion des méta-données du composant (par exemple, `getEJBMetaData()`). L'interface distante offre un accès aux méthodes métiers du composant EJB.

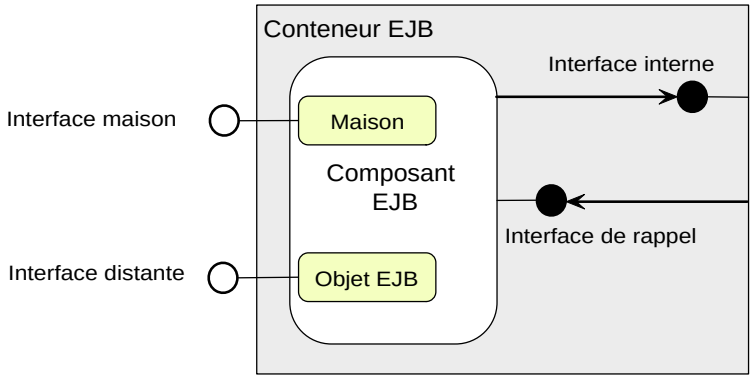


FIG. 2.8 – Composant EJB

Dans EJB, les conteneurs peuvent héberger plusieurs instances de plusieurs types de composants EJB. Ils gèrent les transactions, la sécurité et la persistance selon trois types de configurations prédéfinies correspondant aux trois types de composants. La structure du conteneur EJB respecte le modèle décrit dans le début de cette section. Il est constitué d'objets de contrôle qui interceptent (avant et après) les invocations sur les composants. En outre, les conteneurs sont inclus dans des environnements d'exécution, les serveurs EJB, qui fournissent un accès aux services intergiciel. Ces services sont ceux typiquement requis par les systèmes d'information : transactions, sécurité, persistance, distribution et messagerie. Ainsi, la gestion des déconnexions qui fait l'objet de cette thèse n'est pas prise en compte par EJB.

CCM

CCM [29] est le modèle de composant de l'OMG qui fait partie de la norme CORBA 3. CCM représente un modèle de référence fortement inspiré de EJB. De plus, un composant CCM peut être un client ou un serveur, contrairement à EJB qui est purement un canevas de composants

serveurs. La spécification de CCM définit quatre modèles : le modèle abstrait, le modèle de programmation, le modèle de déploiement et le modèle d'exécution.

Le modèle abstrait définit les caractéristiques d'un composant CCM. Ce dernier 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 2.9). Ces caractéristiques sont décrites dans le langage OMG IDL3, une extension du langage OMG IDL spécifié dans CORBA. En outre, le modèle abstrait de CCM définit la notion de maison comme dans les EJB. Une maison de composants est un gestionnaire pour des instances d'un même type de composant. Elle gère le cycle de vie des instances, éventuellement en leur associant des clés primaires dans le cas des instances de composants persistantes. Pour cela, la maison offre une fabrique d'instances et des opérations de recherche et de suppression utilisant ces clés.

La spécification CCM définit quatre catégories de composants : service, session, processus et entité. Les composants service sont des composants sans état et sans identité. Leur durée de vie correspond à la durée de traitement de l'invocation d'une opération. Les composants session ont un état volatile et une identité non persistante. La durée de vie d'un tel composant est liée à la session d'utilisation de ce dernier. Enfin, l'état des composants processus et entité est persistant. La différence entre les composants processus et entité réside dans l'utilisation de clés primaires pour l'identification des instances entités (sans clé pour les composants processus). En outre, la gestion de la persistance est visible pour le client dans le cas des composants entité et transparente dans le cas des composants processus.

Le modèle de programmation offre aux développeurs[6] de composants le moyen de produire l'implantation d'un composant. Ainsi, CCM définit le langage CIDL (en anglais, *Component Implementation Definition Language*) utilisé pour décrire la structure d'implantation d'un composant CCM. L'utilisation de ce langage est associée à un canevas, le CIF (en anglais, *Component Implementation Framework*), qui définit comment les parties fonctionnelles et extrafonctionnelles doivent coopérer. Il spécifie aussi les interactions entre un composant et un conteneur, et fournit une interface pour l'introspection dynamique des composants au moment de l'exécution. Un autre avantage de CCM est que l'implantation d'un composant peut être monolithique ou segmentée. Pour la deuxième approche, CCM génère pour chaque segment un squelette. Les segments sont activés indépendamment et possèdent un état et une identité.

Le modèle de déploiement définit un processus qui permet d'installer une application CCM de manière simple et automatique sur un réseau de machines hétérogènes. Ce modèle s'appuie sur l'utilisation de paquetages de composants. Chaque paquetage est décrit dans un descripteur de paquetage (en anglais, CSD pour *CORBA Software Descriptor*). Quant à la composition des composants, ils sont décrits dans un descripteur d'assemblage de composants (en anglais, CAD pour *Component Assembly Descriptor*). Le CAD définit également les sites d'installation des composants.

Le modèle d'exécution définit l'environnement d'exécution des instances de composants représentées par le conteneur. Comme dans EJB, le conteneur CCM s'exécute dans un serveur qui

[6] Dans le cadre des applications à base de composants, nous considérons six acteurs principaux : architecte logiciel, développeur de composants, intégrateur de composants, *packageur* de composants, déployeur de composants et administrateur d'application. D'autres acteurs peuvent intervenir dans des domaines spécialisés.

fournit les services extrafonctionnels nécessaires. Étant donné que CCM est défini comme une couche au-dessus de l'ORB CORBA, les services proposés sont les services de transactions, de sécurité, de persistance, de cycle de vie et des communications synchrones et asynchrones. La gestion des déconnexions qui fait l'objet de cette thèse n'est pas prise en compte par CCM.

Le conteneur CCM définit deux types d'interfaces de contrôle externes : les interfaces d'introspection et les interfaces de gestion de ports (cf. figure 2.9). L'interface d'introspection fournit des informations sur le type de composant, sur ses interfaces fournies (facettes et sources d'évènements) et requises (réceptacles et puits d'évènements), ainsi que l'état de ses connexions à d'autres composants. L'interface de gestion des ports est utilisée pour établir ou détruire des connexions entre composants.

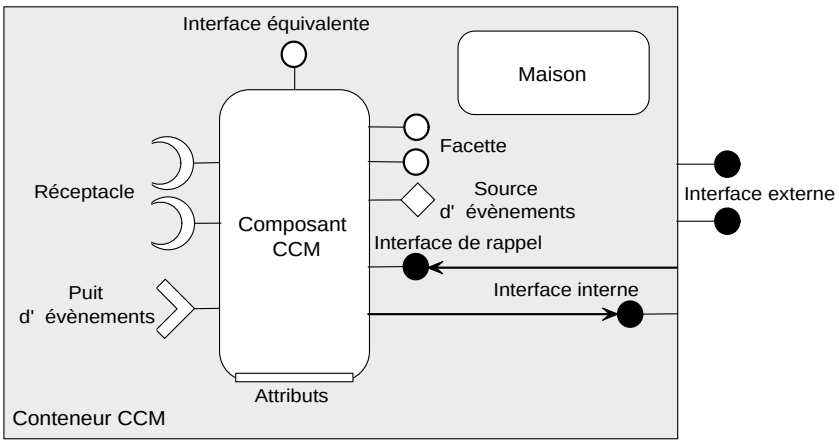


FIG. 2.9 – Composant CCM

Malgré la jeunesse de la spécification, CCM dispose toutefois de mises en œuvre. Elles comptent plusieurs implantations en source libre telles que OpenCCM [127], MicoCCM [109] ou Qedo [137] et commerciales, comme EJCCM [42] ou K2 CCM [78]. Les travaux de recherche sur le conteneur CCM tournent autour de la définition d'un conteneur adaptable et (re)configurable. Un premier prototype de conteneur, dit « ouvert », a été proposé dans le cadre de la plateforme OpenCCM [166]. Le conteneur ouvert définit une chaîne complète pour la description, la production, le déploiement et l'administration des conteneurs CCM. Il est composé de trois couches : une couche d'interception, une couche de coordination et une couche de contrôle. Ce prototype a été raffiné dans le cadre du projet IST COACH [33], pour aboutir à un modèle de conteneur extensible (en anglais, ECM pour *Extensible Container Model*) [165]. ECM est le modèle de conteneur que nous avons utilisé dans notre réalisation. Nous le décrivons dans le chapitre 5.

Fractal

Le modèle de composant Fractal [50] permet la définition, la configuration, et la reconfiguration dynamique de composants. Construit comme un modèle de haut niveau, son objectif est de fournir une grande modularité et des possibilités d'extension étendues. Contrairement à CCM et EJB,

le modèle de composant Fractal est récursif : un composant est de type primitif ou composite. Dans ce dernier cas, le composant correspond à un assemblage d'autres composants primitifs ou composites. Un composant peut également être partagé entre différents composites. Dans la figure 2.10, le composant `Composant 4` est partagé entre deux composants composites : le premier est composé des composants `Composant 1` et `Composant 4`, et le deuxième des composants `Composant 2` et `Composant 4`.

Un composant Fractal est formé de deux parties : le conteneur appelé contrôleur ou membrane et le contenu (cf. figure 2.10). Le contenu est composé de un ou plusieurs composants (primitifs ou composites) qui sont sous le contrôle de leur contrôleur. Le contrôleur expose deux catégories d'interfaces : les interfaces externes, accessibles de l'extérieur du composant, et les interfaces internes, accessibles uniquement par les sous-composants. De plus, les interfaces peuvent être fonctionnelles ou de contrôle. Les interfaces fonctionnelles sont les points d'accès externes à un composant. Fractal offre des interfaces client et serveur. Ainsi une liaison Fractal représente une connexion entre deux composants (liaison primitive). Des liaisons de types multiples (liaisons composites) sont autorisées. Les interfaces de contrôle permettent un certain niveau de contrôle du composant auquel elles sont associées. Ces interfaces ont à charge les préoccupations extrafonctionnelles du composant. La spécification actuelle de Fractal définit quatre interfaces de contrôle : interface de contrôle d'attribut, interface de contrôle des liaisons, interface de contrôle du contenu et interface de contrôle de cycle de vie. Toutefois, Fractal dispose d'un canevas logiciel nommé Julia [50], qui offre la possibilité de définir d'autres contrôleurs.

De plus, comparativement aux conteneurs des modèles de composant CCM et EJB, le contrôleur proposé dans Fractal offre peu de services extrafonctionnels de base. En revanche, Fractal laisse une totale liberté quant à l'ajout de service extrafonctionnel, mais ne définit aucune règle ni d'outil de déploiement particulier pour le faire. [63] propose une solution à cette problématique en définissant les services extrafonctionnels eux-mêmes en composant Fractal. Cette idée à été aussi utilisée dans [132], en définissant la notion de liaison transverse qui représente les interactions entre composants et composants d'aspect. Ces derniers incarnent les préoccupations extrafonctionnelles.

Fractal met à disposition une API permettant de créer, introspecter et gérer les composants, leurs interfaces et leurs liaisons. Par exemple un composant peut être démarré et arrêté et des liaisons peuvent être créées dynamiquement. La définition de l'architecture d'une application s'effectue grâce à un descripteur XML de l'ADL (*Architecture Description Language*) Fractal.

À l'origine, Fractal était un modèle de composant qui n'était pas en adéquation avec la réalisation d'une application client/serveur. Avec Fractal RMI [50], les composants Fractal peuvent être utilisés pour construire des applications client/serveur via Java RMI.

2.3.3 Réflexivité

Le concept de la réflexivité a été introduit par Smith dans sa thèse de doctorat en 1982 [153]. Selon l'auteur, un programme peut posséder une représentation de lui-même, et peut s'en servir pour raisonner sur lui-même (introspection) et éventuellement pour modifier sa propre interpréta-

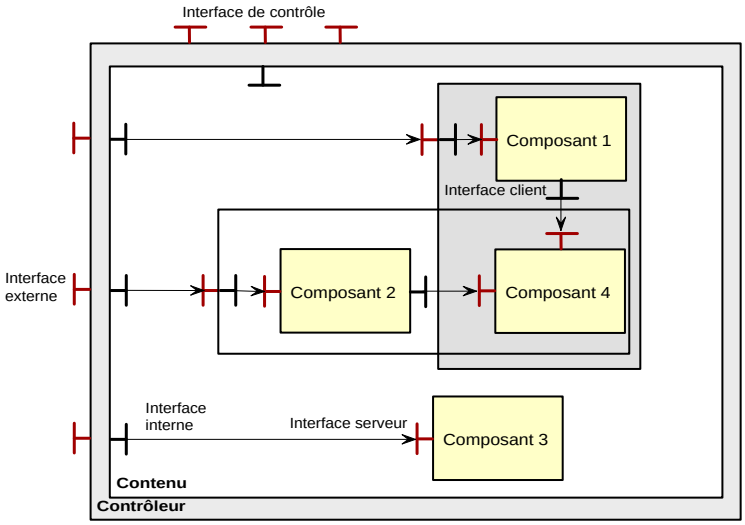


FIG. 2.10 – Composant Fractal

tion (adaptation). Dans l'architecture d'un système réflexif, la séparation des préoccupations fonctionnelles et extrafonctionnelles est assurée par l'utilisation de deux niveaux de programmation : le niveau de base qui définit le code fonctionnel d'un système et le méta-niveau qui correspond au code extrafonctionnel du système. Les opérations assurant le passage d'un niveau à un autre sont la réification et la réflexion. Ces deux opérations suivent un protocole d'interactions appelé protocole à méta-objets (en anglais, MOP pour *Meta-Object Protocol*) [84]. La figure 2.11 tirée de [91] illustre l'ensemble des concepts précédemment définis.

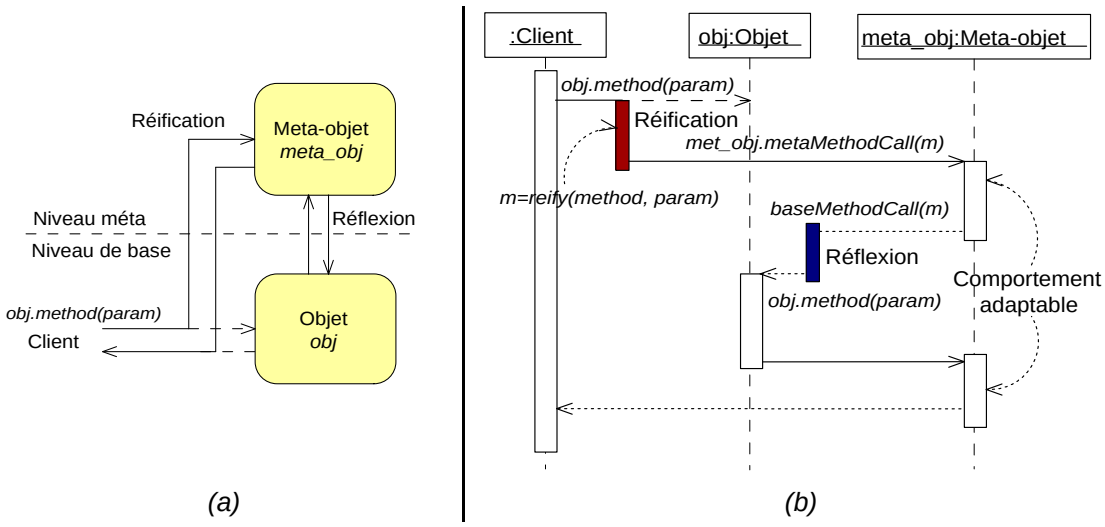


FIG. 2.11 – Principe de fonctionnement des langages à méta-objets

Contrairement aux langages de programmation classiques, les appels de méthodes dans les langages à méta-objets ne sont pas exécutés directement sur les objets cibles. En effet, si l'objet

(*obj* dans la figure 2.11-a) ne possède pas de méta-objet associé, alors la méthode est exécutée directement par l'objet *obj*. Par contre, si l'objet possède un méta-objet, l'invocation de la méthode met en collaboration l'objet et son méta-objet. La figure 2.11-b donne le diagramme UML de séquences d'un client invoquant une méthode de l'objet *obj*. Tout d'abord, l'appel est réifié, c'est-à-dire transformé en un objet manipulable par le méta-objet. Ensuite, l'appel ainsi réifié est confié au méta-objet associé (*meta_obj*) en appelant une méthode (`metaMethodCall`) que tout méta-objet doit fournir. La méthode `metaMethodCall` peut par exemple modifier les paramètres de l'appel initial. Ensuite, le méta-objet exécute le processus de réflexion en appelant une méthode prédéfinie `baseMethodCall` qui possède en argument l'appel de méthode réifié. Enfin, cette méthode effectue l'appel demandé sur l'objet de base et le résultat est envoyé au client.

La réflexivité peut être structurelle ou comportementale [46]. La réflexivité structurelle permet de modifier la structure du système. Par exemple, elle permet de modifier la structure interne d'un objet du niveau de base ou de remplacer complètement ce dernier. La réflexivité comportementale permet de changer le comportement d'un système. Un exemple typique est la redirection des appels vers d'autres objets que l'objet de base initialement ciblé.

La réflexivité est utilisée dans différents domaines. Dans le cadre des langages de programmation, 3 Lisp, une extension du langage Lisp, fut le premier à utiliser le principe de la réflexivité. 3 Lisp a été étendu au paradigme orienté objet pour définir le langage CLOS [84]. CLOS est le premier langage à introduire la notion de protocole à méta-objet. La réflexivité a été aussi utilisée dans les systèmes d'exploitation dont l'exemple le plus cité dans la littérature est Apertos [4]. Dans Apertos, chaque objet du niveau de base possède un ensemble d'objets du méta-niveau appelé *meta-space* qui définit les propriétés comportementales de l'objet de base. Enfin, nous pouvons citer les intergiciels réflexifs, qui visent principalement l'adaptation des services d'intergiciel ou de leur utilisation. Nous décrivons dans le chapitre 3 quelque intergiciels réflexifs pour l'informatique mobile.

2.3.4 Programmation orientée aspects

Une autre solution pour la séparation des préoccupations fonctionnelles et extrafonctionnelles est la programmation orientée aspects, POA (en anglais, AOP pour *Aspect Oriented Programming*) [83]. Un programme orienté aspects est constitué de deux parties : un programme de base, qui définit les préoccupations fonctionnelles de l'application, et un ou plusieurs programmes séparés, écrits dans des langages éventuellement spécialisés, qui définissent les préoccupations extrafonctionnelles à associer. La coordination entre les deux parties se fait par un procédé d'insertion appelé tissage (en anglais, *weaving*), qui peut être statique ou dynamique. Le principe est de spécifier les endroits où le tissage doit être effectué dans le code fonctionnel. Ces endroits sont appelés des points de tissage (en anglais, *join point*).

Les travaux s'étant intéressés à la POA se placent au niveau langage ou au niveau intergiciel. Les langages tels que AspectJ [82] s'occupent de plus en plus de la manière dont les aspects sont programmés. Par contre, les travaux de la POA sur les intergiciels s'occupent principalement de la définition des préoccupations extrafonctionnelles comme des services d'intergiciel et de l'intégration de ces services dans les applications. De plus, l'intégration des préoccupations

extrafonctionnelles dans les intergiciels se base non seulement sur la POA mais aussi la réflexivité, étroitement liée à la POA, et le paradigme composant/conteneur que nous avons décrit dans la section 2.3.2.

2.3.5 Mécanisme d'interception

Une autre solution de séparation des préoccupations consiste à utiliser des mécanismes d'interception entre le fournisseur et le consommateur d'un service. Les mécanismes d'interception suivent le patron de conception intercepteur décrit dans [151]. Ce patron de conception permet aux applications d'étendre un environnement d'exécution de façon transparente en intégrant de nouvelles préoccupations extrafonctionnelles grâce à des interfaces prédéfinies. En guise d'exemple, les intercepteurs portables de CORBA [122] qui sont de deux types : les intercepteurs d'IOR[7] et les intercepteurs de requêtes. Les intercepteurs d'IOR offrent la possibilité de modifier l'IOR et plus précisément les politiques de gestion d'objets CORBA. Les intercepteurs de requêtes interceptent le flux de requêtes-réponses à travers l'ORB. Deux types d'intercepteurs de requêtes existent : les intercepteurs côté clients introduisent cinq points d'interception et ceux côté serveurs cinq autres.

2.4 Synthèse

Dans ce chapitre, nous avons étudié le modèle ainsi que le contexte des applications réparties que nous considérons dans nos travaux. Nous avons entamé cette étude par la description des principes de fonctionnement des intergiciels. Les intergiciels apportent une solution aux problèmes de l'hétérogénéité et de la distribution des machines sur le réseau tout en utilisant un ensemble de services extrafonctionnels. Nous avons étudié deux catégories d'intergiciels : les intergiciels synchrones et les intergiciels asynchrones. Nous avons présenté avec plus de détails les intergiciels synchrones ; cette étude a été consolidée par une description de CORBA, COM/DCOM et Java RMI, trois intergiciels synchrones.

Cependant, l'architecture des intergiciels synchrones que nous avons étudiée dans ce chapitre repose sur un paradigme de communication orienté connexion. Nous justifions notre analyse par deux raisons. D'une part, cette architecture fait abstraction des problèmes de l'informatique mobile (cf. chapitre 1), en particulier, les déconnexions. D'autre part, les services extrafonctionnels proposés sont généralement limités en ce qui concerne leur nombre, leur type et leur interaction. Ainsi, obtenir un meilleur fonctionnement des applications en présence des déconnexions, passe donc par la définition d'un intergiciel qui offre un service de gestion des déconnexions et la définition d'un modèle de développement d'application orienté déconnexions. Ces deux solutions nécessitent sans aucun doute une séparation entre les préoccupations fonctionnelles et les préoccupations extrafonctionnelles d'un système. Ainsi, nous avons consacré la deuxième partie de ce chapitre à l'étude des différentes techniques de séparation des préoccupations.

[7] Un IOR (*Interoperable Object Reference*) est une référence CORBA.

La séparation des préoccupations est une approche visant à simplifier la conception des applications. Selon cette approche, une application contient deux parties distinctes : les préoccupations fonctionnelles et les préoccupations extrafonctionnelles. Les préoccupations fonctionnelles sont celles qui réalisent le service de base rendu par l'application. Les préoccupations extrafonctionnelles représentent la partie qui adapte les préoccupations fonctionnelles à un environnement et/ou à une utilisation particulière. Les différentes études dans la deuxième partie de ce chapitre montrent cinq techniques de séparation des préoccupations : MDA, le paradigme composant/conteneur, la réflexivité, la programmation orientée aspects et les mécanismes d'interception.

L'idée de séparer les préoccupations fonctionnelles des préoccupations extrafonctionnelles pour les applications répond bien à la gestion de déconnexions. Le paradigme composant/conteneur représente le paradigme de programmation le plus adapté à la séparation des préoccupations. Toutefois, dans la majorité des modèles de composants, la séparation des préoccupations n'est appliquée qu'à un ensemble limité et prédéfini de préoccupations extrafonctionnelles. Le modèle de conteneur extensible (ECM) présente une première solution à l'intégration d'autres préoccupations extrafonctionnelles, et nous l'utilisons dans notre réalisation pour l'intégration de la préoccupation gestion de déconnexions (cf. chapitre 5).

Chapitre 3

Gestion de déconnexions

Après avoir introduit l'informatique mobile et les différentes technologies utilisées dans les environnements mobiles (cf. chapite 1), nous avons décrit le modèle ainsi que le contexte des applications réparties que nous considérons dans nos travaux (cf. chapite 2). Nous avons ensuite étudié les différentes techniques de séparations des préoccupations fonctionnelles et extrafonctionnelles (cf. chapite 2).

Dans ce chapitre, nous présentons la problématique de la gestion des déconnexions dans les environnements mobiles. Nous débutons le chapitre avec la description des besoins en termes d'adaptation aux déconnexions et les différents modes de fonctionnement des terminaux mobiles (cf. section 3.1). Ensuite, nous détaillons le concept d'opération déconnectée dans la section 3.2. Dans la section 3.3, nous donnons une classification des solutions proposées à ce jour dans la littérature. Enfin, nous synthétisons les différents aspects abordés dans ce chapitre dans la section 3.4.

3.1 Besoins et modes de fonctionnement

Nous expliquons dans cette section les besoins d'adaptation des applications réparties au problème des déconnexions (cf. section 3.1.1). Puis, nous détaillons les différents modes de fonctionnement des applications dans les environnements mobiles (cf. section 3.1.2).

3.1.1 Besoins d'adaptation

L'adaptation peut être définie comme « rendre un système (un dispositif, des mesures. . .) apte à assurer ses fonctions dans des conditions particulières ou nouvelles » [22]. En se basant sur ce principe, nous définissons l'adaptation aux déconnexions comme « la capacité d'un système à s'ajuster aux problèmes de l'informatique mobile pour fonctionner en présence de déconnexions ». Il existe trois caractéristiques qui définissent l'adaptation. La stratégie d'adaptation

répond à la question « qui intervient dans l'adaptation ? ». Le type d'adaptation répond à la question « à quel moment l'adaptation est-elle effectuée ? ». Enfin, le mécanisme d'adaptation répond à la question « comment réaliser l'adaptation ? », opérant ainsi un lien entre la stratégie et le type d'adaptation.

D'après [144], l'intervalle des stratégies d'adaptation est délimité par deux extrémités. La première extrémité s'appelle « laissez-faire » : l'adaptation est entièrement de la responsabilité des applications. Cela évite le besoin d'un support système (système d'exploitation ou intergiciel). À l'autre extrémité, la stratégie « transparence » fait en sorte que l'adaptation soit entièrement de la responsabilité de l'intergiciel. Entre ces deux extrêmes, il existe une autre stratégie d'adaptation appelée « collaboration », dans laquelle l'adaptation se fait en collaboration entre l'application et l'intergiciel. L'intergiciel surveille les ressources, signale aux applications tout changement significatif et fournit les mécanismes d'adaptation. L'application, quant à elle, fournit les politiques d'adaptation. Les nombreux travaux synthétisés dans [72] montrent que les stratégies « laissez-faire » et « transparence » ne sont pas adéquates à la gestion des déconnexions. En effet, pour la première stratégie, il manque un contrôleur central pour résoudre les demandes de ressources concurrentes et incompatibles entre les différentes applications. Pour la deuxième stratégie, il peut y avoir des situations dans lesquelles l'adaptation réalisée est insatisfaisante ou même contre-productive pour certaines applications car elle ne prend pas en compte la sémantique des applications. Ainsi, la stratégie d'adaptation « collaboration » est la plus adéquate à l'informatique mobile. Dans notre travail, nous nous basons sur la stratégie « collaboration » en introduisant l'utilisateur comme troisième acteur d'adaptation (cf. chapitre 4). Dans ce cas, l'utilisateur peut intervenir de trois façons différentes [147] : en guidant l'application pour changer son comportement, en demandant à l'intergiciel de garantir un certain niveau de ressources, et enfin en répondant à des suggestions de l'intergiciel ou de l'application (par exemple, l'intergiciel peut demander à l'utilisateur de changer du réseau et de passer du Wi-Fi au GPRS suite à des dégradations de la bande passante du premier réseau).

Concernant le type d'adaptation, [22] propose une classification de l'adaptation en trois catégories. Premièrement, l'adaptation statique intervient pendant le développement ou avant l'exécution, soit en modifiant le code, soit en modifiant les options de compilation ou de déploiement. Elle nécessite de connaître a priori l'environnement d'exécution de l'application. Par exemple, [36] propose une solution pour la gestion des déconnexions pour applications à base d'objets CORBA en utilisant dans l'ORB côté client un intercepteur portable, qui orchestre un service de gestion des déconnexions. L'ajout de l'intercepteur portable se fait en modifiant le début du code de l'application. Deuxièmement, l'adaptation dynamique intervient au moment de l'exécution de l'application. Elle est réalisée par une intervention extérieure, la plupart du temps humaine. Enfin, l'auto-adaptation intervient également pendant l'exécution, mais peut être initiée par l'application ou par l'intergiciel. Ce type d'adaptation est utilisé par exemple dans le canevas logiciel MobileJMS [79], qui propose d'utiliser un mécanisme de compression de données si la connectivité entre le client et le serveur est faible. Les choix sont faits par l'intergiciel à partir de descripteurs fournis par l'application.

Le tableau 3.1 présente les relations entre les stratégies et les types d'adaptation. Il est clair que l'adaptation statique peut être choisie dans toutes les stratégies d'adaptation. Au contraire, l'adaptation dynamique ne peut être utilisée que dans le cas de la stratégie « collaboration »

avec comme acteur l'utilisateur. Enfin, l'auto-adaptation peut être effectuée dans toutes les stratégies d'adaptation. Dans notre travail, nous proposons d'effectuer l'adaptation en mixant les trois types d'adaptation. Les adaptations statique et dynamique sont utilisées dans la méthodologie de conception que nous proposons (cf. chapitre 4). L'auto-adaptation consiste en la commutation transparente entre les différentes configurations de l'application définies dans l'adaptation statique (cf. chapitre 5).

	Stratégie laissez-faire	Stratégie transparence	Stratégie collaboration	Stratégie collaboration + vision utilisateur
Adaptation statique	X	X	X	X
Adaptation dynamique				X
Auto-adaptation	X	X	X	X

TAB. 3.1 – Stratégies et types d'adaptation

3.1.2 Modes de fonctionnement

La figure 3.1 schématise les différents modes de fonctionnement des terminaux mobiles ainsi que les différentes transitions possibles entre ces modes. Comme dans [136], nous considérons quatre modes de fonctionnement : connecté, partiellement connecté, déconnecté et veille. Dans le mode connecté, le terminal mobile dispose d'une bonne connexion au réseau, à la manière d'une station fixe. Dans le mode partiellement connecté, le terminal mobile ne dispose plus pour communiquer avec le réseau que d'un lien faible dû par exemple à des perturbations du réseau hertzien ou à un faible niveau de la batterie. Dans le mode déconnecté, le terminal mobile est isolé soit parce qu'il n'est plus physiquement relié au réseau soit parce qu'il est impossible de maintenir une connexion sans fil. Enfin, dans le mode veille utilisé par les terminaux mobiles pour préserver les ressources énergétiques, la connexion réseau est maintenue, mais le terminal mobile ne s'en sert pas. L'objectif de notre travail est d'abord de gérer les déconnexions. Aussi, nous ignorons le mode veille et ne considérons que les modes connecté, partiellement connecté et déconnecté.

Dans la figure 3.1, les transitions (numérotées de *1* à *6*) entres les modes de fonctionnement correspondent aux changements de valeur de la ressource considérée. Les pseudo transitions *5* et *6* correspondent à des déconnexions/reconnexions brusques qui peuvent être volontaires ou involontaires (cf. section 1.1.2). Cependant, le passage entre les modes de fonctionnement connecté et déconnecté ne se fait pas directement. En effet, le mode de fonctionnement passe d'abord par le mode partiellement connecté qui est, dans ce cas, un état transitoire de courte durée. C'est la raison pour laquelle ce sont des pseudo-transitions : nous passons toujours par le mode de fonctionnement partiellement connecté.

Dans les environnements mobiles, la détection de la connectivité peut être assuré par un détecteur de connectivité. Ce dernier donne les informations sur une ressource telle que la bande passante ou la batterie. Les détecteurs de connectivité peuvent être déployés selon plusieurs granularités possibles : un détecteur de connectivité par hôte, par application, par processus, par lien logique.... Le concept de détecteur de connectivité est fortement inspiré du concept de détecteur de défaillance. Ce dernier a suscité une littérature abondante qui traite, d'une part,

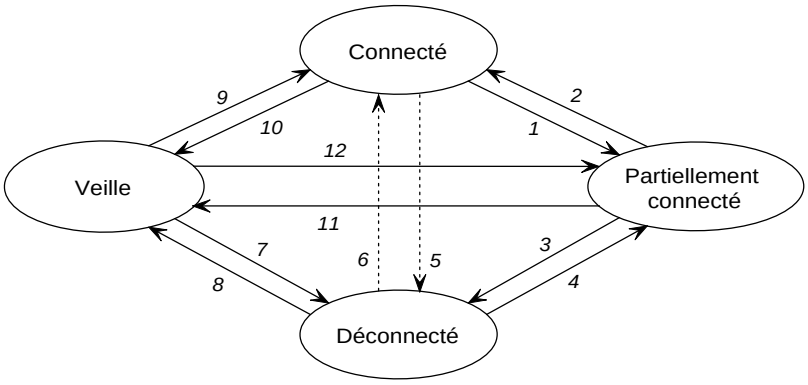


FIG. 3.1 – Modes de fonctionnement des applications

des aspects théoriques (algorithmes et preuves), et d'autre part, des aspects pratiques (qualité de service des détecteurs). Dans notre travail, nous supposons que le détecteur de connectivité existe et nous utilisons celui présenté dans [158]. Pour prédire les déconnexions, le détecteur de connectivité utilise un mécanisme d'hystérésis pour lisser les variations du niveau de disponibilité des ressources (cf. figure 3.2). Pour chaque ressource, l'hystérésis définit quatre seuils (*highUp*, *highDown*, *lowUp* et *lowDown*) qui représentent le niveau de disponibilité de la ressource. Ainsi, quand le niveau de la bande passante augmente et est inférieur au seuil *lowUp* (respectivement *highUp*), le terminal mobile est déconnecté (respectivement partiellement connecté). Quand le niveau de la bande passante diminue mais est plus élevé que la valeur *highDown* (respectivement *lowDown*), le terminal mobile est connecté (respectivement partiellement connecté). Sans le mécanisme présenté dans la figure 3.2-a-2, il peut y avoir un effet *ping-pong*[1] autour des valeurs *highDown* et *lowUp*. L'effet *ping-pong* intervient si le niveau de disponibilité de la ressource oscillant autour d'une valeur frontière (seuil) entre deux modes de connectivité, provoque des changements de mode répétés. Donc, quand le niveau de la bande passante arrive dans l'état F en provenance de l'état E (respectivement de l'état C en provenance de l'état B) et que le niveau de la bande passante dépasse la valeur *highDown* (respectivement *lowUp*), le mode demeure partiellement connecté jusqu'à la valeur *highUp* (respectivement *lowDown*).

Le diagramme de transition d'états correspondant est dessiné dans la figure 3.2-b. L'état qui suit immédiatement l'état initial et qui précède immédiatement l'état final est l'état D. Ainsi, il est supposé que le client démarre et arrête l'application sur le terminal mobile alors qu'il est fortement connecté. Nous justifions ce choix dans le chapitre 5.

La prédiction du niveau de ressource du terminal mobile joue un rôle phare dans la détection de la connectivité. Dans les environnements mobiles, la prédiction est utilisée pour faire des exécutions distantes (en anglais, *off-loading*) afin de sauvegarder les ressources locales (CPU, batterie, mémoire, bande passante. . .) [57]. Les systèmes d'exécution distante doivent prévoir les coûts d'exécutions locale et distante pour déterminer si l'exécution distante est la plus rentable. Cependant, le calcul du coût d'exécution est effectué localement, il ne doit pas consommer beau-

[1] Dans [58], un effet similaire au *ping-pong* appelé *boomerang* a été détecté dans le contexte de la migration de processus.

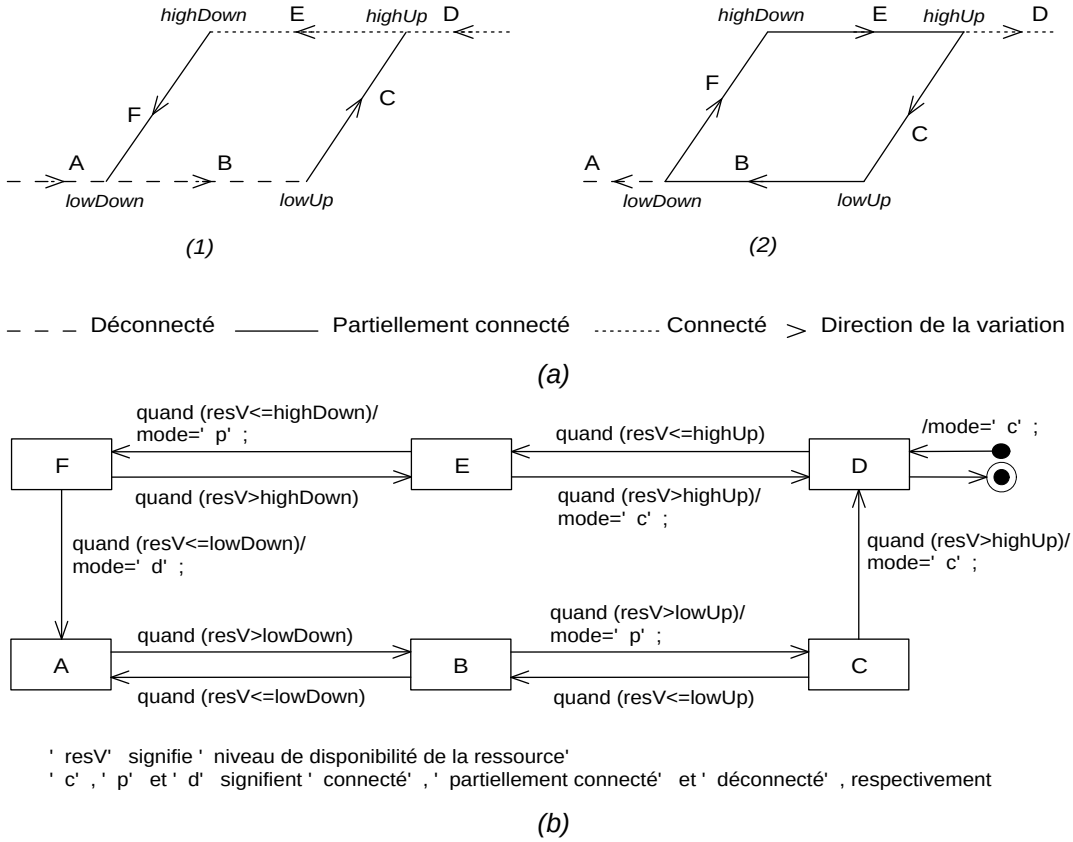


FIG. 3.2 – Hystérésis de la détection de connectivité

coup de ressources en plus d'être précis. Comme exemple de systèmes qui utilisent la prédiction, citons RPF [140], Spectra [47] et NWSLite [57]. Dans notre travail, nous utilisons la prédiction pour vérifier si une invocation distante peut être effectuée en mode partiellement connecté (cf. chapitre 5).

3.2 Opération déconnectée

Dans les environnements mobiles, la continuité de service entre le client et le serveur en présence des déconnexions est assurée par le maintien d'une connexion logique en utilisant le concept d'opération déconnectée [86, 148, 64]. La figure 3.3 illustre le principe des opérations déconnectées. Quatre principaux acteurs interviennent : le client (l'application côté client) sur l'hôte mobile, l'entité[2] distante de l'application répartie, le cache[3] qui représente une partie de la mémoire du terminal mobile, et enfin, l'entité déconnectée qui est un mandataire en terme de code et d'état de l'entité distante dans le cache. Le fonctionnement des opérations déconnectées est le suivant. En présence des déconnexions, le client sur le terminal mobile utilise l'entité déconnectée

[2]Ici, le terme entité peut désigner un fi chier, une base de données, une page web, un objet ou un composant.
[3]Dans la suite de cette thèse, pour des raisons de simplifi cation, le mot cache désigne un cache logiciel.

préalablement déployée dans le cache. Les opérations effectuées sur l'entité déconnectée pendant les phases des déconnexions sont journalisées et des réconciliations lors des reconnexions sont opérées entre l'entité distante et l'entité déconnectée.

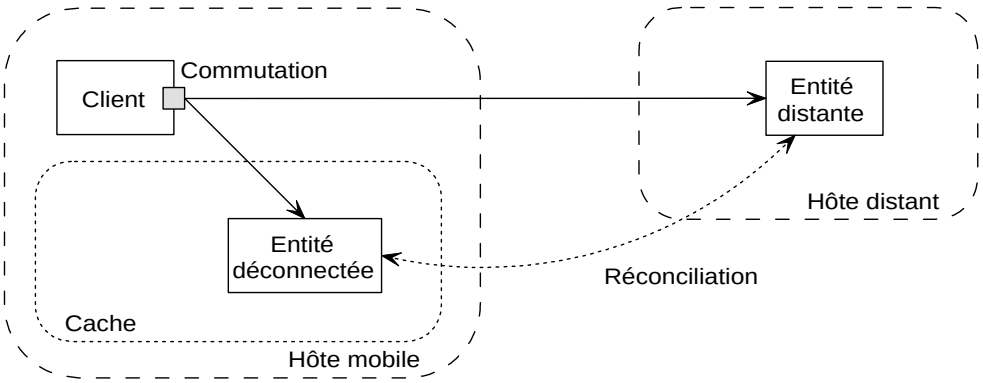


FIG. 3.3 – Fonctionnement des opérations déconnectées

L'utilisation des opérations déconnectées pour le fonctionnement en présence des déconnexions fait apparaître quatre thématiques : la gestion du cache du terminal mobile, la gestion de la cohérence des différentes entités, la détection des déconnexions et la commutation entre l'entité distante et l'entité déconnectée. Concernant la gestion de cache, il doit offrir deux stratégies [52]. La stratégie de déploiement détermine les entités déconnectées à créer dans le cache du terminal mobile, quand et pour quelle durée. La stratégie de remplacement décide quelles entités déconnectées doivent être supprimées lorsqu'il n'existe plus assez d'espace mémoire dans le cache. Par ailleurs, l'utilisation d'une entité déconnectée peut exiger la présence d'autres entités déconnectées dans le cache. Ainsi, les stratégies de déploiement et de remplacement doivent prendre en compte les dépendances entre les entités de l'application. La gestion de cohérence maintient la cohérence entre les entités déconnectées dans le cache du terminal mobile et les entités distantes dans les autres hôtes. Cette cohérence est maintenue en procédant à des journalisations des opérations et des réconciliations entre l'entité distante et l'entité déconnectée. La détection des déconnexions est assurée par le détecteur de connectivité (cf. section 3.1.2). Enfin, la commutation entre l'entité distante et l'entité déconnectée décrit le mécanisme de redirection des requêtes du client vers l'entité déconnectée en présence des déconnexions. Cette dernière thématique est fortement liée à la technique de séparation des préoccupations (cf. section 2.3) et à la stratégie d'adaptation (cf. section 3.1.1). Par exemple, la redirection peut être de la responsabilité de l'application (dans le code source de la partie cliente) ou de l'intergiciel via des mécanismes dédiés, tels que les intercepteurs portables de CORBA utilisés dans [36].

Parmi les thèmes présentés ci-dessus, le travail que nous présentons dans cette thèse ne traite que les thématiques de la gestion du cache et de la commutation entre l'entité distante et l'entité déconnectée. Nous proposons une méthodologie de conception d'applications orientées déconnexions qui correspond au premier travail pour la gestion du cache. La commutation entre l'entité distante et l'entité déconnectée se fait grâce à un modèle de conteneur de composants que nous proposons. Nous proposons aussi un service de gestion de cache pour composants

CORBA. Concernant la gestion de la cohérence, [88] présente une intégration de nos travaux avec un service de gestion de réconciliation dans le cadre du projet RNTL AMPROS [1].

3.3 Gestion de cache pour la déconnexion

Avant d'aborder les principales solutions de gestion du cache pour les déconnexions, nous décrivons dans la section 3.3.1 les différents critères de classification que nous avons retenus. Pour des raisons de clarté, nous présentons ces solutions suivant le critère « granularité », qui représente l'entité de programmation manipulée par l'application (fichier, objet. . .).

3.3.1 Critères d'étude

La plupart des classifications de solutions de gestion des déconnexions existantes se focalisent sur des domaines d'application particuliers comme les systèmes de fichier [52], les objets répartis [101], ou encore les bases de données [51]. De plus, les classifications existantes sont rares et se concentrent sur un sous-ensemble de critères [8, 72]. Dans cette thèse, nous proposons une étude des différentes solutions de gestion des déconnexions en se basant sur dix critères :

- granularité : elle définit le type de l'entité manipulée par l'application. Dans notre étude, nous considérons les granularités fichier, base de données, page web, objet et composant ;
- type de déconnexion : les déconnexions peuvent être volontaires ou involontaires ;
- type de cache : le cache peut être local (dans le terminal mobile) ou réparti ;
- modes de fonctionnement pris en compte : il fait référence aux trois modes de fonctionnement présentés dans la section 3.1.2 : connecté, partiellement connecté et déconnecté ;
- stratégie d'adaptation : ce critère définit la stratégie d'adaptation utilisée pour la gestion des déconnexions. Comme présentée dans la section 3.1.1, la stratégie d'adaptation peut être laissez-faire, transparence ou collaboration ;
- type d'adaptation : ce critère définit le type d'adaptation utilisée pour la gestion des déconnexions (cf. 3.1.1) : statique, dynamique et auto-adaptation ;
- stratégies de déploiement et de remplacement du cache : ce critère décrit les stratégies de déploiement et de remplacement (s'ils existent) proposées pour la gestion de cache ;
- présence d'un mécanisme de commutation entre l'entité distante et l'entité déconnectée ;
- utilisation d'un modèle de conception orienté déconnexion : ce critère vérifie si la solution de gestion des déconnexions propose un modèle de conception d'application devant fonctionner en présence des déconnexions ;
- prise en compte des dépendances entre entités de l'application : ce critère définit si les dépendances entre les entités de l'application sont prises en compte dans les stratégies de déploiement et de remplacement du cache.

Dans la suite de cette section, nous classons les solutions qui traitent le problème des déconnexions suivant la granularité manipulée par les applications. Nous donnons pour chaque solution notre position vis-à-vis des autres critères cités ci-dessous. Le tableau 3.2 donne un récapitulatif des différentes solutions étudiées.

3.3.2 Système orienté fichier

La gestion des déconnexions a été abordée pour la première fois dans le cadre des systèmes de gestion de fichiers. Nous décrivons avec plus de détails dans cette section les trois principales propositions de la littérature : Coda, Amigos et Seer. Ensuite, nous résumons les résultats de quelques travaux de recherche relatifs à la gestion des déconnexions.

Coda

Coda [85, 146] est un système de gestion de fichiers dont le but est d'offrir au client la continuité d'accès aux données en présence des déconnexions volontaires et involontaires en utilisant les opérations déconnectées. Coda hérite de plusieurs caractéristiques de conception et d'utilisation de AFS (Andrew File System) [62]. Les clients visualisent Coda comme un simple système de gestion de fichiers partagés UNIX. L'espace de nommage de Coda est partagé sur plusieurs serveurs d'archivage qui construisent des sous-arbres appelés volumes. De plus, Coda utilise la réplication des serveurs pour permettre la disponibilité et la tolérance aux fautes. Les serveurs de répliques peuvent être vus comme un cache réparti.

En plus des serveurs de répliques, Coda utilise un cache local sur chaque terminal. Dans le cas où aucun serveur de répliques n'est disponible pour un volume donné, Coda utilise les volumes déployés localement. Le contenu du cache local est géré par un gestionnaire de cache appelé Venus. Il peut être dans l'état accumulation, émulation ou réintégration. La figure 3.4 représente le diagramme de transition de Venus. Venus est normalement dans l'état accumulation : il se base sur les serveurs de répliques pour les opérations sur les fichiers. S'il y a une déconnexion, Venus passe dans l'état d'émulation et l'utilisateur continue son travail sur les volumes disponibles dans le cache. La commutation entre les fichiers dans le cache et les fichiers dans les serveurs distants est réalisée par le système de gestion de fichiers de Coda. Cette commutation est transparente à l'utilisateur. À la reconnexion, Venus passe à l'état réintégration pour synchroniser les modifications effectuées durant la déconnexion. Dans Coda, le mode partiellement connecté est exploité principalement pour opérer une propagation des modifications « goutte à goutte » en tâche de fond pour ne pas dégrader les performances [115].

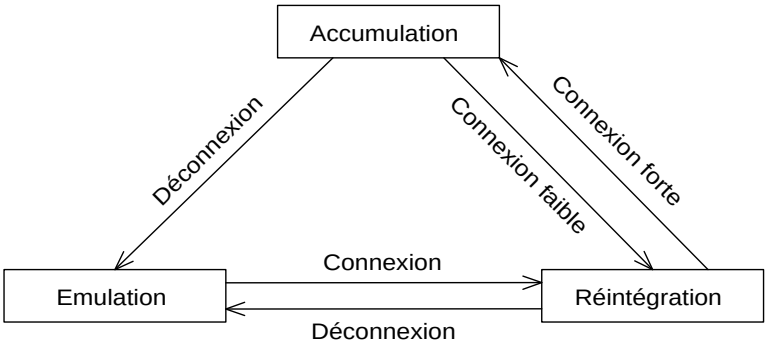


FIG. 3.4 – Diagramme de transitions de Venus

La stratégie de déploiement des fichiers de Coda est basée sur la notion de données implicites et de données explicites. Les données implicites représentent l'historique d'utilisation du client. Elles prennent la forme d'une base de données appelée *HDB* (*Hoard Data Base*) construite par le client de l'application. Un programme appelé *Hoard* permet à l'utilisateur de mettre à jour la HDB directement ou via un script de commandes appelé *Hoard Walking*. En mode accumulation, Coda anticipe les déconnexions en copiant localement (pré-chargement) les fichiers du HDB selon une priorité accordée par l'utilisateur. Les autres fichiers sont chargés sur un défaut d'accès. Coda utilise la politique LRU (*Least Recently Used*) pour le remplacement. Cependant, si l'utilisateur fait un mauvais choix dans le HDB suite à une mauvaise compréhension de l'application, cette dernière peut ne pas fonctionner en mode déconnecté. De plus, l'absence de donnée dans le cache ne peut pas être masquée, elle apparaît à l'utilisateur comme erreur d'accès à un fichier manquant.

La notion de dépendance entre les fichiers ou les volumes n'a pas été abordée dans Coda. Un fichier peut être utilisé localement sans avoir besoin d'autres fichiers.

Amigos

Amigos [2] est un projet de recherche dans le domaine de l'informatique mobile. Il aborde plusieurs aspects tels que la communication réseau, le système de gestion de fichiers et les transactions. L'aspect qui nous intéresse est le système de gestion de fichiers. Ce dernier représente une extension de NFS afin de permettre les opérations déconnectées[4]. Dans Amigos, le terminal mobile oscille entre les modes de fonctionnement connecté et déconnecté. Le mode partiellement connecté n'existe pas. Les déconnexions peuvent être volontaires et involontaires.

En mode connecté, l'accès aux fichiers reste inchangé (de la même manière que NFS). En tâche de fond, Amigos déploie des fichiers sur le terminal mobile en se basant sur un profil d'application défini par l'utilisateur. Le profil contient des références de fichiers et de répertoires. L'utilisateur assigne pour chaque référence une priorité comme dans Coda. L'utilisateur peut à tout moment changer son profil. En mode déconnecté, toutes les requêtes de l'utilisateur sont redirigées, d'une manière transparente à l'utilisateur, vers le cache. Le mécanisme de redirection est intégré dans le système de gestion de fichiers. Dans ce mode, le fichier/répertoire distant est verrouillé pour éviter les problèmes de mise à jour.

Pour le remplacement, Amigos met à jour périodiquement le contenu du cache en consultant le profil de l'utilisateur. Il supprime les fichiers/répertoires enlevés du profil et déploie les nouveaux fichiers/répertoires suivant leur priorité. De ce fait, un fichier/répertoire non référencé dans le profil ne peut pas être déployé dans le cache. Par ailleurs, Amigos ne définit aucune dépendance entre les fichiers/répertoires dans les stratégies de déploiement et de remplacement.

Seer

Le système Seer [96, 95] fait partie d'un projet de recherche appelé Travler [162]. Il propose une approche prédictive de déploiement. Elle est basée sur l'idée qu'un système peut observer le

[4] Les auteurs de [2] utilisent la formulation « opérations semi-connectées » pour designer les opérations déconnectées.

comportement de l'utilisateur, faire des inférences sur les relations sémantiques entre les fichiers référencés et utiliser ces inférences pour sélectionner les fichiers à déployer localement. Seer se base sur le concept de « distance sémantique » pour définir la dépendance entre les fichiers [94]. Les fichiers ayant entre eux une faible distance sémantique sont regroupés en *projets*. Seer définit plusieurs distances sémantiques entre les fichiers :

- la distance sémantique temporelle est la durée qui sépare l'utilisation de deux fichiers ;
- la distance sémantique à base de séquences est le nombre de références vers d'autres fichiers entre l'utilisation de deux fichiers. Par exemple, dans la séquence d'accès suivante A, S, S, X, B, la distance sémantique entre le fichier A et le fichier B est égale à trois ;
- la distance sémantique de vie entre l'ouverture d'un fichier A et l'ouverture d'un fichier B est définie à 0 si A n'a pas été fermé pendant que B est ouvert. Sinon, c'est le nombre d'ouvertures d'autres fichiers y compris l'ouverture du fichier B.

L'architecte de Seer définit deux entités : un observateur et un corrélateur. Le premier observe le comportement de l'utilisateur et ses accès aux fichiers, classant chaque accès selon le type de fichier (texte, vidéo, image...). Ensuite, il fournit ces résultats au corrélateur. Ce dernier calcule les distances sémantiques entre les fichiers et construit les projets. Il est à noter que l'entité de déploiement dans Seer est tout le projet. Quand un nouveau contenu de cache doit être choisi, le corrélateur examine l'ensemble des projets pour trouver ceux qui sont actuellement en activité et choisit les projets ayant la plus grande priorité jusqu'à ce que la taille maximum du cache soit atteinte. La procédure de calcul de la priorité entre projets n'est pas fixée dans Seer. Le gestionnaire de cache peut par exemple utiliser un algorithme tel que LRU pour l'attribution de la priorité. La stratégie de déploiement proposée par Seer se base sur les références de fichiers déjà visités par l'utilisateur. Cependant, un fichier non référencé avant la déconnexion ne peut pas exister dans le cache en mode déconnecté. De plus, Seer ne propose aucune stratégie de remplacement.

Autres projets

PFS est un système de gestion de fichiers pour les environnements mobiles [40]. Son objectif est de fournir une adaptation *laissez-faire* au problème des déconnexions. Il définit une architecture à trois tiers : client sur le terminal mobile, serveur fixe et client fixe. Ce dernier est une réplique du serveur fixe. Contrairement à Coda, le client dans PFS n'interagit qu'avec le serveur réplique. PFS supporte les trois modes de fonctionnement (connecté, partiellement connecté et déconnecté). En mode connecté, le client PFS utilise directement le serveur réplique. En modes déconnecté et partiellement connecté, le client PFS utilise les fichiers dans le cache local. De plus, en modes connecté et partiellement connecté, PFS propage les modifications effectuées sur les fichiers dans le terminal mobile en n'envoyant que les blocs modifiés d'un fichier. La propagation des modifications en mode partiellement connecté est aussi utilisée dans [65] qui propose une extension du gestionnaire de cache de PFS.

Une étude de l'intérêt du pré-chargement a été donnée dans [45]. Cette étude montre que cette stratégie, combinée avec une compression des données, peut réduire la latence observée par l'utilisateur de 23%. Dans le même contexte, [129] montre qu'un mécanisme de pré-

chargement peut réduire la latence d'un facteur de 45%, au prix d'un doublement de la bande passante utilisée. [102] utilise le pré-chargement dans l'objectif d'optimiser l'utilisation de la bande passante. Les documents susceptibles d'être requis sont donc si possible téléchargés durant les périodes creuses d'utilisation de la bande passante.

3.3.3 Système orienté base de données

Dans une grande majorité des cas, les solutions de gestion des déconnexions dans les systèmes orientés bases de données sont inclus dans les solutions de duplication des bases de données, les transactions mobiles et les bases de données embarquées [14]. La problématique que nous abordons dans cette thèse nous a amené à nous focaliser sur la duplication des bases de données dans les terminaux mobiles. Ainsi, nous présentons Bayou, une solution de référence au problème de duplication dans les environnements mobiles. Nous donnons ensuite une brève présentation d'autres travaux de recherche.

Bayou

Bayou est un projet de Xerox PARC [159, 134]. Son but est de construire un support système pour partager des données entre utilisateurs mobiles via une communication point-à-point. Il propose un système de communication client/serveur flexible. Dans ce système, un client peut accéder à des données de n'importe quel serveur avec lequel il peut communiquer. Réciproquement, n'importe quel terminal, y compris un terminal mobile, qui possède une copie de la base de données[5] doit être disponible pour des requêtes d'écriture ou de lecture pour d'autres terminaux. Donc, les terminaux mobiles peuvent être des serveurs pour certains terminaux et des clients pour d'autres.

Comme dessiné dans la figure 3.5, chaque base de données est répliquée dans plusieurs terminaux. Cette réplication est dynamique : le nombre et la localisation des différentes copies évoluent dans le temps. Le déploiement de la base de données sur le terminal mobile se fait à la demande de l'utilisateur. L'application cliente communique avec le serveur à travers une interface spécifique à Bayou. Cette interface est implantée dans la souche client et elle fournit deux opérations : lecture et écriture. De plus, cette interface prend en charge la commutation transparente à l'utilisateur entre la copie locale et les copies distantes de la base de données.

Bayou réalise un schéma de réplication *read-any/write-any*, c'est-à-dire qu'un client peut écrire sur une base de données d'un serveur, et après un moment, lire les données d'un autre serveur. Le client peut voir des résultats contradictoires à moins que les deux serveurs aient mis à jour leurs bases de données de façon cohérente. Bayou utilise un protocole dit anti-entropique pour la propagation des mises à jour [133]. Il assure que toutes les copies d'une base de données sont convergentes vers le même état. En outre, le principe d'un tel protocole autorise deux terminaux quelconques qui peuvent communiquer à propager périodiquement leurs mises à jour.

[5] Les concepteurs de Bayou ciblent des bases de données légères (en anglais, *lightweight*) qui peuvent être déployées sur des terminaux mobiles. Ces bases de données (par exemple, une agenda) sont gérées par des SGBD spécifiques tels que *Sybase Adaptive Server Anywhere*, *Oracle 8i Lite*, *SQLServer* pour Windows CE, *DB2 Everyplace*...

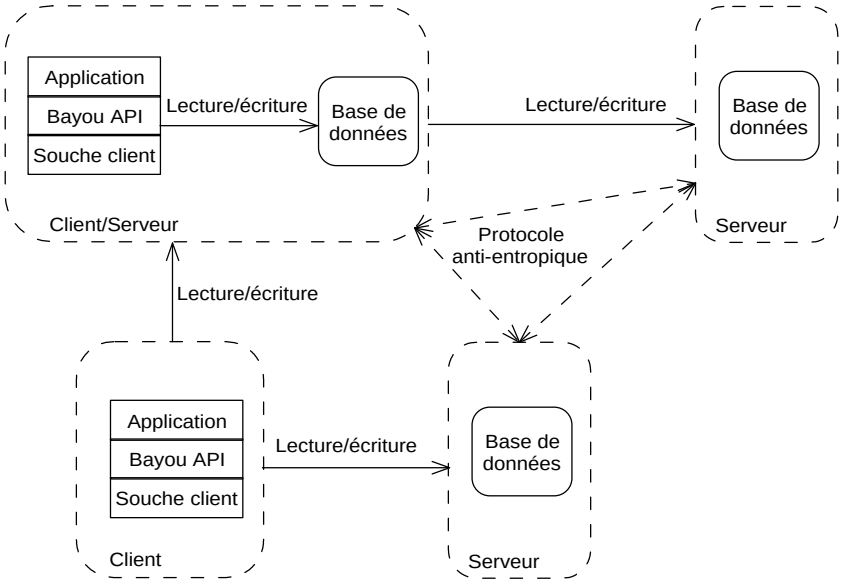


FIG. 3.5 – Architecture de Bayou

Le fonctionnement de Bayou est le suivant : en mode connecté (au moins un lien point-à-point est en mode connecté), le client fonctionne en *read-any/write-any*, contrairement à Coda qui n'utilise qu'un seul serveur de réplique. Les serveurs avec lesquels l'application cliente peut communiquer sont choisis suivant un patron d'usage qui dépend du réseau. Par exemple, le client n'utilise que les serveurs qui se trouvent dans la même cellule. Périodiquement, le protocole anti-entropique met à jour la copie locale. En mode partiellement connecté (aucun lien point-à-point n'est en mode connecté et au moins un lien point-à-point est en mode partiellement connecté), le client n'utilise que la base de données locale. Bayou propage les modifications vers les autres serveurs en tâche de fond. En mode déconnecté (tous les liens point-à-point sont en mode déconnecté), le client n'utilise que la base de données locale et les modifications sont propagées dès qu'un lien passe en mode partiellement connecté. Cependant, si le terminal mobile ne dispose pas d'une copie locale de la base de données, le client ne peut pas fonctionner en modes partiellement connecté et déconnecté.

Autres projets

Oasis est un système de gestion de données pour les environnements mobiles [139]. Il se base sur une communication point-à-point. Dans Oasis, la base de données est répliquée dans plusieurs hôtes. Comme dans Bayou, les hôtes qui disposent d'une copie de la base de données peuvent être des clients et des serveurs en même temps. Cependant, la base de données n'est déployée sur un hôte que si ce dernier dispose des ressources suffisantes (mémoire, CPU. . .). En mode déconnecté, le client sur le terminal mobile utilise la base de données déployée localement. En mode connecté, le client Oasis utilise un mécanisme de vote à base de quorum (en anglais, *weighted voting*) pour déterminer le serveur de répliques le plus à jour. Ce mécanisme assigne

pour chaque réplique un numéro de version pour permettre au client de déterminer la réplique la plus récente. Ce mécanisme de vote est aussi utilisé dans Deno [28].

3.3.4 Système orienté page web

Le domaine des caches web est très actif tant dans la recherche académique que dans le monde industriel [10]. Nous décrivons dans cette section deux prototypes de cache web. Nous débutons par la présentation de WebExpress, un prototype industriel. Puis, nous présentons SliCache, un prototype académique. Enfin, nous décrivons deux algorithmes de remplacement que nous utilisons dans le chapitre 6.

WebExpress

IBM WebExpress est un système d'optimisation d'accès web dans un environnement mobile [61]. Son objectif est de pouvoir fonctionner sur n'importe quel navigateur web (par exemple, Netscape, Mosaic. . .) et sur n'importe quel serveur web sans apporter de changements ni sur l'un ni sur l'autre. Pour atteindre cet objectif, WebExpress utilise le modèle client/intercepteur/serveur. Ce modèle permet d'intercepter et de contrôler toutes les requêtes échangées entre le navigateur web et le serveur web (cf. figure 3.6). L'architecture de WebExpress se décline en deux composants : l'intercepteur côté client CSI (*Client Side Intercept*) qui comporte un cache de pages web, et l'intercepteur côté serveur SSI (*Server Side Intercept*) qui comporte aussi un cache de pages web. Le navigateur web utilise le CSI comme un serveur web mandataire. La communication entre eux se fait via une connexion TCP locale (interface *loopback*).

En modes connecté et partiellement connecté, le déroulement de l'exécution d'une demande de page web d'un utilisateur est le suivant : la requête est acheminée au CSI qui l'exécute sur la copie locale dans le cache si elle existe. Dans le cas où la copie locale n'existe pas, le CSI renvoie la requête vers le SSI. Si une copie de la page web existe dans le SSI, ce dernier exécute la requête sur cette copie et retourne la page web au CSI qui sauvegarde une copie locale avant de l'acheminer au navigateur. Dans le cas où le SSI ne dispose pas d'une copie de la page web, il laisse passer la requête pour atteindre le serveur web, le serveur web considérant alors le SSI comme client. En mode déconnecté, toutes les requêtes du navigateur sont exécutées sur le cache du CSI. Cependant, si le cache ne dispose pas d'une copie d'une page web demandée, le CSI renvoie une erreur au navigateur (page web non trouvée).

WebExpress associe à chaque page web dans le cache (dans le CSI ou le SSI) un intervalle de cohérence qui indique la validité de la page web dans le cache. Ce dernier peut être modifié par les utilisateurs au niveau de chaque CSI et par l'administrateur au niveau du SSI. Quand une page web dans le CSI (respectivement SSI) est référencée et si l'intervalle de cohérence est dépassé, le CSI (respectivement SSI) renvoie la requête vers le SSI (respectivement serveur web) et met à jour son cache au retour du résultat.

Pour le remplacement, WebExpress utilise l'algorithme LRU dans le CSI et le SSI. Dans le CSI, l'utilisateur peut choisir un ensemble de pages web appelées « pages web infiniment persistantes », qui sont exclues du remplacement. Il peut aussi modifier cet ensemble en cours

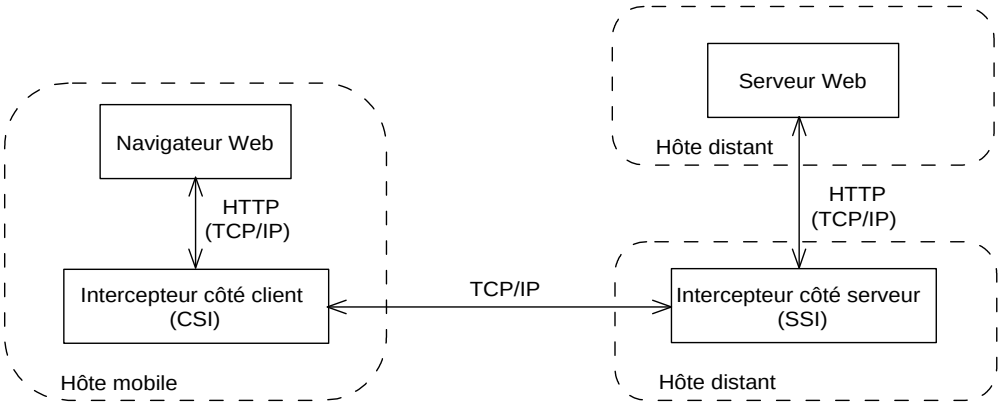


FIG. 3.6 – Architecture de WebExpress

d'exécution. Cependant, l'utilisateur peut choisir un ensemble de pages web très volumineux qui limite la rentabilité du cache. Dans le SSI, la fréquence d'accès utilisée dans l'algorithme LRU est calculée sur tous les accès de tous les utilisateurs. Par ailleurs, le déploiement d'une page web dans le cache correspond au déploiement du code HTML de la page web et de toutes les images associées à la page web. Cependant, WebExpress ne prend pas en compte les liens hypertextuels qui pointent vers d'autres pages web. Donc, WebExpress ne prend pas en compte les dépendances entres les pages web dans les stratégies de déploiement et de remplacement du cache. Contrairement à WebExpress, [92] propose une stratégie de déploiement de pages web qui prend en compte les dépendances entres les pages web. Cette approche pré-charge systématiquement la totalité des liens associés à la page web courante.

SliCache

L'objectif principal de SliCache est de proposer une stratégie de remplacement de pages web dans un cache local [80]. Cette stratégie se base sur trois paramètres : le nombre d'accès, la taille des pages web et le temps moyen nécessaire pour déployer une page web. Le calcul de ces paramètres se fait d'une manière transparente aux serveurs web. En outre, SliCache partitionne le cache en deux partitions : *R-slice* et *I-slice*. Le *R-slice* contient les pages web référencées une seule fois. Le *I-slice* contient les pages web référencées au moins deux fois ou récemment référencées. De plus, SliCache définit la notion d'intervalle d'accès d'une page web dans le cache. Ce dernier est la différence entre le temps courant et le temps de la dernière référence de la page web.

Dans SliCache, chaque partition du cache utilise une fonction de coût. Pour le *R-slice*, le coût correspond au rapport entre le nombre d'accès et la taille de la page web. Pour le *I-slice*, le coût correspond au produit de l'intervalle d'accès et le nombre d'accès de la page web. SliCache définit *R-victim* (respectivement *I-victim*) la page web à moindre coût et la prochaine page web à remplacer dans le *R-slice* (respectivement *I-slice*). Pour sélectionner la page web à remplacer (*R-victim* ou *I-victim*), SliCache utilise un mécanisme de prédiction. Dans ce dernier, SliCache

remplace la *I-victim* s'il y a une éventuelle référence de la *R-victim* avec une dégradation de la popularité de la *I-victim*. Dans le cas contraire, SliCache remplace la *R-victim*.

Autres projets

Il existe plusieurs algorithmes de remplacement dans la littérature. Nous décrivons dans cette section deux algorithmes que nous utilisons dans le chapitre 6 : SIZE [173] et GDSF [31]. SIZE consiste à remplacer le document le plus volumineux par plusieurs documents moins volumineux. L'inconvénient de cet algorithme est que les documents de petites tailles peuvent exister en permanence dans le cache sans être utilisés. GDSF assigne pour chaque document D une clé $\mathcal{K}(D)$. Lorsque un remplacement doit être effectué, GDSF remplace le document avec la plus faible clé. La clé est calculée suivant la fonction $\mathcal{K}(D) = \mathcal{L} + \mathcal{F}(D) * \mathcal{C}(D)/\mathcal{S}(D)$ où $\mathcal{L}$ est la clé du dernier document remplacé (initialement égale à 0), $\mathcal{F}(D)$ est le compteur d'accès du document D, $\mathcal{C}(D)$ est le coût de déploiement du document D (en relation avec la bande passante, la mémoire...) et $\mathcal{S}(D)$ est la taille du document D.

3.3.5 Système orienté objet

Nous décrivons dans cette section deux projets relatifs à la gestion des déconnexions pour applications orientées objets. Dans un premier temps, nous présentons Rover. Ensuite, nous présentons CASCADE. Enfin, nous résumons les résultats de quelques travaux de recherche relatifs à la gestion des déconnexions.

Rover

Rover est un projet de recherche au MIT (*Massachusetts Institute of Technology*) [76, 77]. Son objectif est d'offrir une adaptation aux environnements mobiles en collaboration entre l'application et le système. Il propose un modèle de programmation basé sur deux concepts : objet dynamique ré-adressable RDO (en anglais, *Relocatable Dynamic Object*) et appel de procédure à distance non-bloquant QRPC (en anglais, *Queued Remote Procedure Call*). Un RDO est un objet qui peut être dynamiquement chargé sur le client à partir du serveur. Ce sont des répliques des objets du serveur. Les QRPC constituent un système de communication qui permet aux applications de continuer à faire des RPC non-bloquants lorsque la machine est déconnectée. Les requêtes et les réponses sont journalisées puis échangées après la reconnexion.

L'architecture de Rover est structurée en trois couches, application, support système et transport, et composée de quatre composantes (cf. figure 3.7). Le gestionnaire d'accès est responsable du traitement de toutes les interactions entre les applications clientes et les serveurs, et entre les applications clientes elles-mêmes. Le gestionnaire d'accès est responsable du traitement des demandes d'objets par les clients. Il permet aussi de gérer la connectivité avec le serveur et de contrôler le cache des objets. Le cache d'objets se compose d'un cache privé local situé dans l'espace d'adressage de l'application et d'un cache global partagé situé dans l'espace d'adressage du gestionnaire d'accès. Le journal des QRPC permet d'enregistrer les requêtes

envoyées par le client vers les objets locaux pour les exécuter sur les objets distants. Enfin, le gestionnaire du réseau met en œuvre un mécanisme de contrôle de la bande passante et optimise la transmission du journal en regroupant les opérations destinées au même serveur.

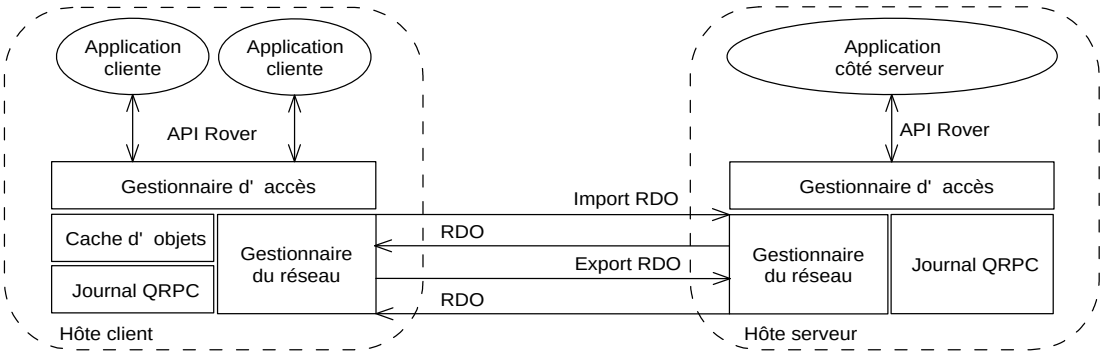


FIG. 3.7 – Architecture de Rover

En mode connecté, lorsque le client invoque une méthode d'un objet, le gestionnaire d'accès contrôle d'abord si l'objet réside dans le cache. Dans ce cas, Rover applique l'invocation directement sur l'objet du cache. Ces objets sont marqués provisoirement valides. Ils seront marqués valides lorsque les invocations seront accomplies sur les objets dans les serveurs. Si l'objet est non présent dans le cache, Rover importe le RDO correspondant et invoque les méthodes fournies par cet objet. En modes déconnecté et partiellement connecté, comme dans le mode connecté, Rover utilise les objets dans le cache s'ils existent. Si les objets n'existent pas, Rover utilise des requêtes QRPC, sauvegarde ces QRPC dans le journal d'opérations et retourne le contrôle à l'application. Ainsi, les applications peuvent continuer de fonctionner même dans les modes déconnecté et partiellement connecté sans que les objets n'existent dans le cache.

Rover propose deux stratégies de déploiement : la première stratégie consiste à donner à l'utilisateur la possibilité de choisir les objets à déployer (via une interface graphique) au lancement de l'application. La deuxième stratégie consiste à déployer les objets à la première invocation (stratégie à la demande). Rover laisse la liberté de prendre d'autres paramètres dans le choix des objets à déployer. Par exemple, dans l'application de messagerie électronique Exmh de Rover [76], la stratégie de déploiement prend en considération la taille des messages. Rover expose cette information à l'utilisateur afin de l'aider à choisir les objets à déployer. Par contre, Rover ne propose aucune stratégie de remplacement.

CASCADE

CASCADE propose un service de gestion de cache pour objets CORBA [32]. Ce service est fourni par plusieurs serveurs de cache appelés DCS (*Domain Caching Servers*). Chaque DCS est responsable d'un domaine logique qui peut correspondre à une zone géographique comme dans le DNS. Le mécanisme de réplication des objets est hiérarchique et dynamique. À chaque fois qu'un client invoque un objet distant, CASCADE déploie une copie de cet objet dans le DCS associé au client. Ainsi, le déploiement des copies dans les DCS se fait à la première invocation.

De plus, ce mécanisme garantit que la copie est obtenue à partir du DCS le plus proche qui héberge une copie de l'objet. Le DCS comportant la copie originale de l'objet est la racine de l'hiérarchie. Tous les DCS sont enregistrés dans le service de nommage et le client utilise ce dernier pour récupérer la référence de son DCS.

Le service de gestion de cache de CASCADE n'est pas destiné à la gestion des déconnexions (pas de cache local). Néanmoins, il propose des stratégies de déploiement et de remplacement de cache. Pour la stratégie de déploiement, lorsque le client décide de travailler sur la copie de son DCS, il invoque ce dernier afin qu'il récupère une copie de l'objet. Si le DCS ne dispose pas d'une copie de l'objet, il contacte le DCS racine de cet objet pour récupérer une copie et joindre la hiérarchie. Une fois la copie déployée sur le DCS local, le client utilise uniquement cette copie. La stratégie de remplacement utilise deux paramètres : le nombre maximum d'objets à déployer sur chaque DCS et la taille maximale de chaque objet dans chaque DCS. Lorsque le nombre d'objet dans un DCS atteint cette limite, il exécute l'algorithme LRU pour supprimer des objets déjà déployés jusqu'à ce que l'espace mémoire devienne suffisant pour les nouveaux objets. Par ailleurs, tout objet dans le DCS dépassant sa taille maximale sera supprimé. [6] propose deux autres algorithmes pour le replacement dans CASCADE : *H-BASED* et *LFU-H-BASED*. *H-BASED* définit pour chaque objet dans le cache la méta-donnée « clé » qui représente le nombre de fois où l'objet a été supprimé pour le remplacement. Dans cet algorithme, c'est l'objet avec la plus petite clé qui sera choisi pour le remplacement. En plus de la clé, *LFU-H-BASED* ajoute la méta-donnée « priorité » entre les objets dans le cache. Cette priorité est dynamique et elle est calculée par CASCADE. Lorsqu'un objet doit être supprimé du cache, *LFU-H-BASED* choisit l'objet le moins prioritaire. Si plusieurs objets sont moins prioritaires, l'algorithme *H-BASED* est appliqué.

La structure des copies secondaires est différente de la structure de la copie originale. En effet, chaque copie secondaire est encapsulée dans un module qui comporte en plus de l'objet CORBA, des méta-données et un contrôleur de requêtes. Les méta-données sont utilisées pour décrire la copie (localisation, DCS racine...). Le contrôleur des requêtes reçoit toutes les requêtes destinées à l'objet pour ajouter des traitements extrafonctionnels (persistance, sécurité...) avant de les rediriger vers l'objet CORBA. De plus, CASCADE utilise des intercepteurs au niveau client pour communiquer avec le DCS local (création de la copie et redirection des requêtes) d'une manière transparente. L'implantation actuelle de CASCADE utilise les intercepteurs de VisiBroker [168].

Autres projets

Dans le contexte de CORBA, deux projets, Π^2 [141] et ALICE [101], traitent le problème de la mobilité du terminal (cf. section 1.2.1) dans le contexte de CORBA sans fil (*Wireless CORBA*). L'architecture de Π^2 se base sur la mise au point de deux caches (un sur le terminal mobile et un autre sur un serveur distant dans le réseau filaire). ALICE utilise uniquement un cache local. Il fournit des mécanismes pour supporter les déconnexions volontaires et involontaires. Dans ALICE, quand une déconnexion se produit, une exception est envoyée par l'ORB à l'application cliente. Le code nécessaire pour la commutation entre les objets déconnectés et les objets distants doit être inclus dans le code de l'application.

Dans le contexte des intergiciels réflexifs, XMIDDLE [25] et CARISMA [26], tous les deux définis dans le laboratoire de l'Université College de Londres, en Angleterre. XMIDDLE fournit une approche de gestion des déconnexions volontaires et involontaires. Les déconnexions volontaires sont gérées au niveau de l'application et les déconnexions involontaires sont gérées au niveau de l'intergiciel. XMIDDLE définit un profil de l'intergiciel qui décrit ce que ce dernier doit faire lorsqu'il se trouve dans un contexte particulier. Ce profil contient des informations sur les ressources du terminal mobile (bande passante, batterie...) et il ne prend pas en compte la sémantique de l'application. Ce type de profil a été aussi introduit dans CARISMA, qui associe à chaque application un fichier de description de comportements.

3.3.6 Système orienté composant

Il existe peu de solution de gestion des déconnexions pour applications à base de composants. Cette limitation est due en particulier à la nouveauté de ce modèle de conception. Nous débutons cette section par la présentation de ACHILLES, un canevas logiciel pour la gestion des déconnexions. ACHILLES n'exploite pas toutes les caractéristiques des composants, en particulier, le paradigme composant/conteneur. Ensuite, nous présentons les travaux de recherche relatifs à la gestion de duplication dans le cadre du projet SARDES.

ACHILLES

ACHILLES [87] est un canevas logiciel pour la gestion des déconnexions volontaires. La granularité utilisée pour la gestion de cache est le composant. Un composant peut être la totalité ou une partie d'un logiciel. Chaque composant est représenté par un paquetage qui comporte le code source du composant. ACHILLES propose un cache local et l'utilisateur utilise systématiquement les composants déployés localement.

ACHILLES propose deux stratégies de déploiement : manuelle et automatique. Dans la première stratégie, c'est à l'utilisateur de décider quels sont les composants à déployer localement via une interface graphique. L'utilisateur peut faire la différence entre des composants permanents qui doivent être toujours présents dans le cache, et des composants qui peuvent être remplacés selon les besoins. Dans la deuxième stratégie, un composant est déployé à la première invocation. Cependant, avant qu'un composant ne soit déployé dans le cache, ACHILLES contrôle la « qualification » du composant, c'est-à-dire l'ensemble des ressources dont le terminal mobile doit disposer pour que le composant puisse être utilisé localement. Ces ressources peuvent être des composants matériels (CPU, mémoire...) ou des composants logiciels. ACHILLES modélise les relations entres les ressources dans un graphe de dépendances dans lequel les nœuds dénotent les ressources et les liens dénotent les relations d'utilisation. Ce graphe de dépendances est utilisé dans la stratégie de déploiement. Ainsi, avant de déployer un composant, ACHILLES vérifie la disponibilité des autres ressources et les déploie si possible.

ACHILLES propose aussi deux stratégies de remplacement : manuelle et automatique. Dans la première stratégie, l'utilisateur peut à tout moment supprimer un composant du cache. Cependant, ACHILLES ne considère pas le cas où l'utilisateur supprime un composant utilisé par

d'autres composants. Dans la deuxième stratégie, le contenu du cache est contrôlé par un algorithme de remplacement appelé algorithme du coût minimum. Le coût d'un composant prend en considération deux paramètres : le coût de redéploiement du composant une fois supprimé et l'importance du composant. Le coût de redéploiement est fonction de la taille du paquetage et de la bande passante disponible. L'importance d'un composant est le nombre de composants dans le cache qui dépendent de lui.

SARDES

Les travaux de recherche dans le projet SARDES portent sur la construction d'infrastructures réparties adaptables [143]. En particulier, ils s'intéressent aux environnements à grande échelle dont les ressources de calcul changent dynamiquement et étudient les mécanismes d'adaptation préservant la qualité de service des applications s'exécutant dans de tels environnements. Ils portent également sur la reconfiguration de la duplication de composants des applications réparties. Les travaux de recherche concernant cette dernière thématique sont publiés dans la thèse référencée dans [105]. Cette thèse propose une méthodologie de configuration non intrusive de la duplication et de la cohérence en se basant sur la séparation entre les préoccupations fonctionnelles et les préoccupations extrafonctionnelles des composants. Elle permet la réutilisation du code de gestion de la duplication et de la cohérence en modélisant les protocoles en termes de composants qui sont indépendants des applications métier. Le modèle de composants utilisé dans ce protocole respecte le paradigme composant/conteneur décrit dans la section 2.3.2.

Comme domaine d'utilisation de la duplication, [105] propose *DisconnectP*, un protocole de gestion des déconnexions volontaires pour applications à base de composants. Dans ce protocole, chaque composant client est surchargé par une interface requise `Connection_itf`. Cette interface est connectée au composant `Gestionnaire` sur le terminal mobile. Ce dernier fournit les primitives nécessaires à la gestion de la cohérence d'un composant et définit les traitements de reconfiguration lors des déconnexions et des reconnexions d'un client en utilisant le principe d'opération déconnectée. Dans *DisconnectP*, le composant distant est surchargé par trois interfaces : deux interfaces offertes `LogControl_itf` et `State_itf` utilisées par le composant `Gestionnaire`, et une interface requise (`Log_itf`). Cette interface est connectée au composant `Journal` dans le serveur. Le composant `Journal` définit les traitements génériques de gestion d'opérations effectuées sur le composant distant, mais repose sur des traitements spécifiques dont l'implantation dépend de la sémantique de l'application. En outre, le composant déconnecté est surchargé par deux interfaces : une offerte (`State_itf`) utilisée par le composant `Gestionnaire` pour la gestion de l'état, et une requise (`Log_itf`) connectée au composant `Journal` sur le terminal mobile. Ce composant est connecté au composant `Journal` sur le serveur pour la gestion de la réconciliation.

La stratégie de déploiement utilisée dans *DisconnectP* est à la demande de l'utilisateur. Cependant, le système de gestion du cache incarné par les composants `Gestionnaire` et `Journal` ne propose ni stratégie de remplacement ni gestion de dépendances entre les composants de l'application.

3.4 Synthèse

Dans le début de ce chapitre, nous avons étudié les besoins en terme d'adaptation aux déconnexions des terminaux mobiles. Cette étude montre trois caractéristiques qui définissent l'adaptation. Tout d'abord, la stratégie d'adaptation définit les acteurs de l'adaptation. Nous en avons énuméré trois : laissez-faire, transparence et collaboration. Ensuite, le type d'adaptation définit le moment de l'adaptation. Il peut être statique, dynamique ou auto-adaptable. Enfin, le mécanisme d'adaptation concerne la procédure et les algorithmes de l'adaptation, opérant ainsi un lien entre la stratégie et le type d'adaptation. Aussi, face à notre problématique, il paraît préférable d'envisager un mixage entre les différents types d'adaptation. Cette adaptation est effectuée suivant la stratégie collaboration. Le choix de la stratégie collaboration est motivé par la non-adéquation des stratégies laissez-faire et transparence.

La continuité de service entre le client et les serveurs en présence des déconnexions est assurée par le maintien d'une connexion logique en utilisant le concept d'opération déconnectée. Dans ce concept, nous avons identifié quatre intervenants : le client sur l'hôte mobile, l'entité distante de l'application répartie, le cache qui représente une partie de la mémoire du terminal mobile et l'entité déconnectée, un mandataire en terme de code et d'état de l'entité distante dans le cache. Nous avons ensuite présenté le fonctionnement des opérations déconnectées. Ce fonctionnement fait apparaître quatre mécanismes de base : la gestion du cache du terminal mobile, la gestion de la cohérence des différentes entités, la détection des déconnexions et la commutation entre l'entité distante et l'entité déconnectée. Le travail que nous présentons dans cette thèse aborde uniquement les thématiques gestion du cache, et commutation entre l'entité distante et l'entité déconnectée.

Par ailleurs, bien que dans la suite de ce document le modèle de conception et de programmation retenu soit orienté composants, nous avons donné dans ce chapitre une étude des différentes solutions de gestion des déconnexions en considérant les granularités fichier, base de données, page web, objet et composant. Cette étude est basée sur plusieurs critères d'étude, mettant en évidence les apports de chaque solution dans chaque critère. Le tableau 3.2 donne un récapitulatif des différentes solutions étudiées. La plupart de ces solutions ne proposent pas une séparation entre les préoccupations fonctionnelles de l'application et la gestion des déconnexions. Cette contrainte limite les possibilités de maintenance, de réutilisation et de reconfiguration. Ces solutions ne proposent pas non plus de modèle de conception d'applications réparties devant fonctionner en présence des déconnexions. En outre, le modèle orienté composant est peu investi dans la gestion des déconnexions ; cette dernière limitation étant due à la nouveauté de ce modèle.

La partie suivante de cette thèse présente nos solutions pour la gestion des déconnexions pour applications à base de composants.

	Granularité	Type de déconnexion	Type de cache	Mode de fonctionnement	Type d'adaptation	Stratégie de déploiement	Stratégie de remplacement	Modèle de conception	Traitement des dépendances
Coda	Fichier	Volontaire et involontaire	Local et réparti	Connecté, partiellement connecté et déconnecté	Statique et dynamique en collaboration	Pré-chargement et à la demande	LRU	Profi l utilisateur, données implicites et explicites	
Seer	projet		Local		Dynamique en transparence	Prédiction		Projet et distance sémantique	Intra-projet
Amigos	Fichier et répertoire	Volontaire et involontaire	Local	Connecté et déconnecté	Statique et dynamique en collaboration	Pré-chargement	À la demande	Profi l utilisateur	
Bayou	Base de données	Volontaire et involontaire	Local et réparti	Connecté, partiellement connecté et déconnecté	Statique et dynamique en collaboration	À la demande		*Read-any/Write-any*, opération R/W et protocole anti-entropique	
WebExpress	Page web	Involontaire	Local et réparti	Connecté, partiellement connecté et déconnecté	Statique en transparence et dynamique en collaboration	À la première invocation	LRU	Serveur web local, intervalle de cohérence et pages web infi niment persistantes	
SliCache	Page web	Involontaire	Local	Connecté et déconnecté	Statique en transparence		Prédiction	Partitionnement du cache (*R-slice* et *I-slice*)	
Rover	objet	Volontaire et involontaire	Local (privé et partagé)	Connecté, partiellement connecté et déconnecté	Statique et dynamique en collaboration	Pré-chargement et à la demande		RDO, QRPC et canevas logiciel	
CASCADE	Objet CORBA		Réparti hiérarchique	Connecté	Statique et dynamique en collaboration	À la demande	LRU, H-BASED et LFU-H-BASED	Structuration de l'application cliente	
ACHILLES	Composant	Volontaire	Local	Connecté et déconnecté	Statique et dynamique en collaboration	Manuelle et automatique	Manuelle et automatique (coût minimum)	Qualifi cation du composant, composant permanent et canevas logiciel	Composants et ressources du terminal
DisconnectP	Composant	Volontaire	Local	Connecté et déconnecté	Statique et dynamique en collaboration	À la demande		Protocole de gestion des déconnexions	

TAB. 3.2 – Récapitulatif des principales solutions de la littérature

Deuxième partie

Contribution

Chapitre 4

MADA

La première partie de ce document nous a permis d'abord de nous familiariser avec les environnements mobiles (cf. chapitres 1) et le monde des intergiciels (cf. chapitres 2). Ensuite, nous avons parcouru à travers le chapitre 3 une synthèse de la littérature concernant la gestion de cache pour la gestion des déconnexions. Dans ce chapitre, nous présentons notre approche de conception d'applications pouvant fonctionner en présence des déconnexions nommée MADA.

Dans un premier temps, nous présentons les motivations et les objectifs de l'approche MADA dans la section 4.1 et donnons une présentation générale de cette approche dans la section 4.2. Nous décrivons dans la section 4.3 le méta-modèle de MADA, avant de présenter la modélisation de la gestion des déconnexions (cf. section 4.4) et le graphe de dépendances des différentes entités de l'application (cf. section 4.5). Enfin, la section 4.6 situe notre approche par rapport à l'existant et la section 4.7 résume le chapitre.

4.1 Motivations et objectifs

Le travail que nous présentons dans cette thèse s'intéresse à la gestion des déconnexions pour applications réparties à base de composants dans les environnements mobiles. Comme étudié dans le chapitre 3, la solution de base pour la gestion des déconnexions consiste à maintenir une connexion logique entre le client et le serveur en utilisant le concept d'opération déconnectée. Cette solution a été utilisée dans plusieurs approches. Cependant, la plupart de ces approches sont souvent des réponses « ad hoc » à des contextes précis. En effet, ces solutions ne proposent pas une séparation entre les préoccupations fonctionnelles de l'application et la gestion des déconnexions. Cette contrainte limite les possibilités de maintenance, de réutilisation et de reconfiguration. Ces solutions ne proposent pas non plus de modèle de conception d'applications réparties pouvant fonctionner en présence des déconnexions.

En outre, la construction d'applications réparties converge de plus en plus vers l'utilisation des intergiciels orientés composants (CCM, EJB, .Net. . .). La conception orientée composants offre une meilleure séparation entre les préoccupations fonctionnelles et extrafonctionnelles réalisée

suivant le paradigme composant/conteneur. Les solutions de gestion des déconnexions que nous avons étudiées dans le chapitre 3 n'exploitent pas les caractéristiques des composants, cette dernière limitation étant principalement due à la nouveauté de ce modèle de conception. Par ailleurs, 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. Dans notre approche, MDA nous permet de modéliser la gestion des déconnexions pour les prendre en compte dans l'architecture de l'application.

4.2 Présentation générale de MADA

Avant de donner une présentation générale de l'approche MADA[1] (cf. section 4.2.4), nous décrivons dans la section 4.2.1 l'application exemple que nous utilisons dans ce chapitre afin d'illustrer cette approche. Nous présentons ensuite deux concepts utilisés dans MADA : le modèle « 4+1 » vues (cf. section 4.2.2) et le patron de conception Façade (cf. section 4.2.3).

4.2.1 Présentation de l'application exemple

Afin de mieux comprendre l'approche MADA, nous l'illustrons par une application de réservation et d'achat de billets d'avions par Internet que nous appelons *InternetTicket*. La figure 4.1 schématise le diagramme UML des cas d'utilisation de cette application. L'application *InternetTicket* est accessible via une interface graphique à partir de laquelle un utilisateur peut réserver un billet d'avion, acheter un billet déjà réservé, réserver une chambre d'hôtel, consulter les tarifs des billets d'avion et d'hôtel, et annuler la réservation d'un billet d'avion ou d'une chambre d'hôtel. Toutes ces fonctionnalités sont accessibles suivant le paradigme client/serveur. Nous décrivons la structuration de l'application *InternetTicket* au fur et à mesure de la description de MADA.

4.2.2 Modèle « 4+1 » vues de l'architecture

Le modèle « 4+1 » vues de l'architecture [93] 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. L'utilisation de vues concourantes permet d'adresser séparément les intérêts des divers groupes d'intervenants (architectes, utilisateurs, développeurs, chefs de projets. . .), et ainsi de mieux séparer les préoccupations fonctionnelles et les préoccupations extrafonctionnelles d'une application.

La figure 4.2 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

[1] MADA est l'acronyme de *Mobile Application Development Approach*.

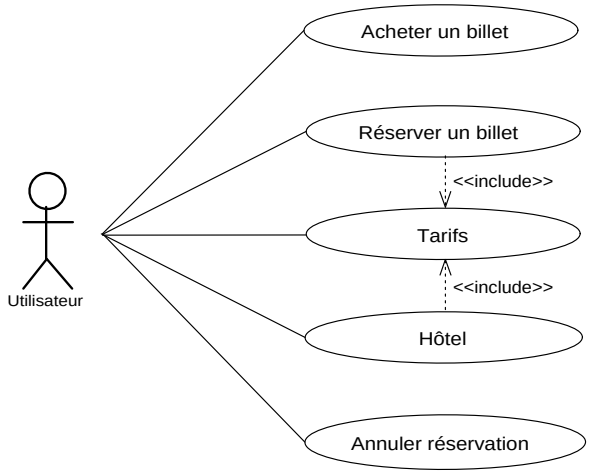


FIG. 4.1 – Diagramme des cas d'utilisation de l'application *InternetTicket*

de la conception, et les décompose en flots d'exécution (processus, fil d'exécution...). Elle se rapporte aux objets actifs et aux interactions. La vue « développement » représente l'organisation statique des modules (exécutable, codes source, 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 de l'application. Cette dernière vue valide en quelque sorte les autres vues et assure la cohérence globale.

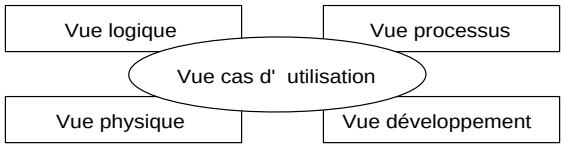


FIG. 4.2 – Modèle « 4+1 » vues de l'architecture

Les concepts que nous présentons dans l'approche MADA sont suffisamment généraux pour être utilisés avec d'autres modèles d'architecture que le modèle « 4+1 » vues à partir du moment où ils séparent les préoccupations fonctionnelles et extrafonctionnelles et utilisent un processus centré sur l'architecture logicielle et piloté par des cas d'utilisation. La décomposition de la description d'une architecture en plusieurs vues existe déjà dans le modèle RM-ODP [68]. Ce dernier 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.

4.2.3 Patron de conception Façade

Le patron de conception Façade [53] permet de fournir un accès unifié à un groupe de services (ou interfaces) d'un sous-système. La façade réduit 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 4.3-a représente une application répartie « sans façade » et la figure 4.3-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 contrôle le mode de fonctionnement : par exemple, pour travailler 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 ou de modifier ces méthodes.

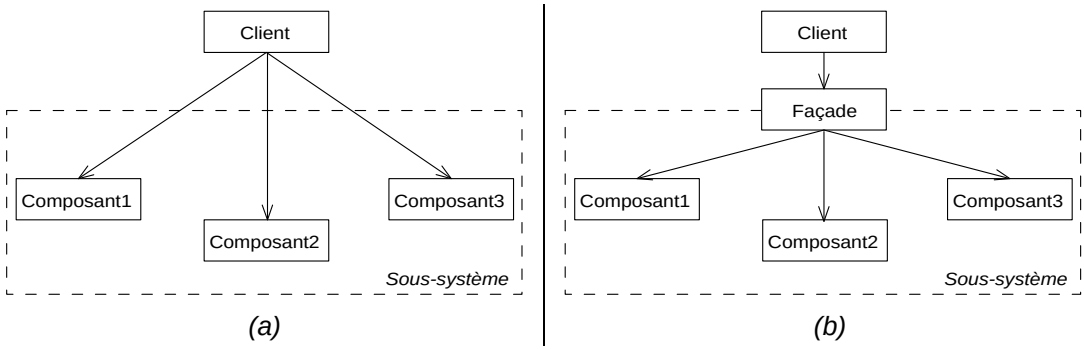


FIG. 4.3 – Application avec et sans façade

4.2.4 Présentation générale de MADA

L'approche MADA est une méthode centrée sur l'architecture logicielle et pilotée par les cas d'utilisation (c.-à-d., partant des fonctionnalités fournies aux utilisateurs finaux), et s'intégrant dans l'approche MDA. L'approche MADA suit l'approche MDA comme l'indiquent les figures 4.4-a et 4.4-b, respectivement. Dans la figure 4.4-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 [41] qui cible les applications construites à base de composants. Ce modèle est indépendant de la plateforme d'implantation et d'exécution (cf. section 2.3.1). Il est transformé en un modèle ayant pour cible le monde CCM et respectant le profil UML CCM [125]. Par analogie, dans la figure 4.4-a, l'architecture logicielle de l'application est surchargée de spécifications dédiées à la gestion des déconnexions (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. Ce dernier donne une modélisation UML d'une application à base de composants CCM tout en prenant en compte la gestion des déconnexions. Nous décrivons UML4CCMDisc dans la section 4.4.3. Dans cette thèse, ne désirant pas fournir un atelier de génie logiciel, nous considérons que la transformation PIMDisc vers PSMDisc existe. En outre, nous nous limitons à une modélisation dépendante de la plateforme CCM. Ainsi, le profil EDOCDisc n'est pas développé.

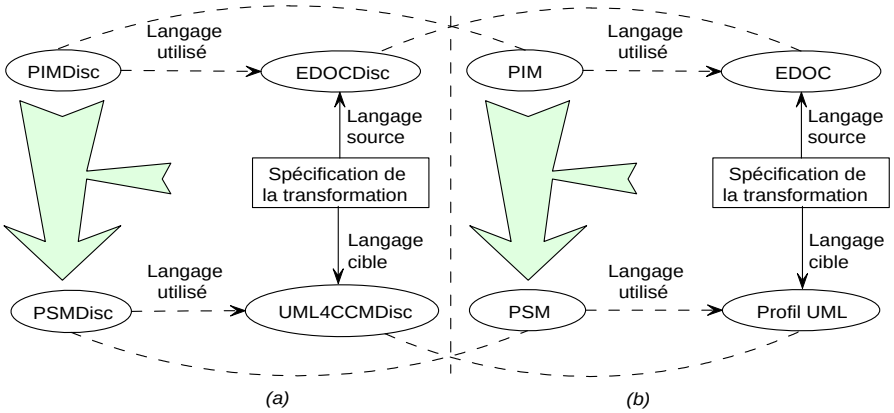


FIG. 4.4 – Vision MDA de MADA

4.3 Méta-modèle de MADA

Dans cette section, nous nous limitons à une brève présentation du méta-modèle de MADA. Nous décrivons en détail chaque élément de ce méta-modèle dans la suite de ce chapitre. La figure 4.5 présente le méta-modèle de MADA. Chaque application est l'assemblage de plusieurs composants qui interagissent entre eux pour réaliser les fonctionnalités de l'application. Dans MADA, les fonctionnalités de l'application sont vues comme des services de l'application qui interagissent entre eux (interactions inter-services). En outre, les entités de l'application (composants et services) sont décrites dans un profil de l'application modélisé en EDOCDisc dans le PIM et en UML4CCMDisc dans le PSM (cf. section 4.2.4). Dans ce profil, l'architecte de l'application assigne pour chaque entité de l'application trois méta-données, représentées par les méta-éléments *Déconnectablité*, *Nécessité* et *Priorité*. La *déconnectabilité* indique si une entité de l'application peut avoir une entité déconnectée sur le terminal mobile. La *nécessité* indique si la présence de l'entité déconnectée dans le cache du terminal mobile est absolument nécessaire pour le fonctionnement de l'application en présence des déconnexions. La *priorité* permet d'ordonner les entités de l'application quand à leur présence dans le cache du terminal mobile. La déconnectabilité des entités est uniquement attribuée par l'architecte de l'application. L'utilisateur quand à lui, peut changer la nécessité et la priorité des entités suivant des règles que nous présentons dans la section 4.5.2. L'affectation des méta-données par l'architecte de l'application correspond à une adaptation statique, tandis que le changement de la nécessité et de la priorité

par l'utilisateur correspond à une adaptation dynamique (cf. section 3.1.1). Nous décrivons ces méta-données avec plus de détails dans la section 4.4.2.

Les interactions intra- et inter-services sont représentées dans un graphe de dépendances. Il sert à la propagation de la *nécessité* entre les entités de l'application. Chaque utilisateur de l'application peut avoir son propre graphe de dépendances. La différence entre ces graphes ne réside pas dans la structure mais dans la *nécessité* des nœuds du graphe (cf. section 4.5). En outre, le graphe de dépendances est utilisé dans la gestion du cache du terminal mobile, plus précisément dans les stratégies de déploiement et de remplacement que nous présentons dans le chapitre 5.

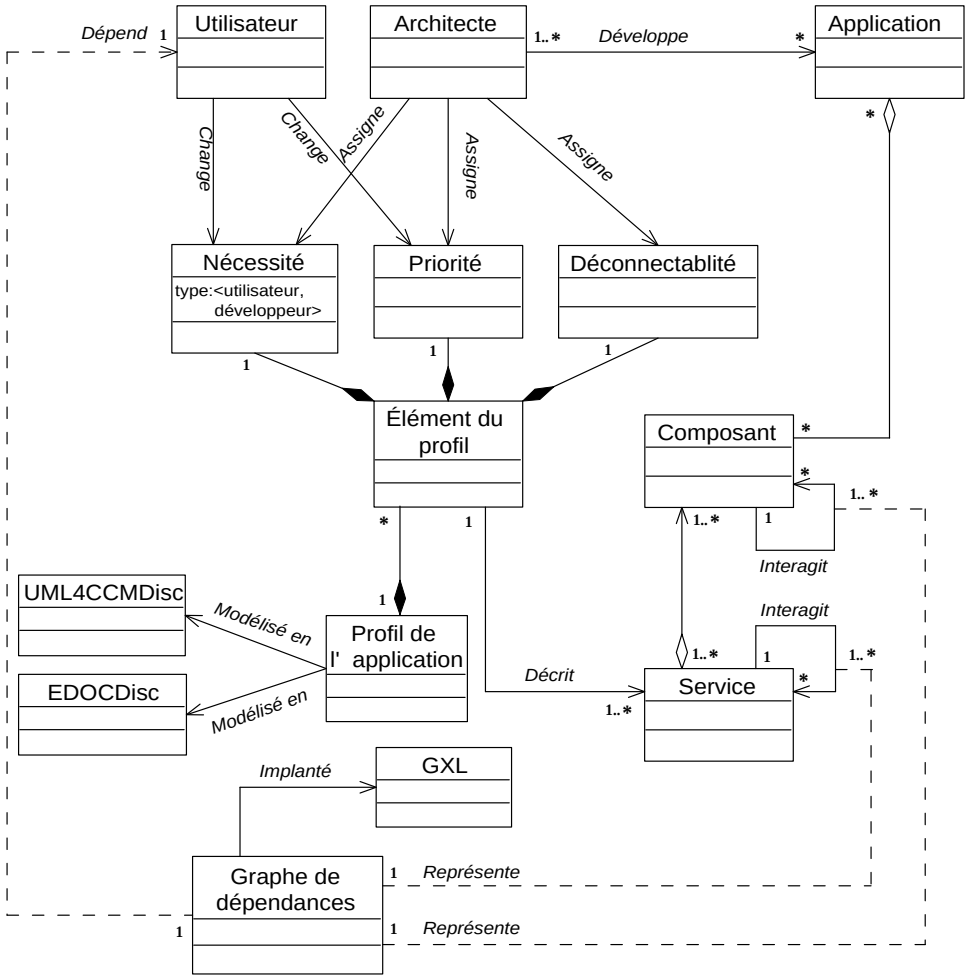


FIG. 4.5 – Méta-modèle de MADA

4.4 Modélisation de l'architecture logicielle

Dans la section 4.3, nous avons donné un aperçu des différents concepts de l'approche MADA. Dans cette section, nous décrivons en détail trois concepts de MADA : la notion de service (cf. section 4.4.1), le profil de l'application (cf. section 4.4.2), et le profil UML4CCMDisc (cf. section 4.4.3). Nous terminons la section par la description des différente vues du modèle « 4+1 » vues dans le cadre de l'approche MADA (cf. section 4.4.4).

4.4.1 Notion de service

Dans le modèle de conception orienté composant, le composant utilise et fournit des fonctionnalités accessibles à travers des connexions entre les composants. L'application répartie est vue comme un assemblage de plusieurs composants qui interagissent entre eux pour accomplir les fonctionnalités de l'application. Chaque ensemble d'interactions est vu comme une composition logique de composants qui définit un service de l'application. Ainsi, l'application dans son ensemble peut être vue comme un ensemble de services. Dans l'approche MADA, l'architecte définit les services de l'application en utilisant les diagrammes UML des cas d'utilisation. En effet, un cas d'utilisation de l'application représente une fonctionnalité offerte par l'application. Ainsi, nous considérons que chaque cas d'utilisation n'est autre qu'un service de l'application. Ces services sont accessibles par les utilisateurs via une interface homme-machine agissant en tant que façade. Par exemple, d'après le diagramme UML des cas d'utilisation (cf. figure 4.1), l'application *InternetTicket* offre cinq services : un service de paiement de billets, un service de réservation de billets, un service d'obtention de tarifs, un service de réservation de chambres d'hôtel et un service d'annulation de réservations.

Par ailleurs, comme c'est le cas pour les composants, l'utilisation d'un service peut mettre en collaboration d'autres services de l'application. Par exemple, les services réservation de billets et réservation d'hôtel de l'application *InternetTicket* utilisent le service tarifs qui peut être utilisé directement par les utilisateurs. De ce fait, nous définissons deux types d'interactions : intra-services (entre les composants dans le même service) et inter-services (entre les services de l'application). Ces interactions sont utilisées dans les stratégies de déploiement et du remplacement du cache. Nous décrivons ces interactions dans la section 4.5.

4.4.2 Méta-données pour la gestion du cache

Dans [90], nous introduisons un patron de conception pour la conception d'un profil de l'application pour la gestion des déconnexions. Ce profil est basé sur trois méta-données : la *déconnectabilité*, la *nécessité* et la *priorité*. Dans cette section, nous décrivons d'abord ces méta-données, ensuite, nous appliquons ce patron de conception sur les composants et les services de l'application.

La méta-donnée *déconnectabilité* indique si une entité distante de l'application résidant sur un hôte distant peut avoir une entité déconnectée sur le terminal mobile. Si c'est le cas, l'entité

distante est *déconnectable*. L'entité à déployer localement est dite *l'entité déconnecté*. C'est l'architecte de l'application qui attribue la méta-donnée déconnectabilité aux entités de l'application car il a la meilleure connaissance de la sémantique de cette dernière. En outre, la déconnectabilité implique des contraintes de conception ou de déploiement à respecter. Par exemple, pour des raisons de sécurité, l'architecte peut décider de déployer quelques entités sur des serveurs sécurisés et d'empêcher la duplication de ces entités sur d'autres machines (par exemple, des terminaux mobiles), empêchant alors que les entités soient déconnectables.

La méta-donnée *nécessité* permet de donner un « poids » aux entités déconnectées de l'application quant à leur présence dans le cache du terminal mobile. Une entité nécessaire est une entité dont la présence dans le terminal mobile est obligatoire pour le fonctionnement de l'application en présence des déconnexions. Une entité non nécessaire est une entité dont l'absence dans le terminal mobile n'empêche pas le fonctionnement global de l'application en présence des déconnexions. L'application doit donc être construite en conséquence. Contrairement à la méta-donnée *déconnectabilité*, la méta-donnée *nécessité* est dynamique. Le développeur fournit une première classification des entités déconnectables en nécessaires développeur et non nécessaires. Cette classification est utilisée pour construire un profil de l'application par défaut. En cours d'exécution, l'utilisateur peut personnaliser le profil en changeant la nécessité de quelques entités non nécessaires pour qu'elles deviennent nécessaires utilisateur. Nous décrivons le mécanismes de changement de la nécessité dans la section 4.5.2.

La méta-donnée *priorité* indique la priorité entre les entités non nécessaires et entre les entités nécessaires utilisateur. Cette méta-donnée est utilisée dans les stratégies de déploiement et de remplacement du cache (cf. chapitre 5) par exemple, dans le choix des entités déconnectées à supprimer du cache lorsque ce dernier est saturé.

Dans MADA, nous appliquons l'attribution des trois méta-données aux composants de l'application. De ce fait, nous définissons les notions de composant déconnectable et non déconnectable, de composant nécessaire (développeur et utilisateur) et de composant non nécessaire, la priorité entre composants nécessaires utilisateurs, et la priorité entre composants non nécessaires. Par analogie, nous appliquons ces méta-données au concept de service. Ainsi, nous définissons un service déconnectable comme un service de l'application pouvant être assuré dans le terminal mobile pendant les déconnexions. Un service déconnectable est la composition logique de plusieurs composants déconnectables. En outre, nous définissons un service nécessaire comme un service dont la fourniture obligatoire pendant les déconnexions. La composition associée à un service nécessaire doit contenir au moins un composant nécessaire. Enfin, une priorité est aussi attribuée aux services.

4.4.3 Profil UML pour la déconnexion

Un profil UML est en cours de standardisation par l'OMG pour la modélisation des applications à base de composants CORBA [125]. Nous l'appelons *UML4CCM*. Ce profil est une spécialisation du profil UML pour CORBA [123] 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 IDL3 et la modélisation CIDL. Ces deux modélisations

définissent deux vues complémentaires de l'application. La vue « client » spécifie dans le langage IDL3 les caractéristiques des composants de l'application en terme de ports (facettes, réceptacles...), de maisons 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 la structure interne des composants ainsi que la persistance, si le composant est persistant. Cette structure décrit essentiellement 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 sources d'évènements, 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 que nous ne détaillons pas dans cette thèse. Le lecteur intéressé peut se référer à [125].

Pour la gestion des déconnexions, 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 comme le montre la figure 4.6. Conformément à la démarche de CCM, nous présentons successivement le méta-modèle du nouveau profil (*UML4CCMDisc*) puis le profil UML lui-même.

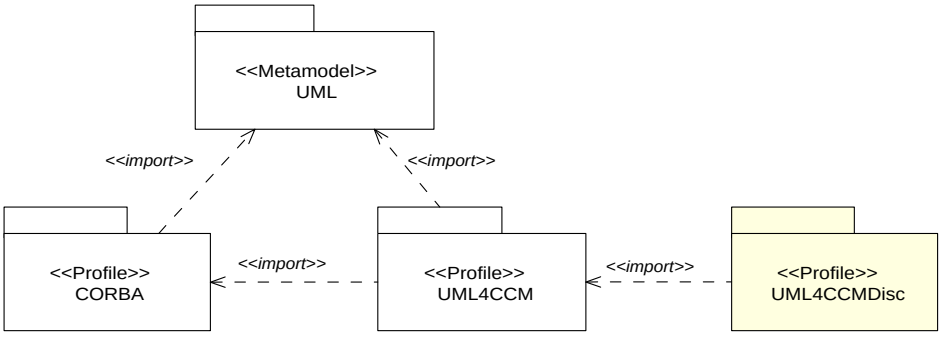


FIG. 4.6 – Dépendances entre le méta-modèle UML et les profils CORBA, UML4CCM et UML4CCMDisc

La figure 4.7 présente le méta-modèle du profil UML4CCMDisc. Un composant déconnectable est représenté par le méta-élément `DisComponentDef`. Conceptuellement, le composant déconnecté est un composant CORBA étendu qui surcharge le composant CORBA d'une sémantique spécifique pour le mode déconnecté, d'où le lien de généralisation avec le méta-élément CCM `ComponentDef`. Un composant déconnecté communique de manière synchrone (offre et requiert des interfaces) et de manière asynchrone (produit et consomme des évènements). Dans le profil UML4CCMDisc, nous représentons ces interfaces et ces évènements de la même manière que dans le profil UML4CCM. Ainsi, les interfaces sont désignées par le méta-élément `InterfaceDef` et les évènements sont désignés par le méta-élément `EventDef`.

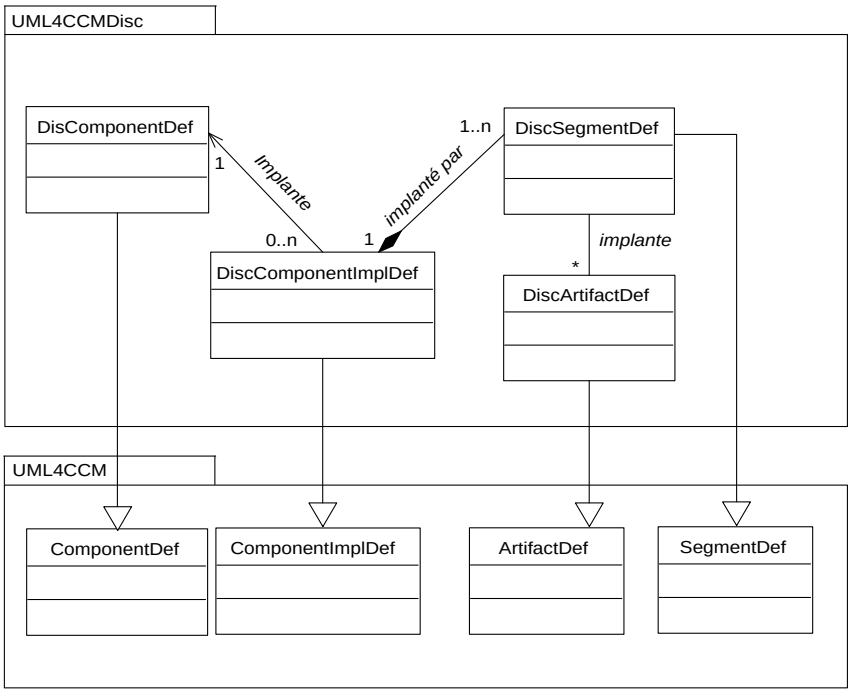


FIG. 4.7 – Méta-modèle de profil UML4CCMDisc

Un composant déconnectable possède également une implantation exprimée par le méta-élément `DisComponentImplDef`. Ce dernier dérive du méta-élément `ComponentImplDef`. Comme pour les composants CCM, une implantation déconnectable peut être répartie sur plusieurs segments. Chaque segment est représenté par le méta-élément `DiscSegmentDef`, d'où le lien de généralisation avec le méta-élément `SegmentDef`.

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.

Comme présenté dans la figure 4.8, le profil UML4CCMDisc contient quatre stéréotypes représentant respectivement la vue client d'un composant déconnectable, l'implantation de ses interfaces[2], l'implantation de ses segments, et l'implantation du composant lui-même. 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. Le tableau 4.1 décrit ces stéréotypes. Un composant CORBA déconnectable est défini en utilisant une classe UML avec le stéréotype `CORBADisComponent`. Cette classe hérite de toutes les caractéristiques de `CORBAComponent`. Par exemple, il est composé de une ou plusieurs `CORBAInterface` (cf.

[2]Le profi l UML4CCM utilise le mot *artifact* pour désigner l'implantation d'une interface. Un artifact peut être représenté par une ou plusieurs classes.

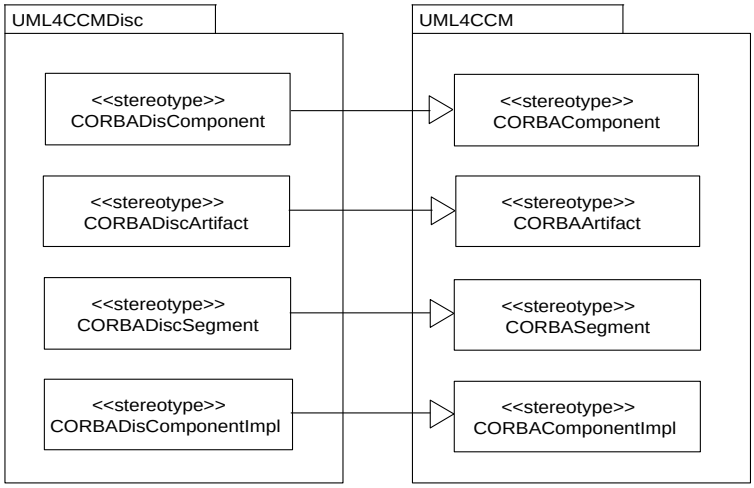


FIG. 4.8 – Profil UML4CCMDisc

l'exemple de la figure 4.9). L'implantation d'un composant déconnectable est défini en utilisant une classe UML stéréotypée avec `CORBADisComponentImpl`. Cette dernière hérite de toutes les caractéristiques de `CORBAComponentImpl`. De plus, l'implantation d'un composant déconnectable est composée de un ou plusieurs segments. Chaque segment est décrit par une classe UML avec le stéréotype `CORBADiscSegment`. Enfin, chaque segment est composé de une ou plusieurs classes UML stéréotypées avec `CORBADiscArtifact` qui représente l'implantation de une ou plusieurs `CORBAInterface`.

Des règles régissent la déconnectabilité des composants et de leurs éléments. Elles servent à vérifier la cohérence de la vue mode déconnecté. Un composant est déconnectable si, et seulement si, le stéréotype `CORBADisComponent` est utilisé dans la modélisation UML de ce composant. De plus, un composant déconnectable contient au moins une implantation déconnectable (utilisation du stéréotype `CORBADisComponentImpl`). Cette dernière doit contenir au moins un segment déconnectable (utilisation du stéréotype `CORBADisSegment`).

Stéréotype	Parent	Élément UML
CORBADisComponent	CORBAComponent	Class
CORBADiscComponentImpl	CORBAComponentImpl	Class
CORBADiscSegment	CORBASegment	Class
CORBADiscArtifact	CORBAArtifact	Class

TAB. 4.1 – Profil UML4CCMDisc

Dans le cas où un composant est non déconnectable, l'architecte modélise ce composant en utilisant le profil UML4CCM. Par contre, si ce composant est déconnectable, l'architecte le modélise en utilisant le profil UML4CCMDisc. Le passage de UML vers IDL3 et de UML vers un langage de programmation (Java, C++...) donne une description du composant et une description de son composant déconnecté. La figure 4.9 donne une représentation UML du composant `AvailableSeat` de notre application exemple. Ce composant appartient au service « réserver

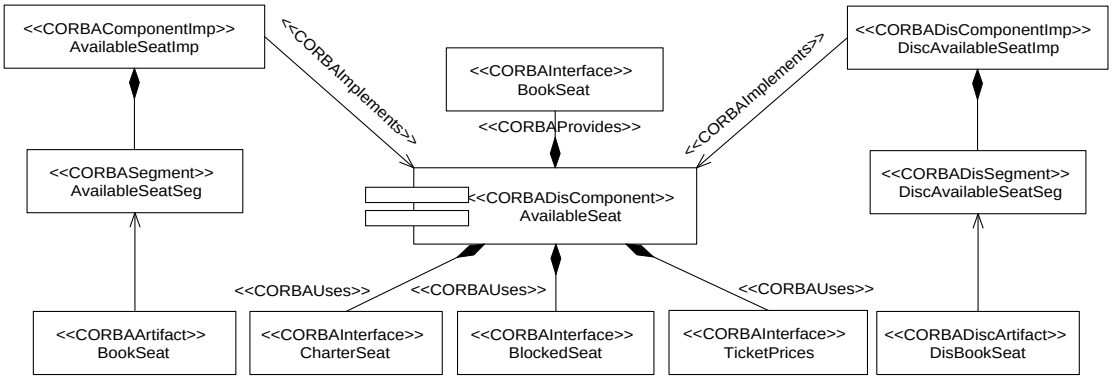


FIG. 4.9 – Représentation du composant `AvailableSeat` en UML4CCMDisc

un billet » considéré comme déconnectable par l'architecte de l'application. De ce fait, le composant `AvailableSeat` est déconnectable. La figure 4.10 présente la déclaration ILD3 résultant du passage de UML vers IDL3 du composant `AvailableSeat`. Ce passage a généré deux composants : `AvailableSeat` (ligne 18) et `DiscAvailableSeat` (ligne 25). Ce dernier est le composant déconnecté du premier composant. De plus, le passage de UML vers le langage d'implantation (Java dans notre cas) génère deux squelettes d'implantation : un pour le composant `AvailableSeat` et un autre pour son composant déconnecté.

En plus des interfaces fonctionnelles offertes, le composant distant doit fournir l'interface `StateTransfer` (ligne 20) utilisé par le composant déconnecté (ligne 27) afin d'initialiser son état. Dans notre approche, l'état d'un composant correspond à la valeur de ses attributs. Ce qui correspond à l'état « passif » du composant. C'est l'approche adoptée, par exemple, par la sérialisation Java [110] ou par le projet sur les agents mobiles Mole [12]. La capture de l'état « actif » d'exécution est surtout considéré dans le cadre de la migration [20] et non de la duplication. Les méthodes de l'interface `StateTransfer` dépendent des attributs du composant distant. Par exemple, le composant `AvailableSeat` dispose de l'attribut *seats* qui donne la description des places disponibles. L'interface `StateTransfer` doit offrir la méthode *initSeats* pour initialiser l'attribut *seats*.

Le gestionnaire du cache que nous présentons dans le chapitre 5 utilise la totalité du composant (implantation monolithique) comme granularité de déploiement et de remplacement. De ce fait, nous supposons que toutes les interfaces et tous les segments d'un composant déconnectable sont déconnectables. L'utilisation des interfaces ou des segments comme granularités manipulées par le gestionnaire du cache est laissée en perspective (nous donnons plus de détails dans la conclusion de cette thèse).

4.4.4 Développement des vues de l'architecture logicielle

Les vues du modèle « 4+1 » vues ne sont pas entièrement indépendantes entre elles. Des éléments d'une vue sont reliés à d'autres éléments dans d'autres vues suivant certaines règles de conception. Par exemple, dans la vue « logique », nous définissons les classes et les associations

```
module InternetTicket{
...
        interface BookSeat{
...
        };
        interface BlockedSeat{
...
            };
        interface TicketPrices{
...
            };
        interface CharterSeat{
...
            };
        interface StateTransfer {
...
            };
        component AvailableSeat {
                provides BookSeat bookSeat;
                provides StateTransfer stateTransfer;
                uses CharterSeat charterSeat;
                uses BlockedSeat blockedSeat;
                uses TicketPrices ticketPrices;
        };
        component DiscAvailableSeat {
                provides BookSeat bookSeat;
                uses StateTransfer stateTransfer;
                uses CharterSeat charterSeat;
                uses BlockedSeat blockedSeat;
                uses TicketPrices ticketPrices;
        };
...
}
```

FIG. 4.10 – Résultat du passage de UML vers IDL3

entres ces classes d'une manière abstraite et statique, et dans la vue « développement », la création des paquetages revient à mettre plusieurs classes dans un seul paquetage. Dans cette section, nous décrivons les différentes vues du modèle « 4+1 » vues de l'architecture dans le cadre de l'approche MADA.

Scénarios et cas d'utilisation

Les cas d'utilisation sont modélisés par des diagrammes UML des cas d'utilisation. Ces diagrammes montrent les divers scénarios d'utilisation de l'application par des acteurs (utilisateur et intergiciel). À partir de ces diagrammes et suivant le cahier des charges de l'application, l'architecte définit les différents services de l'application et classe ces services en services déconnectables et en services non déconnectables. Dans MADA, le comportement du système en présence des déconnexions est modélisé par des cas d'utilisation qui héritent des cas d'utilisation de base. En effet, 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. Dans la figure 4.11, tous les services de l'application *InternetTicket* peuvent être effectués en présence des déconnexions sauf le cas d'utilisation « acheter un billet » pour des raisons de sécurité. Donc, aucun cas d'utilisation spécialisé n'est associé à ce dernier pour le fonctionnement en présence des déconnexions.

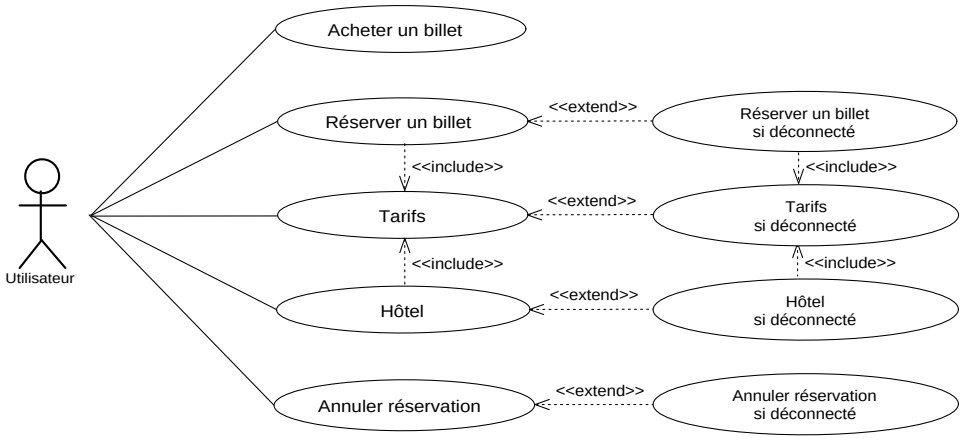


FIG. 4.11 – Diagramme des cas d'utilisation et cas d'utilisation déconnectables

Par ailleurs, le fonctionnement des cas d'utilisation spécialisés peut être différent de celui des cas d'utilisation de base. Par exemple, dans l'application *InternetTicket*, le cas d'utilisation « réserver un billet si déconnecté » remplace le cas d'utilisation « réserver un billet » en présence des déconnexion. Cependant, dans le premier cas, la demande de réservation n'est enregistrée que localement sur le terminal mobile. Elle ne sera exécutée sur le serveur qu'une fois la connexion rétablie.

Vue logique

Dans cette vue, nous appliquons le patron de conception « Façade ». Toutes les opérations demandées par l'utilisateur passent par la façade. Donc, la façade est l'interface directement accessible par l'utilisateur. De plus, avec la notion de service, l'utilisateur n'a pas à connaître tous les composants de l'application, la façade encapsule ou masque ces composants en n'exposant que les points d'entrées des services.

Par ailleurs, comme décrit dans la section 4.4.1, chaque service est réalisé par la collaboration de plusieurs composants. Dans cette vue, l'architecte de l'application commence par la définition des diagrammes UML de composants, de classes et de collaborations (ou de séquences) de chaque service. Ensuite, pour chaque service déconnectable, l'architecte de l'application définit les composants déconnectables en utilisant le profil UML4CCMDisc (cf. section 4.4.3). En outre, à partir des diagrammes UML de collaborations ou de séquences, l'architecte de l'application déduit les différentes interactions intra-services. Ces interactions sont utilisées dans la construction du graphe de dépendances (cf. section 4.5). Par exemple, dans la figure 4.12, la réalisation du service « réserver un billet » met en interaction les composants `AvailableSeat`, `BookedSeat`, `Location`, `Charter` et `PricesProvider`. Le point d'entrée pour utiliser le service « réserver un billet » est l'opération *bookSeat* de la façade. Cette dernière invoque d'abord un composant `Location` pour éventuellement avoir plus de précision sur les localités (de départ et d'arrivée), puis invoque un composant `AvailableSeat` afin de trouver un billet qui correspond aux critères. Si ce composant `AvailableSeat` trouve le billet correspondant, il met à jour la liste

des billets réservés (invocation du composant `BookedSeat`). Sinon, il invoque un composant `Charter` pour éventuellement trouver un billet charter. Enfin, la façade invoque un composant `PricesProvider` pour récupérer le prix du billet trouvé.

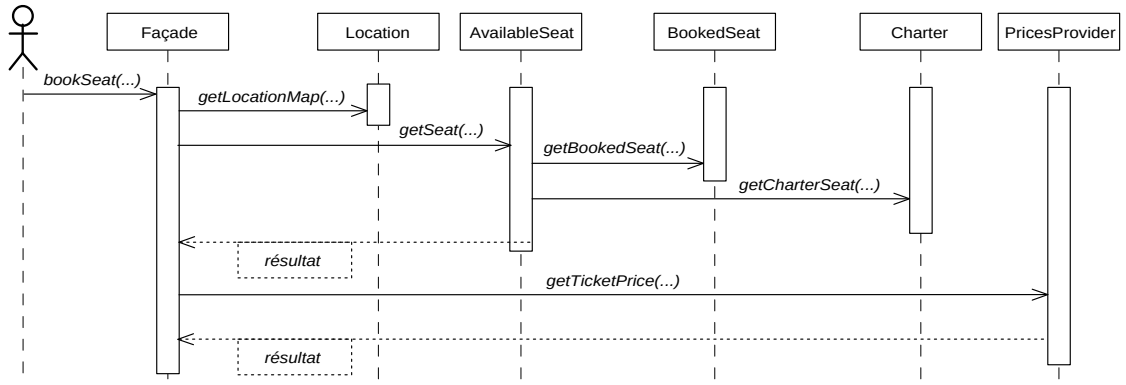


FIG. 4.12 – Diagramme de séquences du service « réserver un billet »

Vue processus

L'approche MADA permet d'exprimer que l'utilisateur doit aussi adapter son comportement en présence de déconnections. Ces adaptations sont spécifiées dans des diagrammes d'activités. Dans notre cas, le diagramme d'activités représente la machine à état[3] de la procédure que l'utilisateur final doit suivre. Les évènements proviennent soit de la plateforme, par exemple le détecteur de connectivité (cf. section 3.1.2), 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. De plus, la vue processus décrit le fonctionnement global de l'ensemble application, intergiciel et cache dans les trois modes de fonctionnement (connecté, partiellement connecté et déconnecté). La figure 4.13 illustre le diagramme d'activités lors du passage au mode partiellement connecté. Cet évènement est capté par le détecteur de connectivité. Il n'est suivi par aucune activité si l'utilisateur était déjà volontairement déconnecté. Dans le cas contraire, l'évènement du passage au mode partiellement connecté est suivi par deux activités concurrentes : la mise à jour de l'état des composants déconnectés dans le cache et la notification à l'utilisateur du changement du mode de connectivité (via l'interface utilisateur). Nous présentons le fonctionnement de l'ensemble application, intergiciel et cache avec plus de détails dans le chapitre 5.

Vue développement

Dans MADA, cette vue consiste d'abord au développement des composants de l'application ainsi que des composants déconnectés associés, et à la construction des paquetages de

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

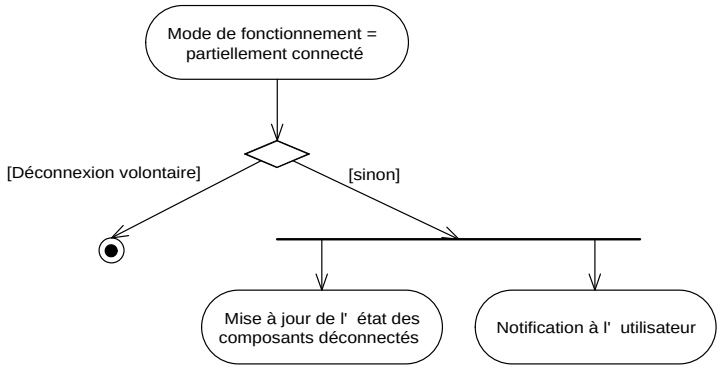


FIG. 4.13 – Diagramme d'activités lors du passage au mode partiellement connecté

chaque composant. Le composant et son composant déconnecté doivent comporter des interfaces conçues pour la capture, le transfert et la mise à jour de l'état du composant. L'interface graphique de l'application doit offrir en plus des interfaces pour la manipulation des services de l'application, un moyen modifier et indiquer les changements de mode de connectivité (cf. figure 4.14). Nous donnons plus de détails dans le chapitre 5.

Vue physique

Par définition, cette vue décrit les différentes ressources matérielles et le déploiement des différents composants dans ces ressources. Dans MADA, cette vue consiste d'abord en la définition du plan de déploiement et d'assemblage de l'application sans prendre en compte les composants déconnectés. Dans CCM, ce plan est décrit dans le fichier CAD (cf. section 2.3.2). Ensuite, les paquetages des composants déconnectés doivent être hébergés dans un serveur d'archivage accessible par les terminaux mobiles. Ces paquetages peuvent être aussi sauvegardés sur les terminaux mobiles si leur mémoire le permet. Enfin, la gestion des instances des composants déconnectés dans le terminal mobile est assurée par les stratégies de déploiement et de remplacement du cache. Nous décrivons ces stratégies dans le chapitre 5.

4.5 Graphe de dépendances

Dans les sections précédentes, nous avons parcouru les différents éléments de MADA qui permettent de construire des applications réparties qui fonctionnent en présence des déconnexions. En plus du profil de l'application, nous avons présenté un profil UML utilisé pour la modélisation d'applications réparties à base de composants CORBA. En outre, ces éléments sont utilisés dans les différentes vues du modèle « 4+1 » vues. Par ailleurs, comme nous l'avons présenté dans le chapitre 3, les stratégies de déploiement et de remplacement du cache doivent prendre en compte les dépendances entre les entités de l'application. En effet, l'utilisation d'une entité déconnectée peut exiger la présence d'autres entités déconnectées dans le cache. Cette problé-

FIG. 4.14 – Interface graphique de l'application *InternetTicket*

matique nous conduit à proposer le graphe de dépendances pour faire apparaître explicitement les dépendances entre les différentes entités de l'application.

Dans cette section, nous décrivons la méthodologie à suivre pour la construction du graphe de dépendances (cf. section 4.5.1). Ensuite, nous présentons la propagation des méta-données dans le graphe de dépendances (cf. section 4.5.2). Enfin, la section 4.5.3 décrit la représentation syntaxique du graphe de dépendances.

4.5.1 Construction du graphe de dépendances

Comme décrit dans la section 4.4.4, un service de l'application est représenté par un scénario ou un cas d'utilisation dans le diagramme UML des cas d'utilisation. Cependant, un cas d'utilisation peut inclure le comportement d'autres cas d'utilisation en utilisant la relation UML `include`. Ainsi, l'utilisation d'un service en mode déconnecté peut exiger la présence d'autres services dans le cache du terminal mobile. La résolution de ce problème exige la détermination des dépendances entre les services. Nous modélisons ces dépendances dans un graphe orienté où les nœuds dénotent les services et les liens dénotent la dépendance entre les services déconnectables. La racine de ce graphe est la façade. Elle représente le composant accessible par l'interface graphique de l'utilisateur et qui donne accès aux services de l'application. Par ailleurs, la disponibilité d'un service en présence des déconnexions implique la présence de certains composants déconnectés dans le cache du terminal mobile. Ainsi, par analogie aux services, les dépendances entre les composants sont aussi modélisées dans le même graphe de dépendances

où les nœuds (en plus des services) dénotent les composants et les liens marqués par la nécessité entre deux composants dénotent les dépendances entre ces composants.

Ainsi, le graphe de dépendances global comporte quatre type d'interactions : entre la façade (la racine du graphe) et les services, entre les services (inter-services), entre les services et les composants (points d'entrées pour utiliser un service) et entre les composants (intra-services). Ces interactions sont marquées par la méta-donnée « nécessité » attribué par l'architecte de l'application. La nécessité d'un lien entre la façade et un service indique la nécessité de la présence de ce service sur le terminal mobile pour le fonctionnement en présence des déconnexions. La nécessité d'un lien entre deux services indique la nécessité du post-service (service en aval) pour l'utilisation du pré-service (service en amont). En outre, la nécessité d'un lien entre un service et un composant (point d'entrée pour l'utilisation du service) indique la nécessité du composant pour l'utilisation du service. Enfin, la nécessité d'un lien entre deux composants indique la nécessité du post-composant (composant en aval) pour l'utilisation du pré-composant (composant en amont) dans le terminal mobile en présence des déconnexions. À la construction du graphe de dépendance, seul les liens entres les différents nœuds sont marqués par la nécessité. La nécessité des différentes nœuds est obtenu en utilisant le mécanisme de propagation de la nécessité que nous décrivons dans la section 4.5.2.

La figure 4.15 présente une partie du graphe de dépendances de l'application *InternetTicket*. D'après ce graphe, le fonctionnement de cette application en présence des déconnexions dans le terminal mobile exige la présence du service « réserver un billet ». De ce fait, la présence des composants déconnectés des composants `AvailableSeat` et `BookedSeat` est exigée pour le fonctionnement en présence des déconnexions. En outre, tous les composants descendant directement ou indirectement d'un service sont reliés à ce dernier (sans tenir compte de la nécessité). Par exemple, dans la figure 4.15, tous les composants à l'intérieur du rectangle descendent directement ou indirectement du service « réserver un billet ». Ils sont donc reliés à ce service.

4.5.2 Propagation des méta-données

Les trois sous-sections suivantes présentent la manipulation de la nécessité et de la priorité, la propagation statique de la nécessité dans le graphe de dépendances, et enfin, la propagation dynamique de la nécessité dans le graphe de dépendances.

Manipulation de la nécessité et de la priorité

Comme indiqué dans la section 4.4.2, l'utilisateur peut changer la nécessité (non nécessaire vers nécessaire utilisateur ou vice versa) et la priorité (augmenter ou diminuer la valeur) d'un service ou d'un composant. En effet, l'application doit offrir une interface graphique sur laquelle l'utilisateur peut manipuler la nécessité et la priorité des services et des composants pendant l'exécution. Dans l'approche MADA, l'utilisateur ne manipule pas directement les composants. Le changement de la nécessité des composants se fait via des options offertes par le service qui manipule le composant. La figure 4.16 présente l'interface graphique pour la manipulation des

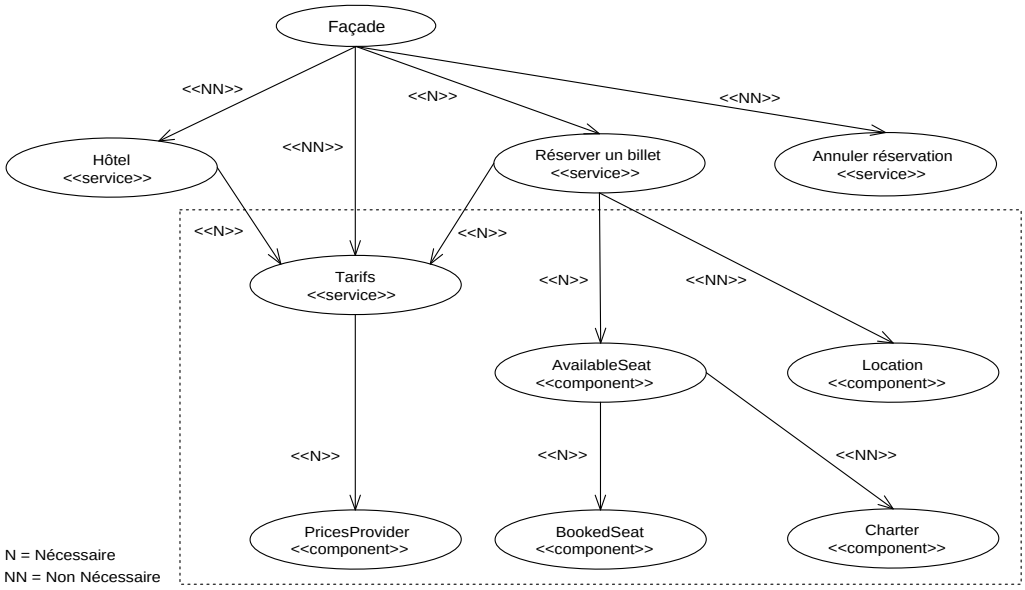


FIG. 4.15 – Graphe de dépendances de l'application *InternetTicket*

méta-données (nécessité et priorité) de l'application *InternetTicket*. Dans cette dernière, l'utilisateur ne peut pas modifier la nécessité du service « réserver un billet » (*Book a ticket* sur la figure) déclaré nécessaire par le développeur de l'application. Par contre, il peut modifier la nécessité des autres services (non nécessaire vers nécessaire utilisateur ou vice versa). Le changement de la nécessité d'un service revient à changer la nécessité entre la façade et le service dans le graphe de dépendances. De plus, l'utilisateur peut changer la nécessité du lien entre le service « réserver un billet » et le composant `Location` en sélectionnant l'option *Location* pour ce service. L'utilisateur peut aussi changer la nécessité du lien entre les composants `AvailableSeat` et `Charter` en sélectionnant l'option *Charter flights* pour le service « réserver un billet » (cf. figure 4.16).

Dans la section 4.4.2, nous avons défini deux types de priorité entre les services : priorité entre les services nécessaires utilisateur et priorité entre les services non nécessaires. Dans notre approche, le changement de la nécessité d'un service affecte la priorité entre les services. L'utilisateur doit donc mettre à jour (suivant ses besoins) la priorité des services. Par ailleurs, le changement de la priorité ne concerne que les services non nécessaires et les services nécessaires utilisateur. Nous considérons que la priorité entre les composants est attribuée par l'architecte de l'application et qu'elle est statique. Cette limitation est due au fait que l'utilisateur ne manipule pas directement les composants. De plus, contrairement à la nécessité, nous considérons que le changement de la priorité d'un service n'affecte en aucun cas la priorité d'un composant ou d'un autre service. Ainsi, nous ne décrivons dans les deux paragraphe suivants que la propagation de la méta-donnée « nécessité » dans le graphe de dépendances.

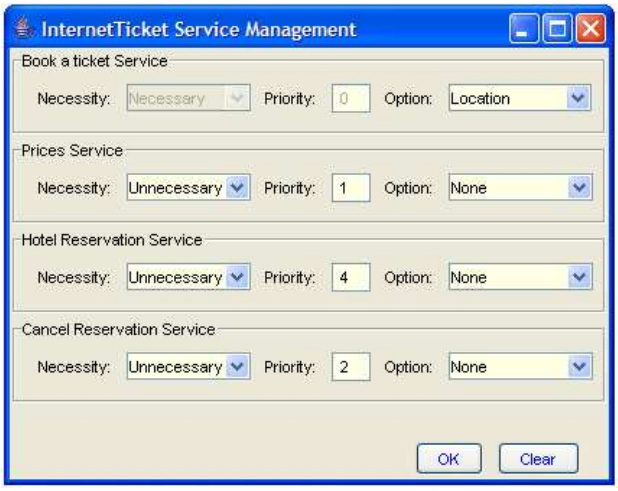


FIG. 4.16 – Interface graphique pour le changement des méta-données

Propagation statique

Le but de la propagation de la nécessité dans le graphe de dépendances est de calculer automatiquement la nécessité des différents nœuds (services et composants). Le principe de base est que pour tout couple de nœuds reliés par un lien, la nécessité du nœud en aval est calculé suivant la nécessité du nœud en amont et la nécessité du lien entre eux. La propagation statique consiste à définir la nécessité par défaut des différents nœuds. Cette propagation est initiée par l'architecte de l'application. Nous décrivons cette propagation à partir de la façade (racine du graphe) vers les composants (feuilles du graphe).

Dans l'approche MADA, la façade représente le point d'entrée pour utiliser les services d'une application. De plus, la façade est encapsulée dans le composant client dans le terminal mobile. Par conséquent, la façade doit être présente dans le terminal mobile. Ainsi, elle est nécessaire. Dans le cas où la nécessité d'un lien entre la façade et un service est nécessaire (développeur ou utilisateur), la nécessité est propagé au service. Dans notre application exemple, le lien entre la façade et le service « réserver un billet » est nécessaire, ce service devient nécessaire. Formellement, soit $\mathcal{S}$ l'ensemble des services de l'application, F la façade, $\mathcal{L}$ l'ensemble des liens (inter- et intra-services), *necLFS(l)* le prédicat évalué a vrai si le lien l entre F et un service s est nécessaire, et *necs(s)* le prédicat évalué à vrai si le service s est nécessaire. Nous modélisons la propagation de la nécessité entre la façade et les services de l'application par la formule suivante :

$$\forall s \in \mathcal{S}, \forall l_{F \to s} \in \mathcal{L} : necLFS(l_{F \to s}) \implies necs(s) \tag{4.1}$$

Dans le cas où le lien entre deux services est nécessaire et le pré-service est nécessaire, la nécessité est propagée au post-service. Dans l'exemple, le service « tarifs » devient aussi nécessaire. Formellement, soit *necLS(l)* le prédicat évalué à vrai si le lien entre deux services est nécessaire. Nous modélisons la propagation de la nécessité inter-services par la formule

suivante :

$$\forall s_1, s_2 \in \mathcal{S}, \forall l_{s_1 \to s_2} \in \mathcal{L} : necLS(l_{s_1 \to s_2}) \wedge necs(s_1) \implies necs(s_2) \tag{4.2}$$

Dans le cas où le lien entre un service est un composant est nécessaire et le service est nécessaire, la nécessité est propagée au composant. Dans exemple, le composant `PricesProvider` devient aussi nécessaire. Formellement, soit $\mathcal{C}$ l'ensemble des composants de l'application, *necLSC(l)* le prédicat évalué à vrai si le lien l entre un service et un composant est nécessaire, et *necc(c,s)* le prédicat évalué à vrai si le composant c est nécessaire pour le service s. Le passage de la nécessité du service aux composants contenus est modélisé par la formule suivante :

$$\forall s \in \mathcal{S}, \forall c \in \mathcal{C}, \forall l_{s \to c} \in \mathcal{L} : necLSC(l_{s \to c}) \ \wedge \ necs(s) \implies necc(c, s) \tag{4.3}$$

Dans le cas où le lien entre deux composants est nécessaire et le pré-composant est nécessaire, la nécessité est propagée au post-composant. Dans exemple, le composant `BookedSeat` devient aussi nécessaire. Formellement, soit *necLC(l)* le prédicat évalué à vrai si le lien l entre deux composants est nécessaire. Cette dernière propagation est modélisée par la formule suivante :

$$\forall s \in \mathcal{S}, \forall c_1, c_2 \in \mathcal{C}, \forall l_{c_1 \to c_2} \in \mathcal{L} : necLC(l_{c_1 \to c_2}) \wedge necc(c_1, s) \implies necc(c_2, s) \tag{4.4}$$

Propagation dynamique

La propagation dynamique consiste à changer la nécessité des services ou des composants de l'application en cours d'exécution. En effet, comme décrit dans le début de cette section, l'utilisateur peut changer la nécessité (non nécessaire vers nécessaire utilisateur ou vice versa) des services et des composants. Cependant, l'utilisateur ne change pas directement la nécessité de ces entités mais à travers les liens entre elles. Les formules 4.1 à 4.4 sont aussi appliquées dans la propagation dynamique lorsque la nécessité d'une entité ou d'un lien passe de non nécessaire vers nécessaire. Il ne reste donc que la définition des formules de propagation à utiliser lorsque la nécessité d'une entité ou d'un lien passe de nécessaire utilisateur vers non nécessaire.

Dans le cas où l'utilisateur change la nécessité d'un service s_1 (changement de la nécessité du lien entre la façade et s_1) de nécessaire utilisateur vers non nécessaire, et s'il n'existe pas d'autre service nécessaire s_2 qui utilise s_1 (le lien $l_{s_2 \to s_1}$ etant nécessaire), s_1 devient non nécessaire. Par exemple, si l'utilisateur remet la nécessité du lien entre la façade et le service « hôtel » à son état initial (non nécessaire), ce service n'est utilisé par aucun autre service nécessaire, alors il devient non nécessaire. Nous modélisons cette description par la formule suivante :

$$\forall s_1 \in \mathcal{S} : \neg necLFS(l_{F \to s_1}) \wedge (\nexists s_2 \in \mathcal{S} \ : \ necs(s_2) \ \wedge \ necLS(l_{s_2 \longrightarrow s_1})) \implies \neg necs(s_1) \tag{4.5}$$

Considérons maintenant le cas où le service s_1, précédemment nécessaire, devient non nécessaire (une conséquence de l'équation 4.5). Si le lien entre s_1 et un autre service s_2 est nécessaire et s'il n'existe pas d'autre service s_3 tel que s_3 et $l_{s_3 \to s_2}$ sont nécessaires, alors s_2 devient non nécessaire. Nous modélisons cette dernière description par la formule suivante :

$$\forall s_1, s_2 \in \mathcal{S} : \neg necs(s_1) \ \wedge \ necLS(l_{s_1 \to s_2}) \ \wedge \ (\nexists s_3 \in \mathcal{S} \ : \ necs(s_3) \ \wedge \ necLS(l_{s_3 \longrightarrow s_2})) \implies \neg necs(s_2) \tag{4.6}$$

Dans le cas où il existe un lien entre un service s_1 et un composant c_1 où s_1 précédemment nécessaire devient non nécessaire, s'il n'existe pas un autre service s_2 tel que s_2 et $l_{s_2 \to c_1}$ sont nécessaires et s'il n'existe pas un autre composant c_2 tel que c_2 et $l_{c_2 \to c_1}$ sont nécessaires, alors c_1 devient non nécessaire. L'exemple typique pour cette dernière propagation est la perte de la nécessité pour le composant `Location` lorsque l'option *Location* du service « réserver un billet » est désactivée. Cette dernière description est modélisée par la formule suivante :

$$\forall s_1 \in \mathcal{S}\ , \forall c_1 \in \mathcal{C}\ :\ \neg necs(s_1)\ \wedge\ necLSC(l_{s_1 \to c_1})\ \wedge\ (\nexists s_2 \in \mathcal{S}\ :\ necs(s_2)\ \wedge\ necLSC(l_{s_2 \longrightarrow c_1})) \\ \wedge\ (\nexists c_2 \in (C)\ :\ necc(c_2)\ \wedge\ necLC(l_{s_2 \longrightarrow c_2}))\ \implies\ \neg necc(c_1) \tag{4.7}$$

Dans le cas où il existe un lien entre un composant c_1 et un composant c_2 où c_1 précédemment nécessaire devient non nécessaire, s'il n'existe pas un service s tel que s et $l_{s \to c_2}$ sont nécessaires et s'il n'existe pas un autre composant c_3 tel que c_3 et $l_{c_3 \to c_1}$ sont nécessaires, alors c_2 devient non nécessaire. Cette dernière description est modélisée par la formule suivante :

$$\forall c_1, c_2 \in \mathcal{C}\ :\ \neg necc(c_1)\ \wedge\ necLSC(l_{c_1 \to c_2})\ \wedge\ (\nexists s \in \mathcal{S}\ :\ necs(s)\ \wedge\ necLSC(l_{s \longrightarrow c_2})) \\ \wedge\ (\nexists c_3 \in \mathcal{C}\ :\ necc(c_3)\ \wedge\ necLC(l_{c_3 \longrightarrow c_2}))\ \implies\ \neg necc(c_2) \tag{4.8}$$

Enfin, un composant de l'application est nécessaire s'il est nécessaire pour au moins un service. Cette contrainte est modélisée ainsi :

$$\forall c \in \mathcal{C}, \forall s \in \mathcal{S}\ : necc(c,s) \wedge necs(s)\ \implies\ necc(c) \tag{4.9}$$

4.5.3 Représentation du graphe de dépendances

La figure 4.17 présente la structure générale du graphe de dépendances. Nous avons utilisé GXL (*Graph eXchange Language*) [60], une notation XML, pour la description du graphe de dépendances. L'utilisation de GXL est motivée par la présence d'un mécanisme flexible et extensible pour décrire des graphes multi-ressources (différents types de nœuds) et pour décrire des graphes hiérarchiques (un nœud peut contenir un sous-graphe). Enfin, GXL dispose d'une implantation en source libre que nous avons adaptée à nos besoins (cf. chapitre 5).

Un graphe de dépendances est composé de quatre groupes d'éléments : nœuds services, nœuds composants, liens entre la façade et les services, et liens inter-services. Chaque nœud dispose, en plus des méta-données *nécessité* et *priorité*, la méta-donnée *entité* qui donne le type du nœud (service ou composant). Chaque nœud service dispose aussi d'un sous-graphe de dépendances. Ce dernier décrit les liens d'interactions intra-services (au moins un lien par service). Les liens sont décrits avec la méta-donnée *nécessité*. Chaque nœud composant dispose, en plus des méta-données *entité*, *nécessité* et *priorité*, de deux autres méta-données : la méta-donnée *nom du composant déconnecté* et la méta-donnée *archive du composant déconnecté* qui donne le chemin de l'implantation du composant déconnecté.

La figure 4.18 présente une partie du graphe de dépendances de l'application *InternetTicket* en GXL. La balise XML *graph* (lignes 4 à 60) définit le graphe de dépendances de l'application référencée par l'élément *id*. Le service « Réserver un billet » est décrit dans la balise XML

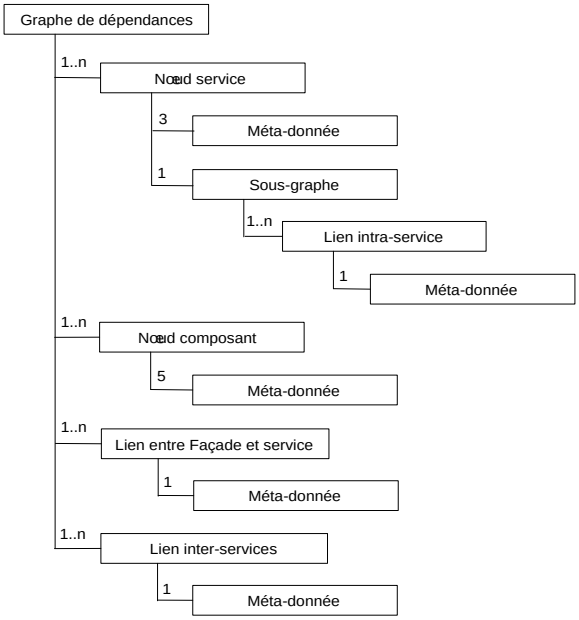


FIG. 4.17 – Structure du graphe de dépendances

node (lignes 5 à 29). Les méta-données du service sont présentées par les balises XML *attr*. Par exemple, la balise *attr* de la ligne 6 à la ligne 8 décrit la nécessité du service et son type (développeur ou utilisateur). Le sous-graphe de dépendances du service « réserver un billet » est décrit dans la balise XML *graph* (lignes 15 à 27). Elle comporte des sous-balises XML *edge* qui décrivent les interactions intra-services. Par exemple, le lien d'interaction entre les composants `AvailableSeat` et `BookedSeat` est décrit dans la balise *edge* de la ligne 16 à la ligne 20. De la même manière que les services, les composants de l'application sont décrits dans des balises XML *node*. Cependant, la balise XML *node* décrivant un composant ne comporte pas de sous-balise XML *graph*. Par exemple, la balise *node* de la ligne 31 à la ligne 47 décrit le composant `AvailableSeat`. Enfin, les interactions entre la façade et les services, et les interactions inter-services sont décrites dans des balises XML *edge*. Par exemple, la balise XML *edge* de la ligne 54 à la ligne 58 définit le lien d'interaction entre les services « réserver un billet » et « tarifs ».

4.6 Discussion

Par rapport aux travaux existants sur la gestion des déconnexions (cf. chapitre 3), l'originalité de notre approche réside dans le fait que nous considérons le problème dès la conception de l'application. L'application est donc construite pour être fonctionnelle en présence des déconnexions.

La plupart des solutions étudiées dans le chapitre 3 propose un profil de l'application pour la gestion du cache défini par l'utilisateur. Dans notre approche, le profil de l'application est créé par l'architecte de l'application et peut être personnalisé par l'utilisateur. Nous motivons ce choix par le fait que l'architecte a la meilleure connaissance de la sémantique de l'application. Ainsi, nous

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM ".../gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
   <graph id="internetTicket" edgeids="true" ...">
      <node id="Book">
         <attr name="isNecessary" kind="developer">
            <bool>true</bool>
         </attr>
         <attr name="priority">
            <int>1</int>
         </attr>
         <attr name="entity">
            <string>Service</string>
         </attr>
         <graph id="Book_Service">
            <edge id="r1_Book" from="AvailableSeat" to="BookedSeat">
               <attr name="isNecessary">
                  <bool>true</bool>
               </attr>
            </edge>
...
            <edge id="r5_Book" from="Book" to="Location">
               <attr name="isNecessary">
                  <bool>false</bool>
               </attr>
            </edge>
         </graph>
...
      </node>
...
      <node id="AvailableSeat">
         <attr name="isNecessary" kind="developer">
            <bool>true</bool>
         </attr>
         <attr name="priority">
            <int>1</int>
         </attr>
         <attr name="entity">
            <string>Component</string>
         </attr>
         <attr name="disComponentName">
            <string>DiscAvailableSeat</string>
         </attr
         <attr name="disComponentArchive">
            <string>../disc_availableSeat.car</string>
         </attr>
      </node>
...
      <edge id="r1" from="Facade" to="Book">
         <attr name="isNecessary" kind="developer">
            <bool>true</bool>
         </attr>
      </edge>
      <edge id="r2" from="Book" to="Prices">
         <attr name="isNecessary">
            <bool>false</bool>
         </attr>
      </edge>
...
   </graph>
</gxl>
```

FIG. 4.18 – Exemple de graphe de dépendances en GXL

avons adopté une stratégie d'adaptation « en collaboration » (cf. section 3.1) pour la construction du profil de l'application. Cette adaptation est effectuée en mixant l'adaptation statique (construction du profil de l'application par l'architecte pendant le développement) et l'adaptation dynamique (personnalisation du profil de l'application par l'utilisateur à l'exécution).

Par ailleurs, nous avons vu dans le chapitre 3 l'importance de la prise en compte des interactions entre les entités de l'application dans la gestion du cache du terminal mobile. Cependant, comme présenté dans le tableau 3.2, hormis le projet Seer (dépendances intra-projets) et le projet ACHILLES (dépendances entre les composants de l'application et entre les ressources du terminal), les dépendances entre les entités de l'application ne sont pas prises en compte dans la gestion du cache. Dans l'approche MADA, nous avons proposé d'utiliser un graphe de dépendances pour faire apparaître explicitement les dépendances entre les différentes entités de l'application.

Comme décrit dans le chapitre 2, nous nous basons sur le paradigme composant/conteneur pour la séparation des préoccupations fonctionnelles et la gestion des déconnexions. Dans ce chapitre, nous n'avons pas abordé ce point qui concerne plus l'implantation des composants et les différentes interactions entre l'application, l'intergiciel et le cache. Nous présentons ce point dans le chapitre suivant.

4.7 Synthèse

Dans ce chapitre, nous avons présenté MADA, une approche de conception d'applications pouvant fonctionner en présence des déconnexions. Le but de l'approche MADA n'est pas de proposer une nouvelle méthode de conception d'applications réparties, mais plutôt de prendre en considération le problème de la gestion du cache pour les déconnexions dès la conception de l'application. De ce fait, nous avons adopté une approche orientée MDA centrée sur l'architecture logicielle et pilotée par des cas d'utilisation de l'application. Les concepts que nous avons proposés dans MADA sont suffisamment généraux pour être utilisés avec d'autres processus de développement, dans la mesure où ils séparent les préoccupations fonctionnelles et extra-fonctionnelles, et utilisent un processus centré sur l'architecture logicielle et piloté par des cas d'utilisation.

L'apport de l'approche MADA peut se résumer en trois éléments : un profil de l'application, un profil UML pour la gestion du cache et un graphe de dépendances. Le profil de l'application sert à décrire les entités de l'application pour la gestion du cache. Il se base sur trois méta-données : *déconnectabilité*, *nécessité* et *priorité*. La *déconnectabilité* indique si une entité de l'application peut avoir une entité déconnectée sur le terminal mobile. La *nécessité* indique si la présence de l'entité déconnectée dans le cache du terminal mobile est nécessaire pour le fonctionnement de l'application en présence des déconnexions. La *priorité* indique la priorité entre les entités de l'application quand à leur présence dans le cache du terminal mobile. Ces méta-données sont utilisées pour décrire les composants et les services de l'application. Le service est une composition logique de composants qui interagissent pour réaliser une fonctionnalité de l'application. Le profil UML pour la gestion des déconnexions nommé UML4CCMDisc permet d'étendre le processus

de développement d'une application CCM pour prendre en compte la gestion des déconnexions. Enfin, le graphe de dépendances fait apparaître explicitement les interactions intra-services (entre les composants d'un même service) et inter-services (entre les services de l'application). Nous avons proposé un ensemble de règles de propagation de la méta-donnée *nécessité* afin de rendre sa manipulation dynamique pendant l'exécution.

Le chapitre suivant décrit le fonctionnement et l'implantation de notre service de gestion du cache que nous proposons dans le cadre du canevas logiciel DOMINT. Nous nous basons sur les différents concepts proposés dans l'approche MADA.

Chapitre 5

Gestionnaire du cache de DOMINT

Nous avons présenté dans le chapitre précédent MADA, une approche de conception d'applications pour le fonctionnement en présence des déconnexions. Comme décrit dans le chapitre 3, la gestion des déconnexions ne se limite pas à une approche de conception. Dans cette thèse, en plus de l'approche de conception, les préoccupations qui nous intéressent sont la gestion du cache du terminal mobile et la commutation entre les entités distantes et les entités déconnectées de l'application. Pour le premier point, nous proposons un service de gestion du cache, et pour le deuxième point, nous proposons d'intégrer le mécanisme de commutation dans les conteneurs des composants.

Ce chapitre décrit l'architecture et l'implantation du service de gestion du cache et du conteneur de composants dans le cadre du canevas logiciel DOMINT[1]. Nous donnons une introduction à ce chapitre dans la section 5.1 avant de présenter dans la section 5.2 le canevas logiciel DOMINT. Nous nous intéressons ensuite au service de gestion du cache (cf. section 5.3) et au conteneur de composants (cf. section 5.4). Enfin, nous donnons une discussion (cf. section 5.5) et une synthèse (cf. section 5.6) sur le travail effectué.

5.1 Introduction

Le canevas logiciel DOMINT résulte d'un travail de recherche dans l'équipe MARGE[2] à l'Institut National des Télécommunications dans le cadre du projet *ITEA Eureka VIVIAN* [169]. Les mécanismes principaux proposés par DOMINT sont, d'une part, le support de la réplication des objets CORBA, et d'autre part, le support des déconnexions volontaires et involontaires en utilisant des mécanismes CORBA tels que les objets par valeur et les intercepteurs portables [35].

L'architecture de DOMINT que nous présentons dans la section 5.2 est le fruit de notre participation dans les projets européens *ITEA Eureka OSMOSE* [128] et *RNTL franco-finlandais AMPROS* [1]. Notre contribution dans DOMINT s'axe essentiellement sur trois parties : l'application

[1] DOMINT est l'acronyme de *Disconnected Operation for Mobile Internetworking Terminals*.
[2] MARGE est l'acronyme de Middleware pour Applications Réparties avec Gestion de l'Environnement.

de l'approche MADA dans la conception des applications pour le fonctionnement en présence des déconnexions, la conception et la réalisation d'un service de gestion du cache pour composants CORBA, et enfin, l'intégration d'un service de gestion des déconnexions dans les conteneurs des composants. Les différents prototypes réalisés s'inscrivent dans la mise en œuvre CCM/Java de DOMINT basée sur la plateforme OpenCCM, le modèle de conteneurs extensibles (ECM) et le modèle de composants Fractal, tous hébergées par le consortium ObjectWeb [119]. Les prototypes que nous présentons dans ce chapitre sont accessibles sur le site de DOMINT [39].

5.2 Canevas logiciel DOMINT

L'architecture de DOMINT est découpée en trois couches : composant, conteneur et intergiciel. La figure 5.1 présente une vue d'ensemble de ces trois couches. La couche composant constitue la partie métier de l'application. Le composant serveur (*Server Component* sur la figure) représente la partie serveur de l'application (sur un terminal fixe ou mobile). Il offre des interfaces utilisées par le composant client (*Client Component* sur la figure). Ce dernier représente la partie client de l'application (sur un terminal mobile). De plus, il encapsule les interfaces de manipulation du profil de l'application et du graphe de dépendances proposés dans MADA. Par ailleurs, comme décrit dans la section 2.3.2, un composant peut offrir des interfaces et utiliser d'autres interfaces offertes pas d'autres composants. De ce fait, le composant peut être un client pour un ensemble de composants et un serveur pour d'autres. Le composant serveur sur la figure 5.1 peut être donc un composant client pour un autre composant serveur.

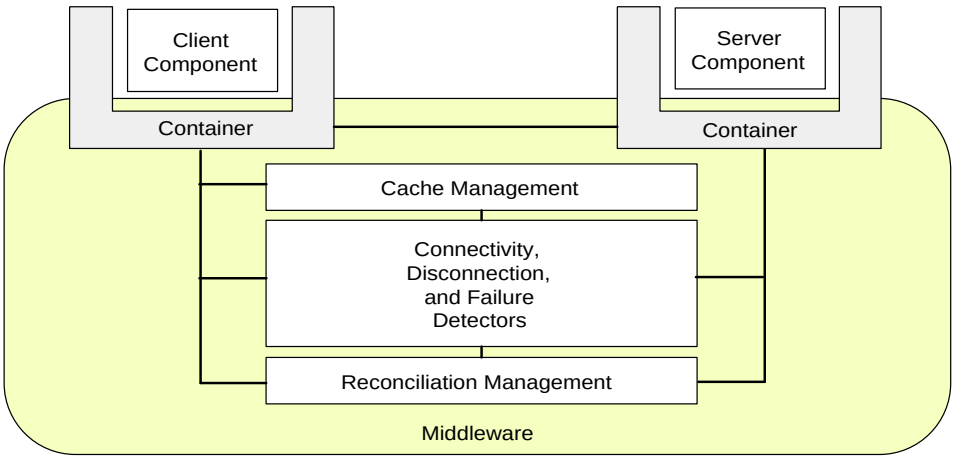


FIG. 5.1 – Architecture générale de DOMINT

La couche conteneur constitue l'environnement d'exécution des composants CORBA. Comme nous l'avons vu dans la section 2.3.2, l'environnement d'exécution d'un composant CORBA est contrôlé par le conteneur (*Container* sur la figure). Ce dernier héberge les instances des composants et fournit des interfaces pour l'activation et la désactivation des serviteurs CORBA, l'accès à l'ORB sous-jacent, la manipulation des POA (*Portable Object Adapter*), et l'accès aux services CORBA tels que la persistance, la sécurité, le nommage, et dans le cas de DOMINT la gestion

des déconnexions. Dans DOMINT, le service de gestion des déconnexions est contrôlée par le conteneur. Il coordonne la commutation transparente entre le composant serveur et le composant déconnecté dans le cache du terminal mobile. Il est aussi responsable de l'orchestration des différents services de la couche intergiciel de DOMINT.

La couche intergiciel offre trois services : le gestionnaire du cache (*Cache Management* sur la figure 5.1), le gestionnaire de réconciliation (*Reconciliation Management* sur la figure 5.1), et les détecteurs de connectivité, de déconnexions et de défaillances (*Connectivity, Disconnection, and Failure Detectors* sur la figure 5.1). Le gestionnaire du cache centralise la gestion de tous les composants déconnectés dans le cache du terminal mobile. Il fournit la mise en œuvre des stratégies de déploiement et de remplacement du cache. 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. Cela permet un gain de place et une gestion de la cohérence facilitée. Le gestionnaire de réconciliation maintient la cohérence entre le composant serveur et son composant déconnecté. Il fournit des mécanismes pour la sauvegarde et la gestion des opérations exécutées sur les composants déconnectés. De plus, il fournit la mise en œuvre de l'algorithme de réconciliation. Enfin, les détecteurs de connectivité, de déconnexions et de défaillances sont des entités dédiées à la gestion du contexte. Le détecteur de connectivité surveille le niveau de disponibilité des ressources locales pour les communications et exprime la connectivité selon les modes connecté, partiellement connecté et déconnecté. Le détecteur de déconnexions exécute un algorithme réparti afin de récolter des informations sur l'état (connecté ou pas) des autres terminaux. Le détecteur de défaillances est l'entité responsable de détecter les hôtes défaillants dans le réseau. Il se base sur l'algorithme dit à « battements de cœur ».

La contribution de cette thèse dans DOMINT consiste en l'intégration du service de gestion des déconnexions dans les conteneurs des composants et le gestionnaire du cache. Pour les autres mécanismes, le lecteur peut se référer à [30] pour le gestionnaire de réconciliation et à [158] pour les trois détecteurs.

5.3 Service de gestion du cache

Comme nous l'avons vu dans la section 3.2, le gestionnaire du cache doit offrir des stratégies de déploiement et de remplacement. La stratégie de déploiement détermine les entités déconnectées à créer dans le cache du terminal mobile, quand et pour quelle durée, tandis que la stratégie de remplacement décide quelles entités déconnectées doivent être supprimées lorsqu'il n'existe plus assez d'espace mémoire dans le cache. Dans cette section, nous présentons les stratégies de déploiement (cf. section 5.3.1) et de remplacement (cf. section 5.3.2) que nous proposons dans DOMINT. Ensuite, la section 5.3.3 présente l'architecture du gestionnaire du cache de DOMINT. Enfin, nous présentons dans la section 5.3.4 les caractéristiques des prototypes du gestionnaire du cache réalisés.

5.3.1 Stratégie de déploiement

Afin de garantir le fonctionnement des applications en présence des déconnexions, nous proposons une stratégie de déploiement qui se décline en trois étapes complémentaires : au pré-chargement, à la demande et à l'invocation. Les trois paragraphes suivants décrivent ces étapes de déploiement. Le dernier paragraphe est consacré à la description des interactions entre les étapes de déploiement.

Au pré-chargement

L'algorithme 1 présente le déroulement du déploiement au pré-chargement. Il prend comme paramètre d'entrée le graphe de dépendances introduit dans MADA. La structure de donnée `Graph` représente le graphe de dépendances. Elle décrit tous les services et tous les composants d'une application. La méta-donnée « nécessité » de ces services et de ces composants correspond à la « nécessité » obtenue après la propagation de cette dernière (cf. formules 4.1 à 4.9 de la section 4.5.2).

Au lancement d'une application sur le terminal mobile, le gestionnaire du cache déploie tous les services nécessaires développeur de cette application. Dans la suite de ce document, le déploiement d'un service correspond au déploiement de tous les composants nécessaires développeur de ce dernier. Comme décrit dans l'approche MADA, l'application ne peut pas fonctionner en présence des déconnexions sans la présence de tous les services nécessaires développeur dans le cache du terminal mobile. De ce fait, le processus de déploiement des services nécessaires développeur (lignes 1 à 12) est bloquant et l'application ne démarre qu'une fois ce processus terminé. Ce processus est interrompu si le déploiement d'un composant nécessaire développeur échoue (ligne 10) suite à un manque d'espace mémoire dans le cache. Dans ce dernier cas, l'utilisateur peut intervenir soit en augmentant la taille du cache, soit en supprimant des services d'autres applications déjà déployés dans le cache (cf. section 5.3.2).

Une fois tous les services nécessaires développeur de l'application déployés, le gestionnaire du cache libère l'application et lance un nouveau processus (lignes 13 à 37) s'exécutant en arrière plan (en anglais, *background execution*) que nous appelons *optProcess*. L'objectif de ce processus est, d'une part, d'exploiter au maximum la taille du cache, et d'autre part de faire face à des éventuelles déconnexions inopinées. Nous comparons les performances du déploiement avec et sans le processus *optProcess* dans le chapitre 6. Le processus *optProcess* se résume en trois étapes séquentielles suivant l'importance des entités à déployer. La première étape consiste au déploiement de tous les services nécessaires utilisateur de l'application (lignes 13 à 22). L'ordre de déploiement de ces services est établi suivant la méta-donnée « priorité » (ligne 13). La deuxième étape consiste au déploiement de tous les composants nécessaires utilisateur des services nécessaires développeur de l'application (lignes 23 à 29). L'ordre de déploiement de ces composants est établi suivant la méta-donnée « priorité » (ligne 25). Enfin, la dernière étape consiste au déploiement de tous les composants nécessaires utilisateur des services nécessaires utilisateur de l'application (lignes 30 à 37). L'ordre de déploiement de ces composants est établi, d'abord suivant la méta-donnée « priorité » entre les services nécessaires utilisateur (ligne 13), ensuite suivant la méta-donnée « priorité » entre les composant d'un service (ligne 33).

Contrairement au déploiement des services nécessaires développeur, lorsqu'un composant ne peut pas être déployé dans le processus *optProcess* (taille du cache restante insuffisante), le gestionnaire du cache ne renvoie aucune erreur et arrête le processus *optProcess* (lignes 20, 28 et 36). Cet arrêt est transparent à l'utilisateur. Les services et les composants non déployés dans le déploiement au pré-chargement peuvent être déployés dans les déploiements à la demande ou à l'invocation que nous décrivons dans les paragraphes suivants.

Algorithme 1: preChargementDeployment(`Graph` *graph*)

```
1  ServiceSet services1 ← getNecessaryDeveloperServices(graph)
2  ServiceSet services2 ← getNecessaryUserServices(graph)
3  ComponentSet components
4  Service s
5  Component cmp
6  for all s ∈ services1
7    components ← getNecessaryDeveloperComponents(s)
8    for all cmp ∈ components
9      if isAlreadyDeployed(cmp) then continue
10     if (getFreeCacheSize() < cmp.getSize()) then return Exception
11     deployComponent(cmp)
12   s.isDeployed ← true
13 orderServicesByPriority(services2)
14 for all s ∈ services2
15   components ← getNecessaryDeveloperComponents(s)
16   orderComponentsByPriority(components)
17   while (¬s.allNecessaryDeveloperComponentsDeployed())
18     for all cmp ∈ components
19       if isAlreadyDeployed(cmp) then continue
20       if (getFreeCacheSize() < cmp.getSize()) then return
21       deployComponent(cmp)
22   s.isDeployed← true
23 for all s ∈ services1
24   components ← getNecessaryUserComponents(s)
25   orderComponentsByPriority(components)
26   for all cmp ∈ components
27     if isAlreadyDeployed(cmp) then continue
28     if (getFreeCacheSize() < cmp.getSize()) then return
29     deployComponent(cmp)
30 orderServicesByPriority(services2)
31 for all s ∈ services2
32   components ← getNecessaryUserComponents(s)
33   orderComponentsByPriority(components)
34   for all cmp ∈ components
35     if isAlreadyDeployed(cmp) then continue
36     if (getFreeCacheSize() < cmp.getSize()) then return
37     deployComponent(cmp)
```

À la demande

Comme décrit dans l'approche MADA, l'utilisateur peut personnaliser le profil de l'application en changeant la nécessité et la priorité des services non nécessaires, des services nécessaires utilisateur, et des composants nécessaires utilisateur et non nécessaires. Le changement de la nécessité et de la priorité des composants est effectué en changeant les options des services (cf. section 4.5.2). Ces changements peuvent être suivis par la propagation de la « nécessité » dans le graphe de dépendances. Ainsi, suite à un changement dans le graphe de dépendances, le contenu du cache (services et composants) peut être en contradiction avec le nouveau graphe de dépendances. Par exemple, supposons deux services *serviceA* et *serviceB* dont le premier est nécessaire utilisateur et le deuxième est non nécessaire. Avant que l'utilisateur ne change la nécessité de *serviceB* et sa priorité (*serviceB* devient nécessaire utilisateur et plus prioritaire que *serviceA*), le cache ne contenait que *serviceA*. Cependant, après ces changements, le contenu du cache est en contradiction avec le graphe de dépendances qui exige la présence (si c'est possible) de *serviceB* puis de *serviceA*. Ainsi, nous proposons le déploiement à la demande, initié à chaque changement de la nécessité dans le graphe de dépendances.

L'algorithme 2 présente le déroulement du déploiement à la demande. Le processus initiant le déploiement à la demande est exécuté en arrière plan comme le processus *optProcess* au pré-chargement. Nous distinguons trois étapes séquentielles dans ce processus. La première étape consiste au déploiement de tous les composants nécessaires utilisateur (non encore déployés) des services nécessaires développeur de l'application (lignes 6 à 13). L'ordre de déploiement de ces composants est établi suivant la priorité entre ces composants (ligne 8). La deuxième étape consiste au déploiement de tous les services nécessaires utilisateur (non encore déployés) de l'application (lignes 14 à 25). L'ordre de déploiement de ces services est établi suivant la priorité de ces services (ligne 14). Enfin, la dernière étape consiste au déploiement de tous les composants nécessaires utilisateur (non encore déployés) des services nécessaires utilisateur de l'application (lignes 26 à 33). L'ordre de déploiement de ces composants est établi, d'abord suivant la priorité des services nécessaires utilisateur (ligne 14), ensuite suivant la priorité des composants nécessaires utilisateur de chaque service (ligne 28).

Contrairement au déploiement des composants dans le processus *optProcess* qui se termine lorsque la taille du cache est insuffisante, le processus de déploiement à la demande fait appel à la fonctionnalité *expliciteReplacement* (lignes 12, 23 et 32). Cette fonctionnalité est utilisée afin de libérer une partie (passée en paramètre) de la mémoire du cache (renvoie *true* si la libération de l'espace mémoire est effectuée avec succès). Nous donnons l'algorithme de *expliciteReplacement* dans la section 5.3.2. Nous justifions ce choix par le fait que l'utilisateur qui a provoqué le déploiement à la demande se prépare à une déconnexion volontaire, et ainsi la présence des nouveaux services et composants nécessaires utilisateur en mode déconnecté augmente la qualité de service de l'application.

Comme nous le décrivons dans la section 5.3.2, la fonctionnalité *expliciteReplacement* peut rester bloquée indéfiniment. La solution que nous proposons consiste à interrompre l'exécution de *expliciteReplacement* après l'expiration d'un compteur (*timer*) dont la valeur est configurable par l'utilisateur. Dans ce cas, la valeur de retour de cette fonctionnalité est *false*, ce qui conduit à l'arrêt de l'exécution du déploiement à la demande. Cet arrêt est présenté à l'utilisateur via un

message d'information. L'utilisateur peut entreprendre des actions telles que l'augmentation de la taille mémoire du cache avant de relancer le déploiement à la demande. Le déploiement des nouveaux services et composants nécessaires utilisateur non déployés peut être initié soit dans un futur déploiement à la demande, soit dans le déploiement à l'invocation que nous présentons dans le paragraphe suivant.

Algorithme 2: onDemandDeployment(`Graph` *graph*)

```
1  ServiceSet services1 ← getNecessaryDeveloperServices(graph)
2  ServiceSet services2 ← getNecessaryUserServices(graph)
3  ComponentSet components
4  Service s
5  Component cmp
6  for all s ∈ services1
7    components ← getNecessaryUserComponents(s)
8    orderComponentsByPriority(components)
9    for all cmp ∈ components
10     if isAlreadyDeployed(cmp) then continue
11     if (getFreeCacheSize() < cmp.getSize())
12       if (¬ explicitReplacement(cmp.getSize()) then return
13     deployComponent(cmp)
14 orderServicesByPriority(services2)
15 for all s ∈ services2
16   if isAlreadyDeployed(s) then continue
17   components ← getNecessaryDeveloperComponents(s)
18   orderComponentsByPriority(components)
19   while (¬s.allNecessaryDeveloperComponentsDeployed())
20     for all cmp ∈ components
21       if isAlreadyDeployed(cmp) then continue
22       if (getFreeCacheSize() < cmp.getSize())
23         if (¬ explicitReplacement(cmp.getSize()) then return
24       deployComponent(cmp)
25     s.isDeployed ← true
26 for all s ∈ services2
27   components ← getNecessaryUserComponents(s)
28   orderComponentsByPriority(components)
29   for all cmp ∈ components
30     if isAlreadyDeployed(cmp) then continue
31     if (getFreeCacheSize() < cmp.getSize())
32       if (¬ explicitReplacement(cmp.getSize()) then return
33     deployComponent(cmp)
```

À l'invocation

En se basant juste sur les deux premières étapes de déploiement, les services non nécessaires et les composants non nécessaires ne sont jamais déployés dans le cache sauf si l'utilisateur change leur nécessité (déploiement à la demande). Bien que la présence de ces entités dans le cache du terminal mobile ne soit pas obligatoire pour le fonctionnement de l'application en présence des déconnexions, leur présence peut améliorer la qualité de service de l'applica-

tion. Ainsi, ces entités peuvent être déployées dans le cache tant que l'espace mémoire de ce dernier le permet. Par exemple, dans l'application *InternetTicket* présentée dans le chapitre 4, le service « annuler réservation » (respectivement le composant `Charter`) peut ne pas exister dans le cache si l'utilisateur ne change pas la nécessité de ce service (respectivement ne sélectionne pas l'option *Charter flights* du service « réserver un billet »). La présence du service « annuler réservation » dans le cache peut être utile à un utilisateur qui veut annuler une réservation en mode déconnecté suite à une erreur de saisie.

Dans notre approche, le déploiement des services et des composants non nécessaires se fait à leur invocation, ce qui permet de déployer ces entités au fur et à mesure de leur invocation. Bien que l'invocation d'un composant corresponde à l'utilisation d'une interface offerte par ce dernier, la détection de l'invocation d'un service est effectuée au niveau de la façade de l'application. Par exemple, dans l'application *InternetTicket*, l'utilisation du service « réserver un billet » correspond au cliquement du bouton *Search* de l'interface graphique (cf. figure 4.14).

L'algorithme 3 (respectivement l'algorithme 4) présente le déroulement du déploiement à l'invocation d'un service (respectivement d'un composant) non nécessaire. Comme dans le déploiement à la demande, le processus de déploiement d'un composant dans le déploiement à l'invocation fait appel à la fonctionnalité *expliciteReplacement* lorsque la taille du cache restante est insuffisante pour déployer un composant (ligne 8 de l'algorithme 3 et ligne 2 de l'algorithme 4). Comme nous le verrons dans la section 5.3.2, *expliciteReplacement* ne peut pas supprimer un service ou un composant nécessaire (développeur ou utilisateur) pour déployer un service ou un composant non nécessaire.

Contrairement au déploiement à la demande, l'arrêt de l'exécution du déploiement à l'invocation causé par la fonctionnalité *expliciteReplacement* ne renvoie pas d'erreur (transparence à l'utilisateur). En effet, comme décrit dans MADA, la présence des services et composants non nécessaires dans le cache du terminal mobile n'est pas obligatoire pour le fonctionnement de l'application en présence des déconnexions.

Algorithme 3: atInvocationDeployment(`Service` *s*)

```
1  ComponentSet components ← getNecessaryDeveloperComponents(s)
2  Component cmp
3  orderComponentsByPriority(components)
4  while (¬s.allNecessaryDeveloperComponentsDeployed())
5    for all cmp ∈ components
6      if isAlreadyDeployed(cmp) then continue
7      if (getFreeCacheSize() < cmp.getSize())
8        if (¬ explicitReplacement(cmp.getSize()) then return
9      deployComponent(cmp)
10   s.isDeployed← true
```

Algorithme 4: atInvocationDeployment(`Component` *cmp*)

```
1  if (getFreeCacheSize() < cmp.getSize())
2    if (¬ explicitReplacement(cmp.getSize()) then return
3  deployComponent(cmp)
```

Interactions entre les étapes de déploiement

Comme décrit dans le premier paragraphe de cette section, seul le déploiement des services nécessaires développeur est susceptible de « bloquer l'application ». Les autres déploiements sont exécutés en arrière plan. Par ailleurs, l'exécution des algorithmes de toutes les étapes de déploiement est en section critique. Ainsi, aucune interaction n'est possible entre les différentes étapes de déploiement. Dans la mise en œuvre que nous présentons dans la section 5.3.3, nous avons utilisé le modificateur *synchronized* de Java.

5.3.2 Stratégie de remplacement

Comme décrit dans le chapitre 1, la mémoire des terminaux mobiles devient de plus en plus consistante. Par exemple, l'*iPAQ hp6340* dispose de 64 Mo de *RAM* et de 64 Mo de *Flash ROM*. Cependant, les applications fonctionnant sur ces terminaux deviennent aussi de plus en plus gourmandes en mémoire. À titre d'exemple, la mémoire nécessaire pour lancer l'exécutable de OpenCCM sur un PDA est de 24 Mo (machine virtuelle IBM J9 2.2, intergiciel ORBacus 4.1.0 et plateforme OpenCCM 0.8.2). Le cache du terminal mobile ne peut pas garantir l'hébergement des composants déconnectés pour tous les composants de toutes les applications. Le gestionnaire du cache doit donc offrir un compromis entre la taille des composants déconnectés des applications et la taille limitée du cache. Ce compromis correspond à la stratégie de remplacement du cache.

Dans DOMINT, nous proposons une stratégie de remplacement qui se décline en trois étapes : à la demande, explicite et périodique. Les trois paragraphes suivants décrivent ces trois étapes. Le dernier paragraphe est consacré à la description des interactions entre les étapes de remplacement et aux interactions entre le déploiement et le remplacement.

À la demande

Le remplacement à la demande consiste à donner à l'utilisateur la possibilité de sélectionner des services (préalablement déployés) à supprimer du cache. Ceci est réalisé en présentant à l'utilisateur, via une interface graphique, les services de toutes les applications déployés dans le cache. Puisque la présence des services nécessaires développeur est obligatoire, l'utilisateur ne peut sélectionner que les services nécessaires utilisateur et les services non nécessaires. Le remplacement à la demande peut par exemple être utilisé lorsqu'une nouvelle application est lancée dans le terminal mobile alors que la taille du cache restante est insuffisante pour le déploiement de tous les services nécessaires développeur de cette application.

L'algorithme 5 présente le déroulement du replacement à la demande d'un service. Dans cet algorithme, la suppression d'un service échoue s'il existe au moins un service nécessaire (développeur ou utilisateur) dans le cache dont le lien d'utilisation entre ce service et le service à supprimer est nécessaire (ligne 6). La suppression d'un service revient à supprimer ces composants (nécessaires et non nécessaires). Cependant, la suppression d'un composant peut échouer

s'il est en conflit[3] avec le composant distant, ou s'il est utilisé par d'autre services dans le cache (ligne 10). L'ordre de suppression des composants suit une politique d'ordonnancement qui prend en compte la nécessité et la priorité (ligne 8). La politique d'ordonnancement utilisée est sélectionnée dans la configuration du gestionnaire du cache. L'utilisateur peut à tout moment changer la politique d'ordonnancement. Le prototype du gestionnaire du cache que nous présentons dans la section 5.3.4 implante six politiques d'ordonnancement : FIFO, LFU, LRU, GDSF, SIZE et LFUPP que nous décrivons dans le chapitre 6.

Algorithme 5: `Boolean` **onDemandReplacement**(`Service` *sr*)

```
1  ServiceSet services ← sr.getAscendantServices()
2  Component c
3  Service s
4  if (sr.getNecessity() and sr.getNecessityKind()="developer") then return false
5  for all s ∈ services
6    if ( s.getNecessity() and s.getNecessityEdge(sr)) then return false
7  ComponentSet components ← getComponents(sr)
8  orderComponents(policy, components)
9  for all c ∈ components
10   if (c.isUpdated() and ¬c.isShared()) then removeComponent(c)
11 s.isDeployed ← false
12 return true
```

Explicite

Dans les étapes de déploiement à la demande et à l'invocation, lorsque le gestionnaire du cache essaye de déployer un composant alors que la taille du cache restante est insuffisante, il lance la fonctionnalité *expliciteReplacement* afin de libérer de l'espace mémoire dans le cache. En effet, cette fonctionnalité correspond à un remplacement anticipé que nous appelons remplacement explicite. Le déroulement de ce remplacement est présenté dans l'algorithme 6. Le gestionnaire du cache lance un remplacement à la demande (ligne 5) sur les services (nécessaires utilisateur et non nécessaires) jusqu'à ce que la taille du cache restante devienne supérieure ou égale à la taille donnée en paramètre de l'algorithme. Les services sont ordonnés selon une politique d'ordonnancement (ligne 2) comme pour les composants dans le remplacement à la demande. Cependant, afin d'éviter un bouclage à l'infini dans la suppression des composants (boucle *while* de la ligne 5), nous utilisons un compteur (*timer*) dont la valeur est configurable par l'utilisateur. La fonctionnalité *expliciteReplacement* renvoie *false* si le *timer* est expiré est la taille du cache restante est toujours inférieure à la taille donnée en paramètre de l'algorithme. Dans le cas contraire, la valeur du retour est *true*.

[3] Comme nous ne traitons pas le problème de la réconciliation ici, nous nous limitons à une simple défi nition du terme conflit. Ainsi, un composant déconnecté est en conflit avec son composant distant si les opérations exécutées sur le composant déconnecté n'ont pas été envoyées au composant distant.

Algorithme 6: `Boolean` **explicitReplacement**(`long` *size*)

```
1  ServiceSet services ← getServices()
2  orderServices(policy, services)
3  Service s ← services.getFirst()
4  int timer ← getTimer()
5  while ((getFreeCacheSize() < size) and (timer ≠ 0))
6    onDemandReplacement(s)
7    s ← services.getNext()
8    if (s = null) then s ← services.getFirst()
9    timer ← updateTimer()
10 if ((timer = 0) and (getFreeCacheSize() < size)) then return false
11 return true
```

Périodique

Comme décrit dans la section 5.3.1, l'utilisateur peut modifier le graphe de dépendances de l'application afin de déployer de nouveaux services (déploiement à la demande). L'utilisateur peut décider d'apporter ces modifications pour se préparer à une déconnexion volontaire (par exemple, avant l'embarquement dans un avion) ou involontaire (par exemple, passage dans un tunnel). Cependant, la taille du cache restante peut être insuffisante pour déployer les nouveaux services et les remplacements à la demande ou explicite peuvent être interrompus par une déconnexion volontaire ou involontaire. La solution que nous proposons consiste à garder une partie du cache vide pour des utilisations critiques. Nous appelons cette partie « la partie critique ». Périodiquement, le gestionnaire du cache essaye de maintenir la partie critique libre de toute utilisation. Nous appelons ce processus le remplacement périodique. La taille de la partie critique (pourcentage de la taille du cache) et la période sont fixés par l'utilisateur.

L'algorithme 7 présente le déroulement du remplacement périodique. Le gestionnaire du cache inspecte la taille du cache restante. Si elle est inférieure à la taille de la partie critique, le gestionnaire du cache lance un remplacement à la demande (ligne 5) sur les services (nécessaires utilisateur et non nécessaires) jusqu'à ce que la taille du cache restante devienne supérieure ou égale à la taille de la partie critique. Les services sont ordonnés selon une politiques d'ordonnancement (ligne 2) comme dans le remplacement explicite.

Algorithme 7: PeriodicReplacement()

```
1  ServiceSet services ← getServices()
2  orderServices(policy, services)
3  Service s ← services.getFirst()
4  while (getFreeCacheSize() < getCriticalSize())
5    onDemandReplacement(s)
6    s ← services.getNext()
7    if s = null then s ← services.getFirst()
```

Interactions entre les étapes de remplacement et entre déploiement et remplacement

De la même manière que dans le déploiement, l'exécution des algorithmes de toutes les étapes de remplacement est en section critique. Ainsi, aucune interaction n'est possible entre les différentes étapes de remplacement.

Par ailleurs, comme les listes des services et des composants de toutes les applications sont les mêmes dans le déploiement et le remplacement, il n'existe aucune interaction entre les étapes de déploiement et les étapes de remplacement. L'exécution des demandes (de déploiement ou de remplacement) est effectuée suivant l'ordre d'arrivée.

5.3.3 Architecture

Après avoir présenté le fonctionnement de base de notre service de gestion du cache (cf. sections 5.3.1 et 5.3.2), nous détaillons dans cette section la mise en œuvre de ce service pour des applications à base de composants CORBA. Nous utilisons le modèle de composants Fractal (cf. section 2.3.2) pour la modélisation et la réalisation du gestionnaire du cache. Nous ne détaillons pas tous les composants et toutes les interfaces. Nous nous limitons à la description des parties clés de notre réalisation. Nous divisons cette section en trois parties : architecture globale, conception détaillée, et enfin, gestion de la mémoire.

Architecture globale

La figure 5.2 illustre l'architecture du gestionnaire du cache. Le service de gestion du cache de DOMINT est représenté par un composite constitué de quatre composants Fractal : `DisComponentFactory`, `DiscEntryFactory`, `DisComponentCreator` et `PerseusCacheManager`. De plus, il fournit une interface fonctionnelle nommée *disComponent-factory* et trois interfaces de contrôles nommées *binding-controller*, *lifeCycle-controller* et *replacementAttribute-controller*. L'interface *disComponent-factory* est le point d'entrée pour utiliser le gestionnaire du cache. Puisque notre service doit être accessible comme un service CORBA, l'interface *disComponent-factory* est déclarée comme étant une interface CORBA. La figure 5.3 présente la déclaration en IDL de l'interface *disComponent-factory*. Elle fournit des méthodes pour permettre à une application de s'abonner au service (*subscribeCM*), désabonner du service (*unsubscribeCM*), effectuer un déploiement à la demande (*onDemandDeployment*), déployer un service (*deployService*), déployer un composant déconnecté (*createDisComponent*), chercher un composant déconnecté (*findDisComponent*), et enfin, lancer l'interface graphique de DOMINT (*startGUI*). Toutes ces méthodes peuvent lever l'exception *DiscCacheException*. Les interfaces de contrôles permettent la gestion des préoccupations extrafonctionnelles du composite. L'interface *binding-controller* gère les connexions entre les différents sous-composants. L'interface *lifeCycle-controller* gère le cycle de vie des sous-composants. Enfin, l'interface *replacementAttribute-controller* offre des méthodes pour la (re-)configuration du gestionnaire du cache (taille du cache, taille de la partie critique, période, politique de remplacement. . .).

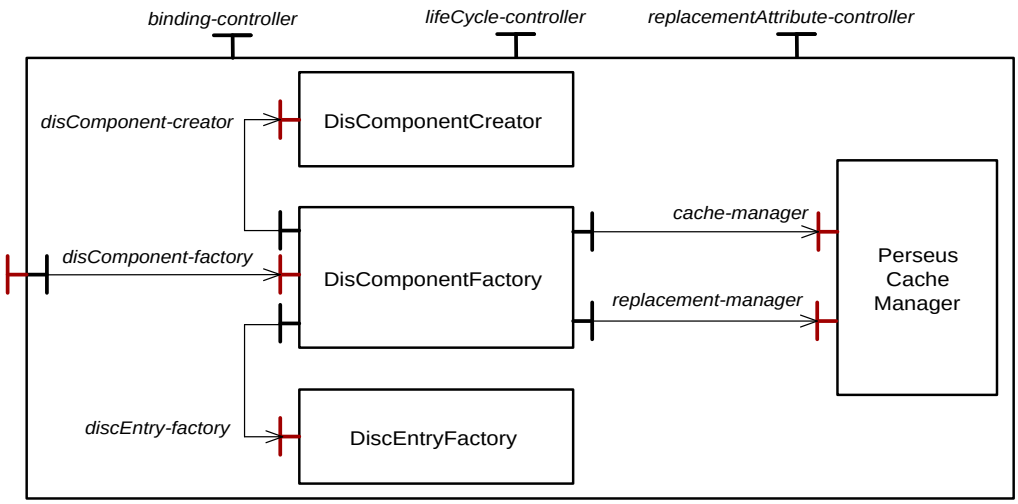


FIG. 5.2 – Architecture du gestionnaire du cache

```
1  module cm {
2     exception DiscCacheException {};
3     interface DisComponentFactory {
4        void subscribeCM(in string graph) raises (DiscCacheException);
5        void unsubscribeCM(in string graph) raises (DiscCacheException);
6        void onDemandDeployment(in string graph) raises (DiscCacheException);
7        void deployService(in string graph, in string service) raises (DiscCacheException);
8        void createDisComponent(in string compName) raises (DiscCacheException);
9        Object findDisComponent(in string compName) raises (DiscCacheException);
10       void startGUI() raises (DiscCacheException);
11    };
12 };
```

FIG. 5.3 – Déclaration IDL de l'interface DisComponentFactory

Conception détaillée

L'implantation de l'interface *disComponent-factory* est déléguée au composant `DisComponentFactory`. Ce dernier coordonne les autres sous-composants pour l'implantation des stratégies de déploiement et de remplacement du cache. Il permet d'indexer et de gérer toutes les applications abonnées au gestionnaire du cache. Chaque application est représentée par une instance de la classe `ApplicationInf` qui donne toutes les informations concernant une application (nom, graphe de dépendances, services, interactions inter-services et composants). Lorsque le composant `DisComponentFactory` reçoit une demande d'abonnement d'une application (via la méthode *subscribeCM*), il crée une nouvelle instance de `ApplicationInf`. Chaque service de cette application est représenté par la classe `Service` (cf. figure 5.4). Elle implante la classe Java `Comparable` afin d'implanter la méthode *compareTo* qui réalise les politiques d'ordonnancement. La version actuelle de cette méthode implante six politiques d'ordonnancement : FIFO, LFU, LRU, GDSF, SIZE et LFUPP (cf. chapitre 6). Elle utilise les différents attributs de la classe `Service` (nécessité, type de nécessité, priorité, fréquence d'utilisation, taille...). La classe `ApplicationInf` fournit un attribut (*sortedService*)

de type Java `TreeSet` qui donne la liste de tous les services de l'application triés d'une manière descendante (du plus nécessaire vers le moins nécessaire, du plus prioritaire vers le moins prioritaire, du plus fréquemment utilisé vers le moins fréquemment utilisé. . .).

```
1  public class Service implements Comparable {
2      private String name;
3      private boolean necessity;
4      private String necessityKind;
5      private int priority;
6      private int frequency;
7      private boolean isDeployed;
8      private long size;
9      private String application;
10     private TreeSet sortedComponent;
11     public Service(boolean _necessity, String _necessityKind,
12           int _priority, String _name ){...}
13     ...
14     public int compareTo(Object o) {...}
15 }
```

FIG. 5.4 – Modélisation d'un service déconnecté dans le cache

De la même manière que les services, un composant déconnecté est représenté dans le cache par une instance de la classe `PreDiscEntry` (cf. figure 5.5) qui implante la classe Java `Comparable`. En plus des attributs « nécessité », « type de nécessité », « priorité », « est déployé », « est en conflit », « fréquence » et « taille », cette classe fournit un attribut de type Java `Hashtable` qui énumère toutes les applications qui utilisent ce composant. Elle fournit aussi un attribut de type `DiscEntry` qui encapsule les références CORBA du composant distant, du composant déconnecté et de sa maison.

```
1  public class PreDiscEntry implements Comparable{
2      private String name;
3      private boolean necessity;
4      private String necessityKind;
5      private int priority;
6      private int frequency;
7      private String disComponentName;
8      private String jarFile;
9      private boolean isChared;
10     private boolean isDeployed;
11     private boolean isInConflict;
12     private long size;
13     private String registerDisComponent;
14     private String registerComponent;
15     private DiscEntry entry;
16     protected Hashtable appliList;
17     public PreDiscEntry(boolean _necessity, String _necessityKind, int _priority,
18     String _name, String _discName, String _jarFile, boolean _isChared){...}
19 ...
20     public int compareTo(Object o) {...}
21 }
```

FIG. 5.5 – Modélisation d'un composant déconnecté dans le cache

Le composant `DisComponentCreator` contrôle le déploiement des composants déconnectés. Il offre et implante l'interface *disComponent-creator* utilisée par le composant `DisComponentFactory`. Le processus de déploiement des composants dans le cache se base

```
<!DOCTYPE componentassembly PUBLIC "..." "...componentassembly.dtd">
<componentassembly derivedfrom="" id="DiscAvailableSeat">
  <componentfiles>
    <componentfile type="" id="DiscAvailableSeatCSD">
      <fileinarchive name="archives/disc_availableSeat.car">
      </fileinarchive>
    </componentfile>
  </componentfiles>
  <partitioning>
    <homeplacement cardinality="1" id="DiscAvailableSeatHome">
      <componentfileref idref="DiscAvailableSeatCSD"/>
      <componentimplref idref="DiscAvailableSeatImpl"/>
      <registerwithhomefinder name="InternetTicket-DiscAvailableSeatHome"/>
      <componentinstantiation id="DiscAvailableSeat">
         <registercomponent>
            <registerwithnaming name="InternetTicket/DiscAvailableSeat"/>
         </registercomponent>
      </componentinstantiation>
      <destination>
         <installation type="componentinstallation">
            <findby>
               <namingservice name="CacheComponentServer"/>
            </findby>
         </installation>
         <activation type="componentserver">
            <findby>
               <namingservice name="CacheComponentServer"/>
            </findby>
         </activation>
      </destination>
    </homeplacement>
  </partitioning>
</componentassembly>
```

FIG. 5.6 – Exemple de descripteur de déploiement généré par le gestionnaire du cache

sur la chaîne de déploiement de OpenCCM. Tous les composants déconnectés sont déployés localement dans un serveur de composants appelé *CacheComponentServer*. Ce dernier est lancé dans le script de démarrage du gestionnaire du cache. Le processus de déploiement d'un composant déconnecté est initié par le composant `DisComponentFactory`. Il récupère l'archive[4] du composant déconnecté à partir du chemin récupéré du descripteur de dépendances (*.gxl*). En utilisant le descripteur CORBA du composant (*.csd*) récupéré dans cette archive et le descripteur de dépendances de l'application (*.gxl*), le composant `DisComponentFactory` génère un descripteur d'assemblage (*.cad*) qui permet de déployer le composant et sa maison dans le serveur de composants *CacheComponentServer*. La figure 5.6 présente le descripteur d'assemblage généré pour le composant `DiscAvailableSeat` de l'application *InternetTicket*.

Une fois le fichier d'assemblage généré, le composant `DisComponentFactory` crée une archive d'assemblage[5] (en utilisant la librairie de classes *java.util.zip*) qui contient le descripteur d'assemblage généré et l'archive du composant (*.car*). Ensuite, le composant `DisComponentFactory` appelle le composant `DisComponentCreator` via la méthode *create* de l'interface *disComponent-creator*. Cette méthode prend en paramètre le chemin de l'archive d'assemblage. La figure 5.7 donne le code simplifié de l'implantation de l'in-

[4] Dans OpenCCM, l'extension de l'archive d'un composant est *.car* ou *.zip*. Cette archive contient l'exécutable du composant (*.jar*) et le descripteur CORBA du composant (*CORBA Software Descriptor*, *.csd*).

[5] OpenCCM accepte les extensions *.aar* et *.zip* pour l'archive d'assemblage.

```
1  public class DisComponentCreatorImpl implements DisComponentCreator{
2      private org.objectweb.ccm.deploytool.DeploymentApplication dep;
3      private String[] args2;
4      public DisComponentCreatorImpl(String[] args){
5          args2 = new String[args.length +3];
6          for (int i=0; i<args.length; i++){
7             args2[i]=args[i];
8          }
9          args2[args2.length-3] = "-F";
10         args2[args2.length-2] = "DefaultFactory";
11         dep = new DeploymentApplication();
12     }
13     public void create(String aarFile) {
14         args2[args2.length-1] = aarFile;
15         dep.start(args2);
16     }
17 }
```

FIG. 5.7 – Utilisation du mécanisme de déploiement d'OpenCCM

terface `DisComponentCreator` (c'est l'interface Java qui correspond à l'interface Fractal *disComponent-creator*). Le constructeur de cette classe crée un tableau de chaînes de caractères qui comporte en plus du contenu du tableau passé en paramètre au constructeur, les options *-F* et *DefaultFactory* pour spécifier au mécanisme de déploiement de OpenCCM l'utilisation d'une fabrique de composants nommée *DefaultFactory* auparavant lancée. Le tableau de chaînes de caractères passé en paramètre au constructeur comporte principalement les références du service de nommage et du DCI (*Distributed Computing Infrastructure*) qui gère les assemblages des composants. Enfin, le constructeur crée une instance de la classe `DeploymentApplication` de OpenCCM. Cette classe est utilisée pour déployer une application CCM à partir d'un descripteur d'assemblage. Ce déploiement consiste à instancier les conteneurs dans les serveurs des composants, instancier les maisons dans les conteneurs, instancier les composants à partir des maisons, et enfin, connecter les différentes interfaces des composants. À chaque invocation de la méthode *create*, nous ajoutons le chemin de l'archive d'assemblage comme élément du tableau de chaînes de caractères créé dans le constructeur. Le tableau qui résulte est donné en paramètre à la méthode *start* de la classe `DeploymentApplication`. La méthode *start* lance le déploiement du composant représenté par l'archive d'assemblage.

Dans CCM, la fin du déploiement d'un composant est suivie par l'appel de la méthode *configuration_complete* par le conteneur. C'est une méthode de l'interface de rappel `EnterpriseComponent`. Le développeur peut implanter cette méthode afin de configurer le composant, par exemple, pour initialiser les attributs du composant ou lancer son interface graphique. Dans notre approche, nous utilisons cette méthode pour l'initialisation de l'état du composant déconnecté. Comme décrit dans la section 4.4.3, le composant déconnecté dispose de l'interface requise `StateTransfer` offerte par le composant distant. Elle est utilisée pour la capture et la restauration de l'état du composant déconnecté. Ainsi, pour initialiser son état, le composant déconnecté récupère la référence du composant distant, connecte son interface requise `StateTransfer` avec la même interface offerte par le composant distant et utilise les méthodes de cette interface afin de récupérer la valeur de ces attributs.

Concernant la mise en œuvre de la stratégie de remplacement, nous avons utilisé le composant `PerseusCacheManager` de Perseus [131]. Dans cette thèse, nous ne décrivons pas ce composant et nous nous limitons à la description des modifications que nous avons apportées. Afin d'intégrer notre stratégie de remplacement, nous avons défini le composant `DiscReplacementManager` qui implante le composant abstrait `ReplacementManager` de Perseus (cf. figure 5.8). Le composant `DiscReplacementManager` est utilisé pour la mise à jour des différentes propriétés associées à un composant déconnecté (fréquence d'utilisation, date de la dernière utilisation) et utilisées dans les politiques de remplacement.

Une fois le composant à remplacer choisi, le composant `DisComponentFactory` initie le processus de remplacement via la méthode interne *removeComponent* présentée dans la figure 5.9. Cette méthode prend en paramètre un `PreDiscEntry` (cf. figure 5.5). La suppression d'un composant se résume en six étapes : supprimer l'enregistrement du composant dans le service de nommage, supprimer le composant, supprimer la maison, supprimer le conteneur, supprimer l'entrée correspondante dans Perseus, et enfin, mettre à jour l'espace mémoire du cache.

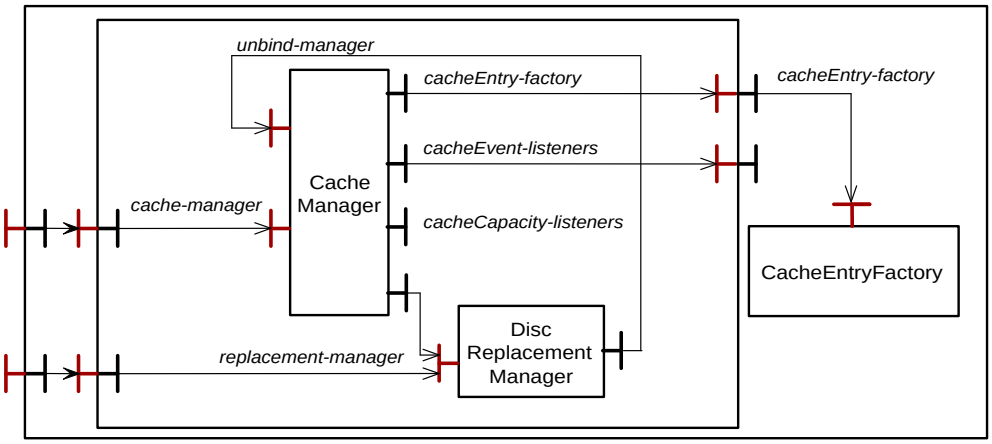


FIG. 5.8 – Architecture de Perseus

Gestion de la mémoire

L'espace mémoire occupé dans le cache par un composant déconnecté correspond à la taille de l'instance de ce composant, de sa maison et de son conteneur. Cependant, le calcul de cet espace est non trivial. La difficulté réside dans la spécification des machines virtuelles Java (JVM) qui n'offre pas de mécanismes dédiés au calcul précis des espaces mémoires occupés. Théoriquement, l'espace mémoire occupé par un objet est la différence entre l'espace mémoire disponible avant la création de l'objet et l'espace mémoire après la création de l'objet [174]. Cette technique donne approximativement l'espace mémoire occupé par l'objet à la création. Pratiquement, le seul mécanisme offert par Java pour le calcul de l'espace mémoire disponible dans une machine virtuelle Java est la méthode *freeMemory* de la classe `java.lang.Runtime`. À titre d'expérimentation, nous avons calculé la moyenne de l'espace mémoire occupé par un objet Java sur une série de 1000 créations d'instance. Pour éviter les interactions entre les créations

```
1  private void removeCompnent(PreDiscEntry pde){
2     String name = pde.getRegisterDisComponent();
3     DiscEntry de = pde.getDiscEntry();
4     CCMObject comp = de.getDisComponentRef();
5     CCMHome home = comp.get_ccm_home();
6     Container container=home.get_container();
7     NameComponent[] nam;
8     try {
9        nam = cm.getUsedNamingS().to_name(name);
10       cm.getUsedNamingS().unbind(nam);
11    } catch (InvalidName e2) {...}
12    try {
13       comp.remove();
14       container.remove_home();
15       container.remove();
16    } catch (RemoveFailure e1) {...}
17    pde.setIsDeployed(false);
18    perseusCM.unbind(de.getCacheEntry);
19    cm.setUsedMemory(cm.getUsedMemory()-pde.getSize());
20 }
```

FIG. 5.9 – Suppression d'un composant déconnecté

d'instance, nous invoquons explicitement le ramasse miettes de Java avant et après la création de l'instance de l'objet. Les résultats sont satisfaisants puisque la variance est moins de 1% de la taille moyenne obtenue. Nous avons donc décidé d'utiliser cette technique pour calculer l'espace mémoire occupé par un composant déconnecté dans le cache.

Cependant, la création du conteneur, de la maison et du composant se fait dans une JVM autre que celle lancée par le gestionnaire du cache. Il est ainsi impossible de récupérer l'espace mémoire disponible avant et après la création d'un composant à partir de la JVM lancée par le gestionnaire du cache. Cet espace doit être calculé à partir de la JVM qui a lancée le serveur de composants *CacheComponentServer* utilisé pour la création des conteneurs, des maisons et des composants déconnectés. En attendant de trouver une autre solution, nous avons apporté des modifications au code source de OpenCCM (version 0.8.2). Nous avons ajouté deux méthodes à l'objet CORBA `ComponentServer` (déclaré en OMG IDL dans *Components.idl* et implanté dans la classe `ComponentServerImpl`) : la méthode *getFreeMemory* qui renvoie l'espace mémoire disponible dans la JVM utilisée dans le déploiement des composants déconnectés et la méthode *expliciteGC* qui invoque explicitement le ramasse miettes de Java sur cette dernière JVM.

Nous avons refait les mêmes expérimentations décrites ci-dessus sur le déploiement des composants CORBA. Nous avons eu les mêmes résultats. De plus, le surcoût de la première création d'un composant CORBA était raisonnable. En effet, sur un jeu de dix tests, la taille du composant obtenue à la première création est entre 92% et 98% de la taille moyenne obtenue sur 1000 créations. Nous considérons donc que l'espace mémoire occupé par un composant déconnecté correspond à la taille obtenue à la première mesure. Ainsi, nous évitons l'attente due au calcul de la moyenne sur plusieurs créations.

Par ailleurs, nous avons présenté dans la section 5.3.2 le remplacement explicite (cf. algorithme 6). Ce dernier prend en paramètre la taille de l'espace mémoire à libérer. Cette taille correspond à la taille du composant en cours de déploiement. La question qui se pose alors est comment connaître la taille du composant avant le déploiement. Dans le cas où le composant

a été déployé auparavant (avant d'être supprimé), la taille du composant à utiliser dans l'algorithme 6 est la taille calculée dans le dernier déploiement. Dans le cas contraire, la taille à utiliser comme paramètre de l'algorithme 6 est la taille obtenue à partir du descripteur de dépendances. Le développeur peut calculer la taille des composants déconnectés et mettre cette information dans le descripteur de dépendances.

5.3.4 Caractéristiques des prototypes réalisés

Le prototype de la mise en œuvre pour composants CORBA a été réalisé dans le cadre de la plateforme libre DOMINT [39] et en partie dans les projets *ITEA Eureka OSMOSE* [128] et *RNTL franco-finlandais AMPROS* [1]. Dans cette section, nous présentons d'abord le prototype de base de notre service de gestion du cache, ensuite, nous présentons le portage de ce prototype sur terminaux mobiles (PDA).

Prototype de base

Le prototype de base a comme objectif le fonctionnement du service de gestion du cache sur des machines standards (ordinateur personnel et ordinateur portable). Le tableau 5.1 présente les caractéristiques de la distribution actuelle (version 0.1.0). La figure 5.10 présente une capture d'écran de l'interface graphique du service de gestion du cache. Via cette interface, l'utilisateur peut manipuler les applications dans le cache (par exemple, modification des méta-données et remplacement à la demande) et contrôler la taille du cache restante.

Prototype pour terminaux mobiles

L'implantation GXL que nous avons utilisée dans le prototype de base utilise la librairie *org.w3c.dom*. Cependant, la machine virtuelle pour PDA dont nous disposons (IBM J9 version 2.2) n'implante pas cette librairie. Ils en est de même pour l'interface graphique basé sur la librairie *javax.swing*. En conséquence, nous avons adapté notre prototype de base pour fonctionner sur les PDA. Nous avons implanté une version allégée de la librairie GXL que nous avons appelée *lightGXL*. Cette implantation se base sur la librairie nanoxml [117] pour l'analyse des fichiers XML (*.gxl*). Pour l'interface graphique, la dernière distribution opérationnelle du prototype pour PDA que nous avons implantée n'offre pas d'interface graphique. Nous espérons en proposer une en SWT[6]. Le tableau 5.2 présente les caractéristiques de la distribution actuelle du gestionnaire du cache pour PDA (version 0.1.0).

[6] SWT (*Standard Widget Toolkit*) est une bibliothèque graphique libre pour Java, initiée par IBM. Elle est plus légère que les bibliothèques Swing et AWT. De plus, elle est intégrée dans la JVM J9 d'IBM.

Source	– 258 Ko de sources dont 31 classes Java, 9 déclarations Fractal et 1 déclaration en OMG IDL – Fichier de confi guration (*confi g.properties*) – Script Jakarta Ant pour la compilation de la distribution – Deux démonstrations (*InternetTicket* et *Messenger*) – Scripts en ligne de commande (Unix/Linux et DOS)
Exécutable	– 112 Ko pour l'archive (*.jar*) du service de gestion du cache – 2.86 Mo nécessaire pour les archives externes utilisées par le service de gestion du cache (Fractal, GXL, xerces, log4j et Perseus)
Testé avec l'ORB	ORBacus-4.1.0
Testé sur les systèmes d'exploitation	– Linux RedHat 9.0 (noyau 2.4.20-8) – Windows 2000 – Windows XP
Requiert les logiciels	– OpenCCM (version 0.8.2) – ORBacus (version 1.4.0) – Jakarta Ant (version >= 1.6) – Environnement J2SE (version 1.4.2)

TAB. 5.1 – Caractéristiques du prototype de base

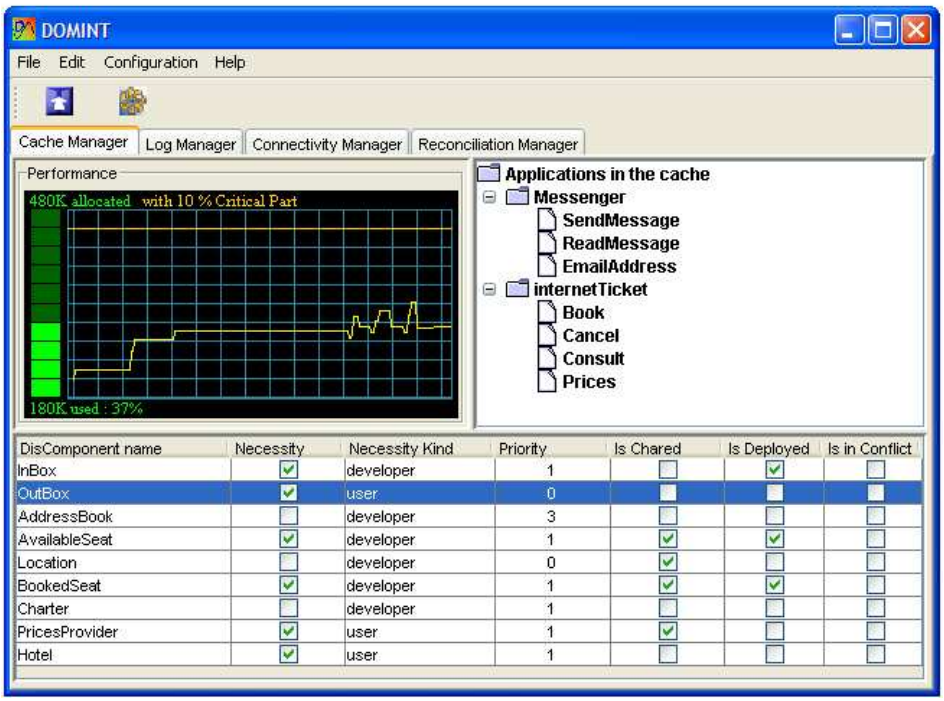


FIG. 5.10 – Interface graphique du service gestion du cache

Source	– 177 Ko de sources dont 20 classes Java, 9 déclarations Fractal et 1 déclaration en OMG IDL – Fichier de confi guration (*confi g.properties*) – Script Jakarta Ant pour la compilation de la distribution – Nouvelle tâche Ant pour la compilation Java avec *j9c* d'IBM J9 – Scripts en ligne de commande (DOS)
Exécutable	– 56.7 Ko pour l'archive (*.jar*) du service de gestion du cache – 646 Ko nécessaire pour les archives externes utilisées par le service de gestion du cache (Fractal, lightGXL, nanoxml, log4j et Perseus)
Testé avec l'ORB	ORBacus-4.1.0
Testé sur les systèmes d'exploitation	Windows Mobile 2003
Testé sur le PDA	*iPAQ hp6340*
Requiert les logiciels	– OpenCCM (version 0.8.2 pour PDA) – ORBacus (version 1.4.0) – Jakarta Ant (version >= 1.6) – Environnement J2ME (IBM J9 version 2.2 *Personal Profi le*)

TAB. 5.2 – Caractéristiques du gestionnaire du cache pour PDA

5.4 Conteneur du composant

Avant toutes considérations relatives à la mise en œuvre (cf. section 5.4.3), nous présentons dans cette section le modèle statique du conteneur du composant pour la gestion des déconnexions (cf. section 5.4.1). Ensuite, nous décrivons le modèle dynamique de ce conteneur dans les modes de fonctionnement connecté, partiellement connecté et déconnecté (cf. section 5.4.2).

5.4.1 Modèle statique

L'objectif de la proposition d'un conteneur de composants n'est pas de proposer un nouveau modèle de conteneur mais d'intégrer la gestion des déconnexions dans le conteneur. Cette gestion des déconnexions se résume aux différentes interactions entre le composant client (application côté client), le service de gestion du cache de DOMINT, et les composants serveurs (application côté serveur).

L'architecture du conteneur du composant pour la gestion des déconnexions est illustrée dans la figure 5.11. Cette architecture est basée sur le modèle présenté dans la section 2.3.2. Le conteneur offre deux intercepteurs : l'intercepteur des appels entrants et l'intercepteur des appels sortants. En plus de ces deux intercepteurs, nous définissons trois objets de contrôle : un contrôleur d'accès au service de gestion du cache (*CacheManager Access*, CA), un accès au détecteur de connectivité local (*Local Connectivity Detector*, LCD) et un gestionnaire de connecteurs (*Connectors Manager*, CM). Ces objets de contrôle sont des interfaces internes du conteneur.

Ils sont utilisés par les intercepteurs du composant (pré- et post-requête) et par le composant afin de gérer les déconnexions. Nous décrivons l'implantation de ces objets de contrôle dans la section 5.4.3.

Le contrôleur CA offre un accès au service de gestion du cache de DOMINT. Il permet à l'application côté client (composant client) de s'inscrire au service de gestion du cache, de chercher les références des composants déconnectés, de lancer le déploiement à la demande et à l'invocation, et de lancer le remplacement à la demande. Le contrôleur LCD définit une dépendance vers le service de détection de connectivité, de déconnexions et de défaillances de DOMINT (cf. section 5.2). Il offre un moyen local au conteneur pour gérer le mode de fonctionnement entre le composant hébergé et les composants connectés avec ce dernier. À l'instanciation du conteneur, le contrôleur LCD s'abonne au niveau du détecteur de connectivité local de DOMINT afin de recevoir des notifications des changements de mode de fonctionnement. Enfin, le contrôleur CM gère les connexions/déconnexions des connecteurs du composant client. Suivant le mode de fonctionnement, le composant client peut être lié soit avec les composants distants, soit avec les composants déconnectés.

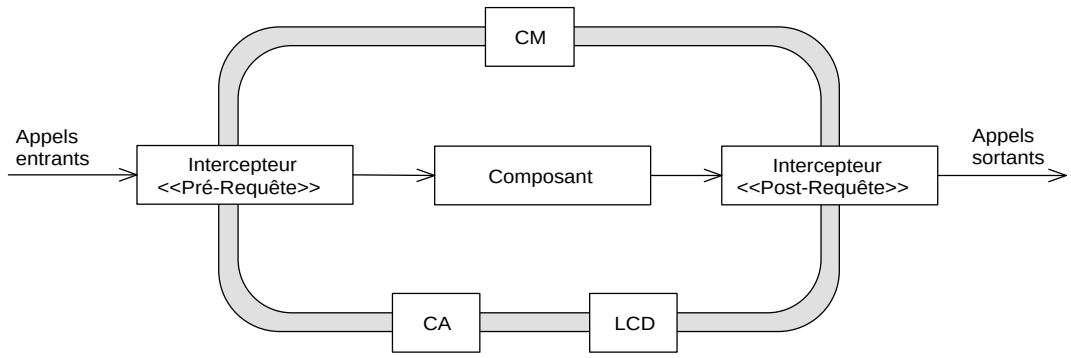


FIG. 5.11 – Architecture du conteneur du composant pour la gestion des déconnexions

5.4.2 Modèle dynamique

Nous illustrons dans cette section les différentes interactions entre le composant client (application côté client), le conteneur du composant client, le gestionnaire du cache et les composants distants (application côté serveur). Ces interactions sont en relation avec le mode de fonctionnement du terminal mobile (connecté, partiellement connecté et déconnecté). Ainsi, les trois paragraphes suivants décrivent ces interactions suivant le mode de fonctionnement.

Fonctionnement en mode connecté

Comme décrit dans la section 5.3.3, l'application doit souscrire au service de gestion du cache pour fonctionner en présence des déconnexions. Cette inscription correspond au lancement du déploiement au pré-chargement de l'application. Cependant, dans notre approche, l'application

(composant client) n'interagit pas directement avec le service de gestion du cache. En effet, c'est le conteneur de ce composant qui interagit avec ce service. La figure 5.12 présente le diagramme de séquences du déploiement au pré-chargement. Une fois l'installation et la configuration du composant client terminées (exécution de la méthode *configuration_complete*), l'intercepteur des appels entrants intercepte le retour de la méthode *configuration_complete* et lance le processus du déploiement au pré-chargement. Il récupère d'abord le chemin du descripteur de dépendances en invoquant la méthode *getgxlFile* de l'interface de rappel `DisconnectionCallBack` implantée par le composant client (cf. section 5.4.3). Ensuite, il récupère le mode de fonctionnement du terminal mobile à partir de LCD. Nous supposons que le lancement de l'application survient toujours en mode connecté afin de permettre aux composants déconnectés d'initialiser leur état. Ainsi, l'intercepteur des appels entrants demande à CA de lancer le déploiement au pré-chargement de l'application. Enfin, CA lance le déploiement au pré-chargement en invoquant la méthode *subscribeCM* du *CacheManager*.

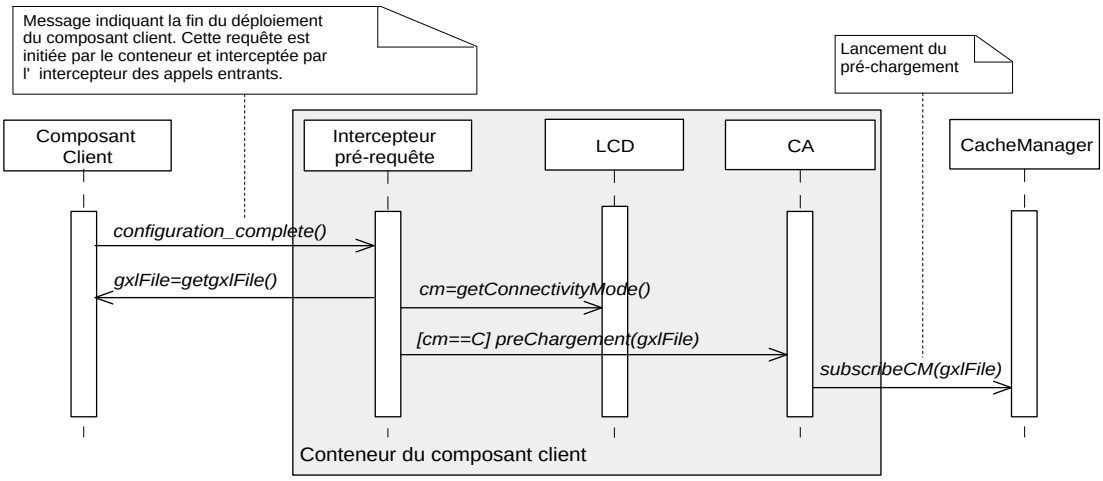


FIG. 5.12 – Lancement du pré-chargement par le conteneur du composant

Nous ne l'avons pas présenté sur le diagramme de séquences de la figure 5.12, mais à chaque fois que LCD reçoit une notification de changement du mode de fonctionnement, il informe le composant client de ce changement via la méthode *setConnectivityMode* de l'interface de rappel `DisconnectionCallBack`. Le composant client utilise cette information pour mettre à jour l'indicateur du mode de fonctionnement dans l'interface graphique de l'application et pour d'autres utilisations que nous décrivons dans la suite de cette section.

De la même manière, le déploiement à la demande est aussi géré par le conteneur du composant. Par contre, ce déploiement est initié par le composant lui-même. La figure 5.13 présente le diagramme de séquences du déploiement à la demande. Lorsque l'utilisateur apporte des modifications dans le graphe de dépendances, le composant client demande à CA d'initier le déploiement à la demande via la méthode *onDemandDeployment*. Ce dernier transmet la demande au gestionnaire du cache. Le composant client ne lance le déploiement à la demande que si le terminal mobile est en mode connecté. Dans le cas contraire, le déploiement à la demande sera lancé lorsque le mode de fonctionnement devient connecté.

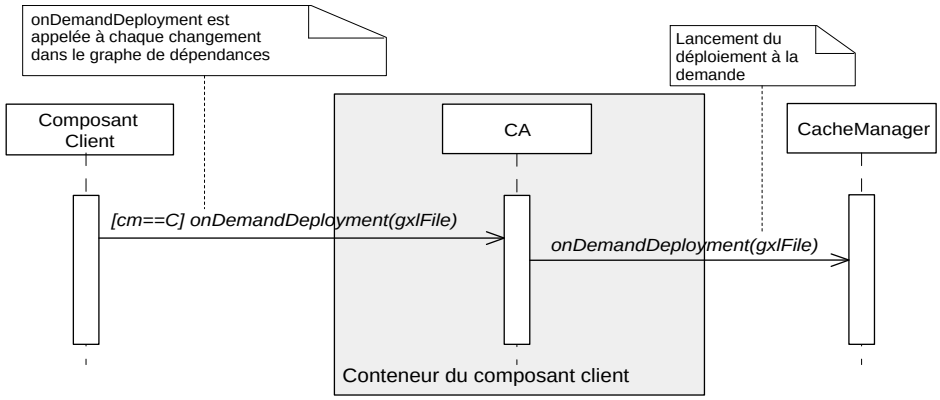


FIG. 5.13 – Lancement du déploiement à la demande via le conteneur du composant

Concernant le déploiement à l'invocation, le fonctionnement n'est pas le même pour le déploiement d'un service et pour le déploiement d'un composant. En effet, dans le premier cas, le déploiement est initié par le composant client et géré par le conteneur. Dans le deuxième cas, le déploiement est initié et géré par le conteneur et il est transparent au composant client. La figure 5.14 (respectivement 5.15) présente le diagramme de séquences du déploiement à l'invocation d'un service (respectivement d'un composant). Quand un utilisateur invoque un service, le composant client demande à CA de déployer ce service dans le cache (s'il n'existe pas) via la méthode *deployService*. Ce dernier transmet la demande au gestionnaire du cache. De la même manière que dans le déploiement à la demande, le composant client ne lance le déploiement à l'invocation que si le terminal mobile est en mode connecté. Pour le fonctionnement du déploiement d'un composant à l'invocation, toutes les requêtes du composant client vers les composants distants sont interceptées par l'intercepteur des appels sortants. Avant d'acheminer la requête au composant distant, l'intercepteur des appels sortants récupère le nom (*name* sur la figure 5.15) de ce composant distant. Ce nom est passé en paramètre à la méthode *createDisComponent* de CA qui achemine la demande de création d'un composant déconnecté au gestionnaire du cache. Cependant, la création d'un composant déconnecté ne peut être effectuée que si le terminal mobile est en mode connecté.

Fonctionnement en mode partiellement connecté

En mode partiellement connecté, les requêtes du composant client peuvent être exécutées soit par les composants distants, soit par les composants déconnectés (cf. figure 5.16). En effet, LCD offre la méthode *remoteExec* qui teste la possibilité de faire une invocation sur le composant distant passé en paramètre à cette méthode. Pour qu'une requête puisse être exécutée sur le composant distant, le temps nécessaire pour l'exécution de cette requête doit être inférieur au temps qui reste pour une éventuelle déconnexion (passage au mode déconnecté). Cependant, le calcul de ces temps est non trivial. Dans cette thèse, nous ne décrivons que le calcul du temps nécessaire pour l'exécution d'une requête sur un composant distant. Le temps qui reste pour une

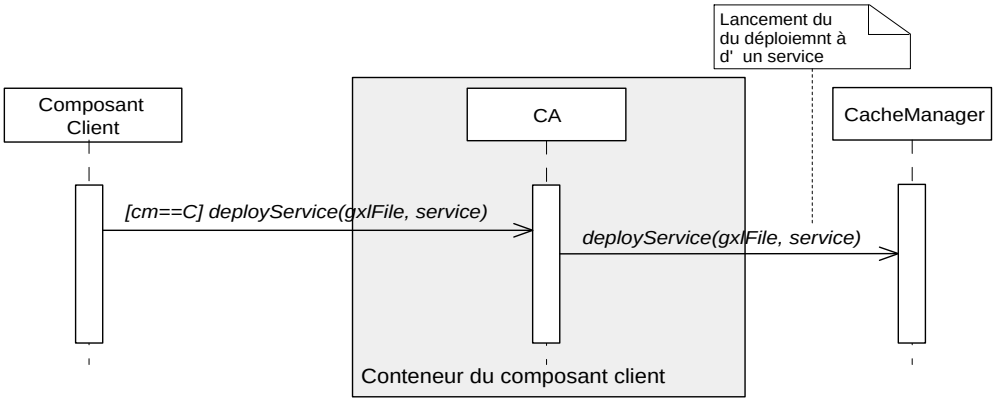


FIG. 5.14 – Lancement du déploiement d'un service à l'invocation

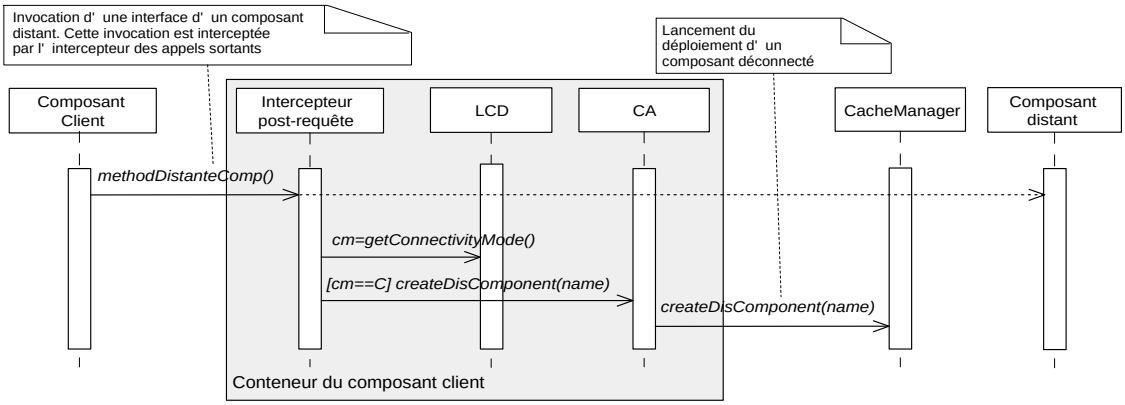


FIG. 5.15 – Lancement du déploiement d'un composant à l'invocation

éventuelle déconnexion est calculé par le service de détection de connectivité, de déconnexions et de défaillances de DOMINT [158].

Soit SZ_{input} (respectivement SZ_{output}) la taille des données de la requête (respectivement de la réponse), $BW_{cmp \rightarrow dst}$ (respectivement $BW_{dst \rightarrow cmp}$) la bande passante disponible du composant *cmp* vers le composant *dst* (respectivement du composant *dst* vers le composant *cmp*), T_{exec} le temps d'exécution dans le serveur de la méthode invoquée, et $T_{process}$ le temps nécessaire pour obtenir les valeurs de SZ_{input}, SZ_{output}, $BW_{cmp \rightarrow dst}$, $BW_{dst \rightarrow cmp}$ et T_{exec}. En se basant sur [57], le temps nécessaire pour l'exécution d'une requête du composant *cmp* vers le composant *dst* peut être calculé par la formule suivante[7] :

$$T(cmp, dst) = \frac{SZ_{input}}{BW_{cmp \rightarrow dst}} + \frac{SZ_{output}}{BW_{dst \rightarrow cmp}} + T_{exec} + T_{process} \quad (5.1)$$

[7] D'autre paramètres peuvent être considérés dans cette formule, par exemple, la latence du réseau, la latence d'exécution sur le terminal mobile, la latence d'exécution sur le serveur...

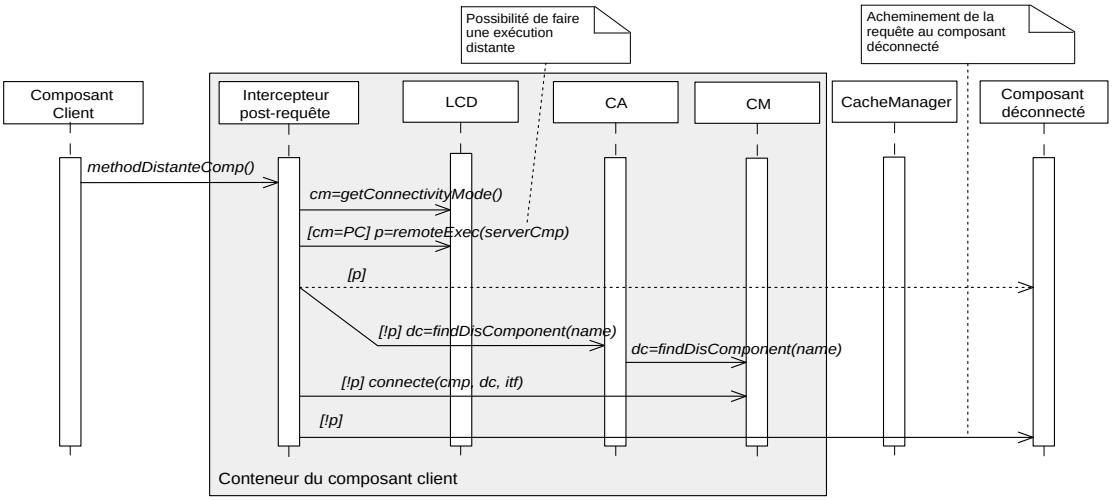


FIG. 5.16 – Invocation d'un composant distant en mode partiellement connecté

La difficulté avec cette formule réside dans le calcul des valeurs des variables. En effet, ces valeurs ne peuvent être obtenues sans effectuer des tests d'exécution. Par exemple, pour obtenir la valeur de T_{exec}, la requête doit être exécutée sur le serveur afin de calculer le temps d'exécution. Cette solution est inacceptable puisque c'est l'objectif même de la méthode *remoteExec*. La solution à cette problématique consiste à utiliser des mécanismes de prédiction. Ces mécanismes se basent sur le passé pour prédire le futur. Cependant, pour prédire la valeur d'une variable à l'instant t, il est nécessaire de connaître la valeur prédit et la valeur réelle de cette variable à l'instant $t-1$ (l'instant de la dernière mesure). De ce fait, nous supposons que la première requête du composant client vers un composant distant survient toujours en mode connecté, ce qui permet d'avoir la valeur réelle de la variable et pouvoir donc prédire la valeur de la prochaine invocation (si elle survient en mode partiellement connecté).

Pour prédire les valeurs des bandes passantes $BW_{cmp \rightarrow dst}$ et $BW_{dst \rightarrow cmp}$, et des temps d'exécution T_{exec} et $T_{process}$, nous utilisons le modèle de prédiction de Spectra [47] et Odyssey [48] : soit f_{t-1} la valeur d'une variable prédit à l'instant $t-1$, m_{t-1} la valeur mesuré de cette variable à l'instant $t-1$, et *k* le facteur de gain (entre 0 et 1). La valeur d'une variable prédit à l'instant t est donnée par l'équation suivante :

$$f_t = k \ * \ m_{t-1} + (1-k) \ * \ f_{t-1} \tag{5.2}$$

Par ailleurs, concernant le calcul de la valeur réelle de m_{t-1} de l'équation 5.2, les temps d'exécution sont donnés par la différence des temps après et avant l'exécution. Pour le calcul de la bande passante, nous utilisons l'équation donnée dans [118] : soit D un bloc de données, T_{win} le temps nécessaire pour envoyer le bloc D à un destinataire et recevoir la notification de réception, et T_{rtt} le temps nécessaire pour un aller/retour d'un RPC. La bande passante BW entre un émetteur et un récepteur est donné par l'équation suivante :

$$BW = \frac{D}{T_{win} - \frac{T_{rtt}}{2}} \tag{5.3}$$

Dans la solution proposée dans [34], l'exécution d'une opération en mode partiellement connecté ne dépend pas seulement de la possibilité d'envoyer la requête au serveur distant mais aussi de la sémantique de l'opération. En effet, les opérations possédant uniquement des paramètres en entrée (mot clé « `in` » en OMG IDL) sont exécutées d'abord localement, l'objet distant est mis à jour ensuite. Si le prototype de l'opération ne contient que des paramètres en sortie (« `out` ») ou une valeur de retour, l'opération est d'abord exécutée par l'objet distant, puis l'objet déconnecté mis à jour. Cependant, cette solution ne permet pas le mixage de paramètres « `in` » et « `out` », ni les paramètres « `inout` ». De plus, comme c'est le cas de notre solution, cette solution ne gère pas les exceptions renvoyées par l'objet distant.

Par manque de temps, et comme nous le présentons dans la section 5.4.3, la mise en œuvre actuelle de LCD n'implante pas la méthode *remoteExec*. De ce fait, nous assimilons le fonctionnement du mode partiellement connecté au mode déconnecté que nous décrivons dans le paragraphe suivant.

Fonctionnement en mode déconnecté

En mode déconnecté, toutes les requêtes du composant client sont exécutées sur les composants déconnectés. La figure 5.17 présente le diagramme de séquences lors d'une invocation d'un composant distant en mode déconnecté.

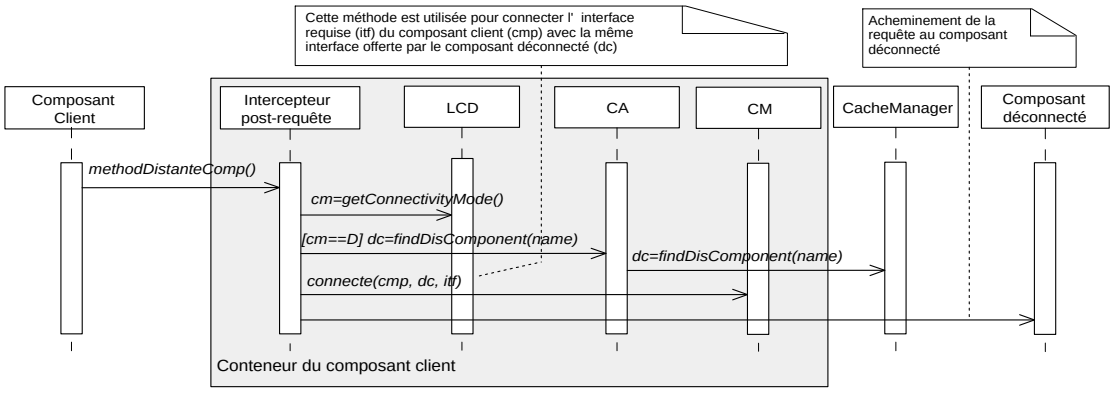


FIG. 5.17 – Invocation d'un composant distant en mode déconnecté

Dans notre approche, le mode déconnecté peut être le résultat soit d'une déconnexion involontaire, soit d'une déconnexion volontaire. Pour le deuxième type de déconnexion, l'objet de contrôle CA offre deux méthodes : *voluntaryDisconnection* et *voluntaryConnection*. La première méthode est utilisée pour effectuer une déconnexion volontaire, tandis que la deuxième est utilisée pour effectuer une connexion volontaire après une déconnexion volontaire. Le tableau 5.3 présente les effets des opérations *voluntaryDisconnection* et *voluntaryConnection* sur le mode de fonctionnement présenté à l'utilisateur. La première colonne de ce tableau correspond au mode de fonctionnement réel du terminal mobile avant la déconnexion ou la reconnexion volontaire. La deuxième colonne correspond à l'opération exécutée. Enfin, la dernière colonne correspond au

mode de fonctionnement du terminal mobile après l'exécution de l'opération. Après une déconnexion volontaire, le mode de fonctionnement devient toujours déconnecté. Par contre, après une reconnexion volontaire, le mode de fonctionnement résultant est le mode de fonctionnement réel du terminal mobile.

Mode de fonctionnement avant l'exécution de l'opération	Opération	Mode de fonctionnement après l'exécution de l'opération
Connecté	*voluntaryDisconnection*	Déconnecté
Connecté	*voluntaryConnection*	Connecté
Partiellement connecté	*voluntaryDisconnection*	Déconnecté
Partiellement connecté	*voluntaryConnection*	Partiellement connecté
Déconnecté	*voluntaryDisconnection*	Déconnecté
Déconnecté	*voluntaryConnection*	Déconnecté

TAB. 5.3 – Conséquences des déconnexions/reconnexions volontaires sur le mode de fonctionnement

5.4.3 Implantation avec ECM

Dans cette section, nous présentons d'abord le modèle du conteneur extensible (en anglais, ECM pour *Extensible Container Model*). Puis, nous détaillons la mise en œuvre de l'architecture du conteneur présentée dans les sections 5.4.1 et 5.4.2.

Présentation de ECM

L'objectif de cette section n'est pas de décrire le modèle du conteneur extensible en détail, mais de présenter juste les principaux concepts que nous utilisons et qui sont nécessaires pour comprendre l'implantation. La spécification CCM spécifie un modèle de conteneur du composant limité en terme de préoccupations extrafonctionnelles : les transactions, la persistance, la sécurité, le cycle de vie et les communications synchrone et asynchrone. En outre, la spécification CCM n'indique pas la manière dont le conteneur est construit. Pour surmonter ces limitations, ECM [165] a été proposé dans le cadre du projet *IST COACH* [33], afin de permettre à des applications de tirer bénéfice de l'approche composant et de la plateforme CCM. ECM propose de voir le conteneur CCM comme un ensemble de services extrafonctionnels (services CCM) contrôlés par le conteneur. Dans ECM, chaque service est construit suivant un modèle de service pour concevoir, implanter, empaqueter et déployer les services dans le conteneur. Ainsi, ECM offre des mécanismes et des API pour l'enregistrement et l'intégration des services dans le conteneur d'une manière transparente aux composants.

Un service CCM est défini par quatre éléments : un identifiant, des politiques, des interfaces internes et des interfaces de rappel. L'identifiant est utilisé pour identifier le service et ses interfaces (internes et de rappel) dans le conteneur. L'identifiant est simplement une chaîne de caractères spécifiée dans le langage OMG IDL. Les politiques représentent les différents types de politiques CORBA supportées par le service. Elles représentent les fonctionnalités déclaratives du service

associées à un comportement, comme les démarcations transactionnelles ou le contrôle d'accès. Les interfaces internes représentent un ensemble d'opérations implantées par le conteneur et utilisées par le composant pour la configuration du service dans le conteneur. Enfin, les interfaces de rappel représentent un ensemble d'opérations implantées par le composant et utilisées par le conteneur pour répondre à des évènements spécifiques.

Chaque service dispose d'un paquetage contenant un descripteur de service CSD très légèrement modifié par rapport à celui de CCM et un descripteur d'interface (CCSD pour *CORBA Component Service Descriptor*) introduit spécifiquement dans ECM. Ce dernier traduit l'interface du service dans un vocabulaire XML. En outre, afin d'intégrer un service dans un conteneur, ECM ajoute deux éléments XML dans le descripteur CORBA du composant (CCD) : l'élément *usedservices* et l'élément *policies*. L'élément *usedservices* permet de décrire l'ensemble des services utilisés par le composant. L'élément *policies* permet de décrire les associations entre les politiques du service et les éléments du composant (opération, interface, port ou composant lui-même).

Par ailleurs, la mise en œuvre des politiques d'un service est basée sur le patron Intercepteur Paramétrable[8] défini dans ECM. Ce patron est constitué de plusieurs éléments : l'intercepteur, l'aiguilleur et la fabrique d'intercepteurs. L'intercepteur est une interface qui capture un groupe d'événements paramétrés par l'interface du composant. Les événements interceptés sont paramétrés par rapport à la définition OMG IDL3 de l'interface du composant, c'est-à-dire l'opération, l'interface, le port et le composant lui-même. L'aiguilleur est responsable de l'ordonnancement des intercepteurs. La mise en œuvre des intercepteurs paramétrables est basée sur les intercepteurs portables de CORBA [122]. Cependant, par rapport aux fonctionnalités des intercepteurs portables de CORBA, l'implantation actuelle des intercepteurs paramétrables souffre de quelques limitations que nous avons identifiées dans notre réalisation. Nous les décrivons dans le paragraphe suivant.

Mise en œuvre

Afin d'intégrer la gestion des déconnexions dans les conteneurs des composants, nous utilisons le modèle du conteneur extensible. De ce fait, l'architecture du conteneur que nous avons présentée dans la section 5.4.1 doit être spécifiée comme étant un service CCM que nous appelons *DisconnectionMgtService*. La réalisation de ce service doit couvrir trois éléments : la spécification du service en OMG IDL, sa mise en œuvre et la production du paquetage.

Concernant la spécification du service *DisconnectionMgtService*, la figure 5.18 définit sa déclaration en OMG IDL. Les lignes 3 à 9 présentent la déclaration des identificateurs du service. Les lignes 10 à 29 correspondent à la définition des interfaces internes (`CA_Interface`, `LCD_Interface` et `CM_Interface`) et de rappel (`DisconnectionCallBack`). Enfin, les lignes 30 et 31 correspondent à la définition de notre politique de gestion des déconnexions. Dans CCM, les interfaces internes et de rappel doivent être définis comme interfaces locales (non accessibles de l'extérieur de l'ORB). Cependant, par rapport aux fonctionnalités de l'interface `LCD_Interface` (cf. section 5.4.2), cette dernière doit être accessible de l'extérieur. Afin

[8] Dans la spécifi cation COACH, les intercepteurs paramétrables sont nommés intercepteurs portables orientés composants (en anglais, COPI pour *Component-Oriented Portable Interceptor*).

```
...
module DisconnectionMgtService {
   const ::ECM::ServiceId     DISCONNECTION_SERVICE_ID  = "DiscServiceId";
   const ::CORBA::PolicyType  DISCONNECTION_POLICY_TYPE = 9999;
   const ::ECM::ServiceInternalId CA_INTERNAL_ID = "CAInternalId";
   const ::ECM::ServiceInternalId LCD_INTERNAL_ID = "LCDInternalId";
   const ::ECM::ServiceInternalId CM_INTERNAL_ID = "LCDInternalId";
   const ::ECM::ServiceCallbackId CALLBACK_ID = "DiscServiceCallbackId";
   const ::ECA::PolicyType    DISCONNECTION_SERVICE_POLICY = "DiscServicePolicy";
   local interface CA_Interface{
       void subscribeCM(in string gxlFile);
       void onDemandDeployment(in string gxlFile);
       void deployService(in string gxlFile, in string service);
       void createDisComponent(in string name);
       Object findDisComponent(in string name);
   };
   local interface LCD_Interface{
       char getConnectivityMode();
       boolean remoteExec(in Object serverComp);
       void voluntaryDisconnection();
       void voluntaryConnection();
   };
   local interface CM_Interface{
       void connect(in Object cmp, in Object dc, in string itf);
   };
   local interface DisconnectionCallBack{
       string getgxlFile();
       void setConnectivityMode(in char c);
   };
   local interface DiscServicePolicy : ::CORBA::Policy {
   };
};
```

FIG. 5.18 – Spécification en OMG IDL du service *DisconnectionMgtService*

de masquer cette limitation, l'implantation de l'interface `LCD_Interface` doit créer un objet CORBA qui interagit avec le service de détection de connectivité, de déconnexions et de défaillances de DOMINT afin de recevoir les notifications et doit les transmettre à l'interface interne `LCD_Interface`.

La mise en œuvre du service *DisconnectionMgtService* doit couvrir la mise en œuvre de la politique, des interfaces internes `CA_Interface`, `LCD_Interface` et `CM_interface`, de l'interface de rappel `DisconnectionCallBack` ainsi que les éléments requis par l'architecture du conteneur, c'est-à-dire l'intercepteur des appels entrants et l'intercepteur des appels sortants. L'implantation de ces deux intercepteurs se base sur les intercepteurs paramétrables. La figure 5.19 présente le diagramme de classes de la mise en œuvre du service *DisconnectionMgtService*. La classe `DiscServiceInitializerImpl` enregistre les différents éléments du service au travers des opérations *pre_init* (pré-initialiseur) et *post_init* (post-initialiseur) définies par l'interface ECM `ServiceInitializer`. Dans ECM, le pré-initialiseur enregistre uniquement l'initialiseur de l'ORB du service CORBA. Or, le service *DisconnectionMgtService* n'utilise aucun initialiseur de l'ORB d'un service CORBA. Ainsi, aucun traitement n'est ajouté dans le pré-initialiseur. Le post-initialiseur enregistre la mise en œuvre des interfaces internes (`CAInterfaceImpl`, `LCDInterfaceImpl` et `CMInterfaceImpl`), de la fabrique de politiques CORBA (`DiscPolicyFactoryImpl`) et de la fabrique d'intercepteurs (`DiscCallControllerFactoryImpl`). La fabrique `DiscPolicyFactoryImpl` permet d'instancier les politiques CORBA. Dans notre service, la seul politique qui existe est représentée

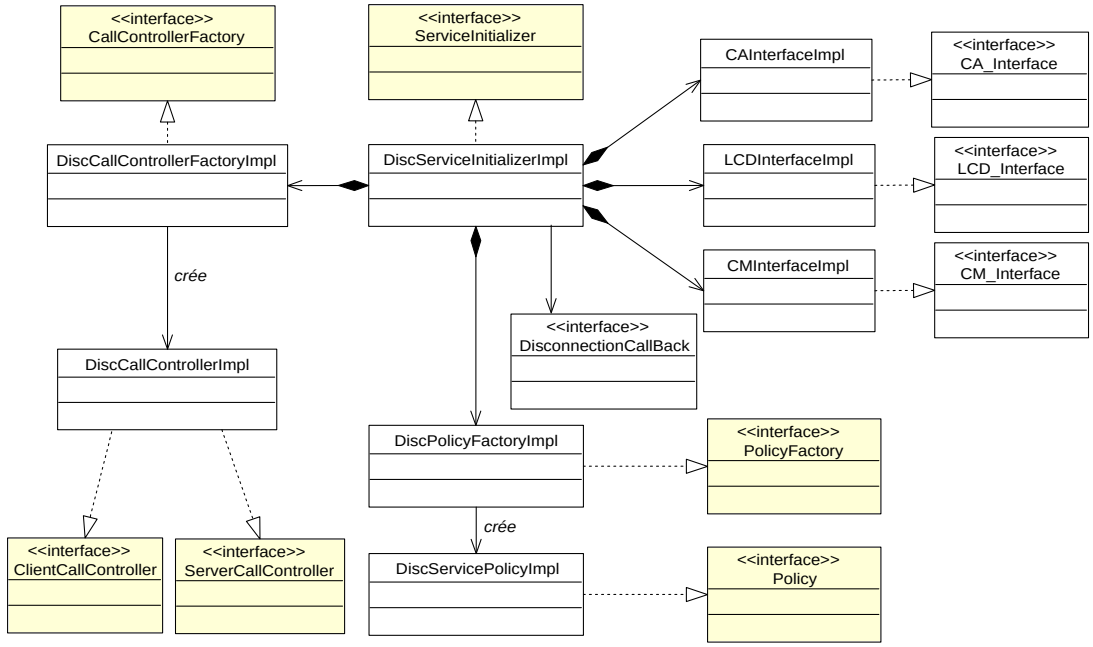


FIG. 5.19 – Diagramme des classes du service *DisconnectionMgtService*

par la classe `DiscServicePolicyImpl`. La fabrique `DiscCallControllerFactoryImpl` permet d'instancier l'intercepteur des appels sortants qui hérite de l'interface ECM `ClientCallController`, et l'intercepteur des appels entrants qui hérite de l'interface ECM `ServerCallController`. Ces deux intercepteurs sont implantés dans la classe `DiscCallControllerImpl`.

Nous ne détaillons pas plus en avant les programmes des différentes classes du fait de leur forte technicité. Nous notons juste que nous avons eu un problème dans la mise en œuvre de l'interface interne `CM_Interface`. Comme décrit dans la section 5.4.2, cette interface offre la méthode *connecte* qui permet de connecter une interface requise du composant client avec la même interface offerte par le composant déconnecté ou distant. Notre objectif était de récupérer les références du composant client et du composant déconnecté, et le nom de l'interface à partir de l'intercepteur des appels sortants. Malheureusement, nous n'avons pas pu aller loin dans cette idée du fait de la limitation de la réalisation actuelle de ECM (version 0.2.0). En effet, la seule information que les intercepteurs peuvent récupérer en interceptant une requête est le nom de l'opération invoquée. Cette information nous a été utile dans le lancement du pré-chargement (cf. figure 5.12). De plus, l'intercepteur des appels sortants de ECM ne permet pas la redirection de la requête vers une autre destination. La solution envisageable à ces limitations était soit de terminer l'implantation de ECM, soit de faire la redirection à l'extérieur du conteneur. La première solution ne rentre pas dans le cadre de nos objectifs (manque de temps). La deuxième solution nous semble la plus raisonnable.

Ainsi, la solution que nous proposons pour la redirection des requêtes consiste à intégrer le module de redirection des requêtes dans le composant lui-même. Le composant client doit im-

planter la méthode locale *connect*. Elle est appelée par le composant avant chaque invocation d'un composant distant. Le fonctionnement de cette méthode dépend du mode de fonctionnement du terminal mobile. Dans le cas où le terminal mobile est connecté, la méthode *connecte* récupère la référence du composant distant et connecte l'interface requise du composant client avec l'interface offerte du composant distant. Le nom de l'interface est passé en paramètre à la méthode *connecte*. Dans le cas où le terminal mobile est partiellement connecté ou déconnecté, la méthode *connecte* récupère la référence du composant déconnecté et connecte l'interface requise du composant client avec l'interface offerte du composant déconnecté.

Enfin, la dernière étape de la réalisation de notre service est la production du paquetage. Ce paquetage contient l'exécutable, le descripteur CSD et le descripteur CCSD de notre service. En outre, pour que notre service soit intégré dans le conteneur du composant client, le descripteur CCD de ce composant doit référencer ce service dans les éléments XML *usedServices* et *policies*.

5.5 Discussion

Comme décrit dans la section 3.1.1, nous nous basons sur une stratégie d'adaptation « en collaboration » pour traiter le problème des déconnexions des applications réparties. Cette adaptation est effectuée en mixant les trois types d'adaptation : statique, dynamique et auto-adaptation. L'approche MADA que nous avons proposée (cf. chapitre 4) est basée sur une collaboration entre l'utilisateur et l'application pour l'adaptation au problème des déconnexions. Cette adaptation est effectuée en mixant l'adaptation statique (construction du profil de l'application par l'architecte pendant le développement) et l'adaptation dynamique (personnalisation du profil de l'application par l'utilisateur à l'exécution). Pour atteindre nos objectifs, nous avons proposé un service intergiciel pour la gestion du cache du terminal mobile dans le cadre du canevas logiciel DOMINT. Ainsi, nous avons introduit l'intergiciel comme acteur d'adaptation. En effet, le déploiement et la gestion des composants déconnectés dans le cache définissent une auto-adaptation, guidée par le service de gestion du cache. De plus, cette adaptation définit une réflexivité structurelle de l'application (cf. section 2.3.3).

En outre, l'architecture du conteneur des composants que nous avons proposée offre un moyen flexible pour l'intégration du service de gestion des déconnexions d'une manière transparente à l'utilisateur. Ce conteneur offre aussi une commutation transparente entre les composants distants et les composants déconnectés suivant le mode de fonctionnement du terminal mobile. Le mécanisme de commutation définit ainsi une réflexivité comportementale de l'application (cf. section 2.3.3).

5.6 Synthèse

Dans la première partie de ce chapitre, nous avons présenté le service de gestion du cache de DOMINT. Comme décrit dans la section 3.2, un gestionnaire du cache doit offrir des stratégies

de déploiement et de remplacement du cache. De ce fait, nous avons présenté notre stratégie de déploiement qui se décline en trois étapes complémentaires : au pré-chargement, à la demande et à l'invocation. Nous avons présenté aussi notre stratégie de remplacement qui se décline en trois étapes : à la demande, explicite et périodique. Après avoir présenté ces stratégies, nous avons décrit la mise en œuvre de notre service pour des applications à base de composants CORBA. Cette mise en œuvre a été consolidée par deux prototypes de notre service : un prototype pour machines standards (ordinateur personnel et ordinateur portable) et un prototype pour terminaux mobiles (assistants personnels numériques).

Dans la deuxième partie de ce chapitre, nous avons présenté notre architecture du conteneur de composants. L'objectif de cette architecture est d'intégrer la gestion des déconnexions dans les conteneurs. Cette gestion des déconnexions se résume aux différentes interactions entre le composant client (application côté client), le service de gestion du cache de DOMINT, et les composants serveurs (application côté serveur). Nous avons présenté ces interactions suivant le mode de fonctionnement du terminal mobile. Enfin, en nous basant sur le modèle du conteneur extensible (ECM), nous avons proposé une réalisation de notre architecture.

Dans le chapitre suivant, nous présentons les différentes expérimentations qui ont été réalisées avec l'implantation.

Chapitre 6

Étude de performances

Nous avons présenté dans les deux chapitres précédents l'ensemble de nos propositions pour la gestion des déconnexions des applications réparties à base de composants en environnements mobiles. Dans ce chapitre, nous présentons les tests de performances du service de gestion du cache de DOMINT. Dans la section 6.1, nous introduisons les objectifs de ces tests. Ensuite, la section 6.2 décrit les résultats obtenus sur la manipulation du graphe de dépendances. La section 6.3 présente les tests de performances des stratégies de déploiement et de remplacement. Nous mesurons l'impact du service de gestion du cache de DOMINT sur l'utilisation de la batterie des terminaux mobiles dans la section 6.4. Enfin, nous synthétisons dans la section 6.5 les différents tests de performances que nous avons effectués.

6.1 Objectifs

Les tests que nous décrivons dans la suite de ce chapitre portent sur les performances du service de gestion du cache de DOMINT. Le premier objectif de ces tests est de quantifier le temps nécessaire au parcours du graphe de dépendances avant le lancement du déploiement au pré-chargement (cf. section 5.3.1). Ce temps correspond à la création de la structure `Graph`. Nous comparons les temps obtenus avec le prototype de base et le prototype pour PDA. Le deuxième objectif de nos tests concerne les stratégies de déploiement et de remplacement. Dans un premier temps, les tests ont pour but, comme nous l'avons précisé dans la section 5.3.1, de valider l'utilité du processus *optProcess* dans le déploiement au pré-chargement. Dans un deuxième temps, nous effectuons une suite de tests afin de valider l'utilisation du service et non du composant comme granularité de déploiement et de remplacement. Enfin, le dernier objectif de nos tests porte sur l'étude de l'impact de l'utilisation de notre service de gestion du cache sur la consommation de la batterie des terminaux mobiles.

6.2 Manipulation du graphe de dépendances

Nous présentons dans cette section les différents tests réalisés sur la manipulation du graphe de dépendances sur le prototype de base (cf. section 6.2.1) et sur le prototype pour PDA (cf. section 6.2.2).

6.2.1 Prototype de base

Cette section détaille les résultats obtenus sur la manipulation du graphe de dépendances dans le prototype de base du service de gestion du cache. Nous présentons dans le premier paragraphe l'environnement du test, avant de présenter les résultats dans le deuxième paragraphe.

Environnement du test

Afin de varier le nombre de services et de composants dans un graphe de dépendances, nous avons réalisé un simulateur de construction de graphes de dépendances. Il permet d'avoir des graphes de dépendances dont le nombre de services (respectivement composants) peut varier de 1 à n (respectivement de 1 à m). n et m sont des entiers supérieurs ou égales à 1. De plus, ces graphes de dépendances comportent des interactions intra- et inter-services, générées d'une manière aléatoire.

La machine utilisée pour les tests est celle du département Informatique de l'INT. C'est une machine *Pentium 3* dont la fréquence du processeur est de 933 MHz et la taille de la mémoire RAM est de 512 Mo. Le système d'exploitation utilisé est une distribution Linux RedHat 9.0 (noyau 2.4.20-8). La machine virtuelle Java utilisée est celle de SUN Microsystem et plus particulièrement celle du JDK 1.4.1.

Chaque test est exécuté 1000 fois afin d'avoir des moyennes significatives. De plus, un appel explicite au ramasseur de miettes de Java est effectué avant chaque exécution afin d'éviter d'avoir des interférences entre les différentes mesures.

Résultats

La figure 6.1 montre le temps moyen pour la construction de la structure `Graph` par le service de gestion du cache (prototype de base) avant le lancement du déploiement au pré-chargement. Les résultats montrent que le temps nécessaire pour la manipulation du graphe de dépendances avant le lancement du déploiement au pré-chargement est très faible, même dans les cas extrêmes avec une vingtaine de services et deux centaines de composants. Par exemple, le service de gestion du cache prend 1,15 ms (respectivement 8,99 ms) pour manipuler un graphe de dépendances de 1 service et 10 composants (respectivement 20 services et 200 composants). Par ailleurs, le temps de manipulation du graphe de dépendances n'est pas trop influencé par le nombre de services. Par exemple, la différence entre les temps de manipulation d'un graphe de dépendances de 1 service et 20 composants, et un graphe de dépendances de 20 services et

20 composants est 0,16 ms. Ainsi, la notion de service que nous avons introduite dans MADA n'ajoute qu'un surcoût très faible au déploiement dans le service de gestion du cache.

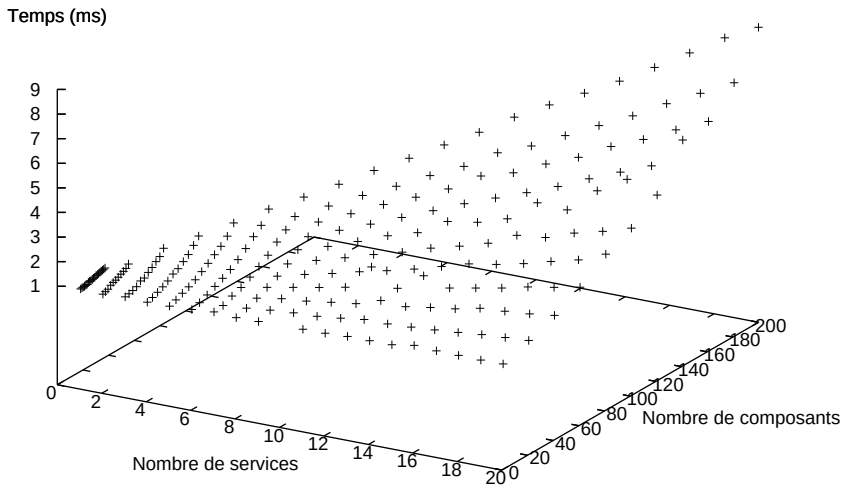


FIG. 6.1 – Temps d'interprétation du graphe de dépendances pour PC

6.2.2 Prototype pour PDA

Dans cette section, nous décrivons les mêmes tests présentés dans la section 6.2.1 mais sur le prototype pour PDA. Nous présentons dans le premier paragraphe l'environnement du test, avant de présenter les résultats dans le deuxième paragraphe.

Environnement du test

Nous avons apporté des modifications au simulateur décrit dans la section 6.2.1 afin d'utiliser la librairie *lightGXL* (cf. section 5.3.4).

Le PDA utilisé est un *iPAQ hp6340* équipé d'un processeur *Texas Instruments OMAP 1520* (fréquence équivalente à 192 MHz). Ce PDA dispose de 64 Mo de mémoire RAM et de 64 Mo de mémoire *Flash* ROM. Le système d'exploitation utilisé est Microsoft Windows Mobile 2003. La machine virtuelle Java utilisée est la J9 2.2 d'IBM.

De la même manière que pour les tests de la section 6.2.1, chaque test est exécuté 1000 fois et un appel explicite au ramasseur de miettes de Java est effectué avant chaque exécution.

Résultats

La figure 6.2 présente le temps moyen pour la construction de la structure `Graph` par le service de gestion du cache avant le lancement du déploiement au pré-chargement. Les résultats montrent que le temps nécessaire pour la manipulation du graphe de dépendances avant le lancement du déploiement au pré-chargement est très faible dans le cas où le nombre de composants est inférieur à 10. Cependant, au-delà de 10 composants, le temps nécessaire pour la manipulation du graphe de dépendances devient de plus en plus important. Par exemple, le service de gestion du cache prend 1,4 ms (respectivement 130,87 ms) pour manipuler un graphe de dépendances de 1 service et de 10 composants (respectivement de 20 services et de 200 composants). En comparant ces résultats avec ceux obtenus dans la section 6.2.1, nous constatons que le temps nécessaire pour la manipulation du graphe de dépendances avant le lancement du déploiement au pré-chargement augmente d'une manière exponentielle en fonction du nombre de composants.

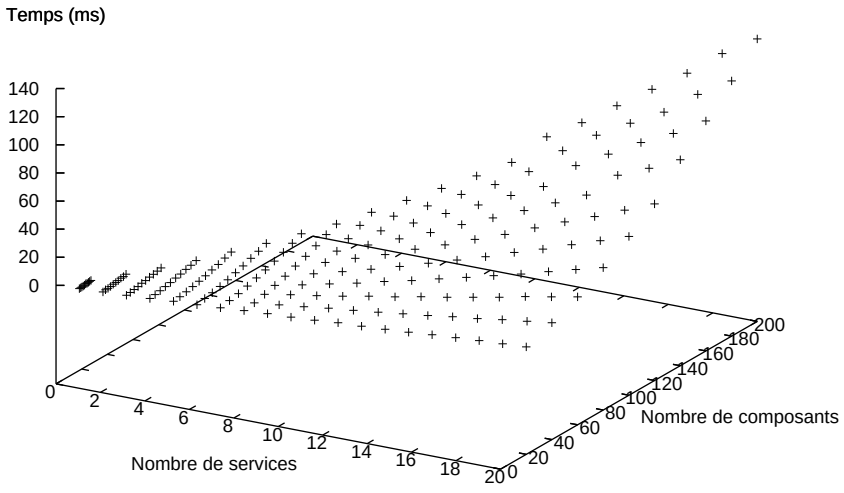


FIG. 6.2 – Temps d'interprétation du graphe de dépendances pour PDA

Nous justifions les valeurs obtenues dans ces tests par les capacités médiocres des ressources du PDA tant en mémoire qu'en vitesse d'exécution. Par ailleurs, afin de vérifier que ces résultats n'ont pas de relation avec la librairie *lightGXL*, nous avons effectués les mêmes expérimentations (prototype pour PDA et simulateur avec lightGXL) en utilisant l'environnement du test décrit dans la section 6.2.1. La figure 6.3 présente les résultats obtenus. Malheureusement, en comparant les résultats de la figure 6.3 avec ceux de la figure 6.1, nous constatons que le surcoût observé sur PDA est généré en grande partie par la librairie *lightGXL*. Cependant, dans tous les cas, les résultats obtenus sont acceptables pour les applications dont le nombre de composants est petit (entre 1 et 10 composants). Nous rappelons que l'objectif de base de la librairie *lightGXL*

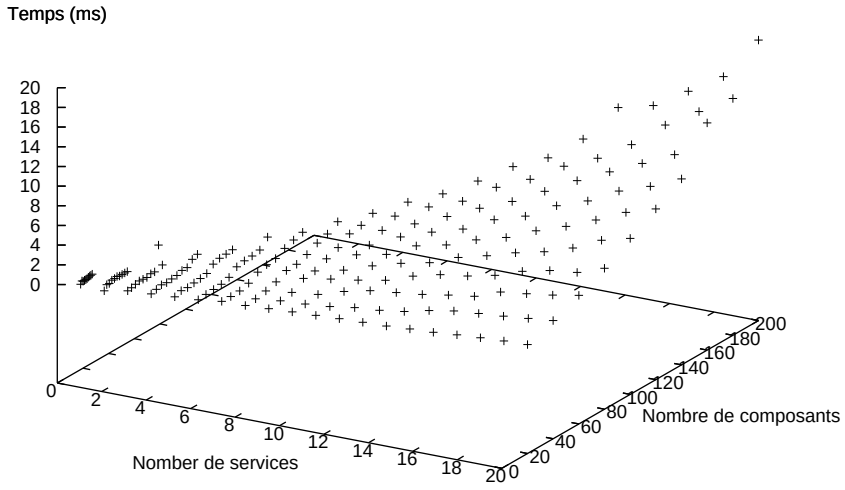


FIG. 6.3 – Temps d'interprétation du graphe de dépendances pour PC avec lightGXL

est de pouvoir manipuler des fichiers GXL avec la machine virtuelle J9. En conséquence, un travail d'optimisation est donc à faire afin de minimiser le temps de manipulation des graphes de dépendances.

6.3 Stratégies de déploiement et de remplacement

Les tests que nous présentons dans cette section ont pour objectif de mettre en valeur les stratégies de déploiement et de remplacement que nous avons présentées dans le chapitre 5. Nous décrivons d'abord l'environnement du test, avant de présenter l'ensemble des résultats des différents tests.

6.3.1 Environnement du test

L'environnement du test que nous avons utilisé est constitué de deux machines : un PC portable *Pentium 3* dont la fréquence du processeur est de 700 MHz avec une mémoire RAM de 256 Mo, et un PC fixe décrit dans la section 6.2.1. Le système d'exploitation utilisé dans le PC portable est Microsoft Windows 2000 avec la machine virtuelle J2SDK 1.4.2. Le bus logiciel utilisé dans les deux machines est l'ORB ORBacus 4.1.0 pour Java au-dessous de OpenCCM 0.8. Le PC portable est relié au PC fixe à travers un réseau filaire avec une bande passante de 100 Mo/s. Nous avons choisi d'utiliser un réseau filaire pour deux raisons : l'objectif n'est pas de mesurer le débit ou la latence du réseau, mais l'objectif est d'étudier la disponibilité du cache. Par ailleurs,

toutes les requêtes du client sont systématiquement redirigées vers les composants déconnectés, et les composants distants ne sont utilisés que pour le transfert d'état.

Nous avons utilisé une application CCM de 15 services et 50 composants simples (sans attributs ni méthodes). Les 50 composants sont déployés sur le PC fixe. Au niveau PC portable, l'utilisation des services et des composants de l'application se fait via un composant client. Dans le descripteur du graphe de dépendances, nous avons déclaré 5 services nécessaires développeur, 5 services nécessaires utilisateur et 5 services non nécessaires. Par ailleurs, afin de générer les traces d'accès (invocations) du PC portable vers le PC fixe, nous avons implanté un générateur. Ce dernier prend en paramètre le nombre de services, le nombre de composants, le graphe de dépendances et le nombre de requêtes à générer. La trace d'accès générée peut être paramétrée suivant les méta-données des services et des composants. Par exemple, dans nos tests, nous avons utilisé deux traces d'accès de 1000 requêtes : la première (respectivement la deuxième) comporte 80% de requêtes sur des services (respectivement des composants) nécessaires, 10% de requêtes sur des services (respectivement des composants) non nécessaires avec des priorités élevées, et 10% de requêtes générées d'une manière aléatoire sur l'ensemble des services (respectivement des composants). Le temps séparant deux requêtes est de 30 secondes. Pour les tests, nous n'utilisons que les déploiements au pré-chargement et à l'invocation, et uniquement les remplacements périodique et explicite. La période du remplacement est 1 minute et la taille de la partie critique est 10% de la taille du cache.

L'ensemble des tests que nous présentons dans cette section se basent sur deux métriques : le *Hit Rate* (HR) et le *Byte Hit Rate* (BHR). HR est le rapport entre le nombre de requêtes satisfaites par le cache et le nombre total de requêtes. BHR représente le pourcentage de la taille des données utilisées dans le cache plutôt que dans le serveur distant. Les tests portent sur la mesure des HR et BHR en faisant varier la taille du cache. Les mesures sont effectuées au niveau du PC portable (côté de l'utilisateur final).

Dans les tests que nous présentons dans la sous-section suivante, la gestion des tailles du cache et des composants est émulée. En effet, nous initialisons l'attribut *size* de la classe `PreDiscEntry` des composants déconnectés (cf. section 5.3.3) à une valeur comprise entre 10 Ko et 150 Ko. La taille des services est la somme des tailles de ces composants nécessaires. Elle est comprise entre 50 Ko et 150 Ko. Nous avons opté pour une émulation de la gestion des tailles afin de gagner le temps correspondant au calcul des tailles réelles.

Comme décrit dans le chapitre 5, la stratégie de remplacement utilise une politique d'ordonnancement afin de choisir l'entité du cache à supprimer. Nous calculons les HR et BHR pour cinq politiques d'ordonnancement : deux politiques traditionnelles, LFU et LRU, deux politiques utilisés dans le domaine des caches web, GDSF et SIZE (cf. section 3.3.4), et enfin, LFUPP pour *Least Frequently Used with Periodicity and Priority*, que nous proposons. LFUPP est dérivée de LFU. Lorsqu'une entité doit être supprimée du cache, l'entité choisie est celle la moins utilisée. Si plusieurs entités ont la même fréquence d'utilisation, la moins prioritaire est choisie. L'objectif de LFUPP n'est pas de proposer une nouvelle politique d'ordonnancement, mais plutôt de calculer les HR et BHR avec une politique qui prend en compte la méta-donnée « priorité ».

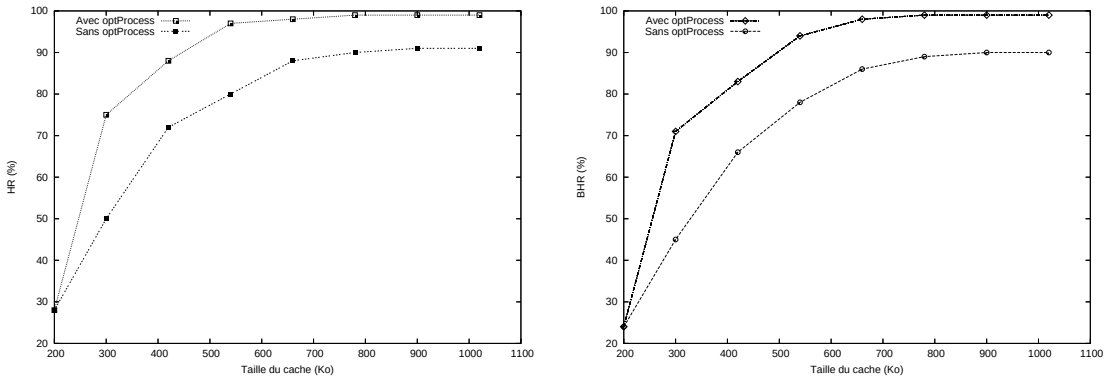
6.3.2 Résultats

Nous présentons d'abord les tests portant sur le déploiement au pré-chargement sans et avec le processus *optProcess*. Ensuite, nous justifions l'utilisation de la granularité service dans les stratégies de déploiement et de remplacement.

Déploiement au pré-chargement sans et avec le processus *optProcess*

La figure 6.4-a (respectivement 6.4-b) présente différentes mesures de HR (respectivement de BHR) en faisant varier la taille du cache. La politique d'ordonnancement utilisée est LFUPP. Comme décrit dans le chapitre 5, le déploiement au pré-chargement échoue si la taille restante du cache est insuffisante pour déployer tous les services nécessaires développeur. La taille de tous les services nécessaires développeur de notre application est de 200 Ko. Ainsi, les premières mesures des HR et BHR commencent à partir de 200 Ko.

Les résultats obtenus dans les courbes des figure 6.4-a et 6.4-b montrent que les valeurs des HR et BHR sont nettement supérieures (à l'exception du cache de 200 Ko qui est saturé dès le déploiement des services nécessaires développeur) dans le cas où le processus *optProcess* est utilisé dans le déploiement au pré-chargement. Clairement, dans le cas où le processus *optProcess* est utilisé, le cache comporte le maximum de services nécessaires (développeur et utilisateur). Par contre, les services nécessaires utilisateur sont tous déployés à l'invocation dans le cas où le processus *optProcess* n'est pas utilisé.



(a) HR *(b)* BHR

FIG. 6.4 – HR et BHR avec et sans le processus *optProcess*

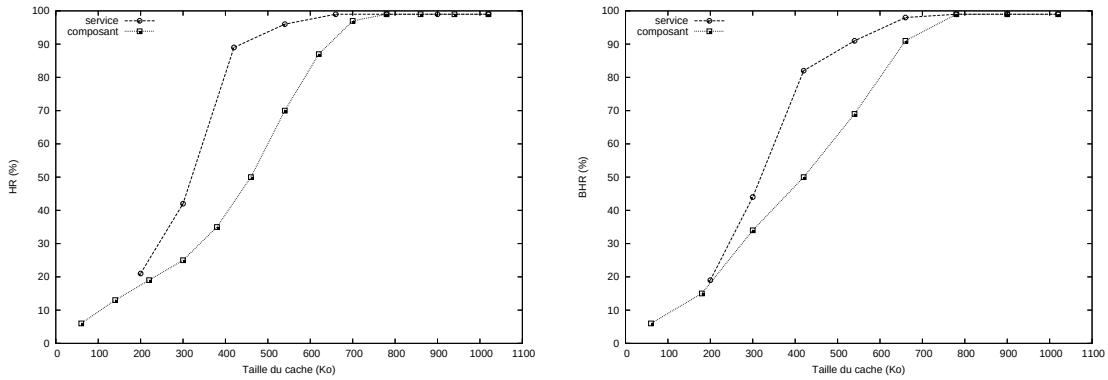
Granularité de déploiement et de remplacement

L'objectif de ces tests est de comparer la disponibilité du cache en utilisant d'abord le service comme granularité de déploiement et de remplacement, et ensuite le composant. L'utilisation du service comme granularité de déploiement et de remplacement correspond aux algorithmes et aux implantations que nous avons présentés dans le chapitre 5. Cependant, pour la granularité

composant, nous avons apporté des modifications à ces algorithmes et à ces implantations afin de faire abstraction de la notion de service. Pour la stratégie de déploiement, le déploiement au préchargement déploie d'abord tous les composants nécessaires développeur (processus bloquant), puis les composants nécessaires utilisateur dans le processus *optProcess*. Le déploiement à l'invocation déploie les composants nécessaires utilisateur et non nécessaires non encore déployés. Pour la stratégie de remplacement, les remplacements périodique et explicite ne concernent que les composants nécessaires utilisateur et non nécessaires. De plus, les interactions entre les composants sont prises en compte dans le déploiement et le remplacement.

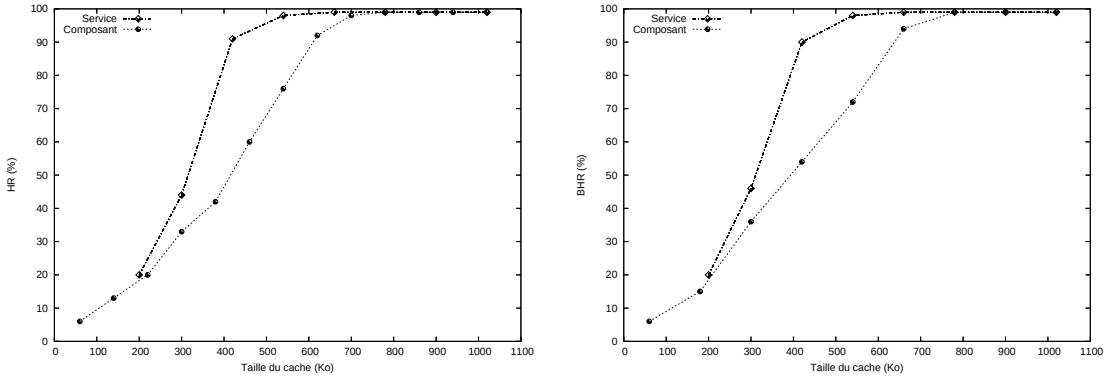
Les figures 6.5, 6.6, 6.7, 6.8 et 6.9 présentent différentes mesures des HR et BHR pour les politiques LRU, LFU, LFUPP, GDSF et SIZE respectivement. Ces résultats valident notre choix d'utiliser le service comme granularité dans les stratégies de déploiement et de remplacement. En effet, dans toutes les politiques que nous avons testées, la disponibilité du cache est supérieure dans le cas où le service est utilisé comme granularité comparé aux résultats obtenus en utilisant le composant comme granularité. Par exemple, en utilisant le service comme granularité, le HR (respectivement BHR) obtenu avec la politique LFUPP (cf. figure 6.7) pour un cache de 300 Ko est de 75% (respectivement 71%). Cependant, en utilisant le composant comme granularité, le HR (respectivement BHR) obtenu avec la politique LFUPP pour le même cache est de 53% (respectivement 46%). Nous expliquons ces résultats par le fait que la présence d'un service dans le cache est équivalente à la présence de ses composants nécessaires développeur.

Par ailleurs, en comparant les différentes courbes des figures 6.5, 6.6, 6.7, 6.8 et 6.9, nous constatons que la politique LFUPP offre une meilleure disponibilité pour les caches de petite taille (200 Ko à 350 Ko) par rapport aux autres politiques. Pour les caches de grande taille, toutes les politiques offrent une disponibilité comparable. Ces résultats démontrent donc l'intérêt de l'utilisation de la méta-donnée « priorité » dans la gestion du cache.



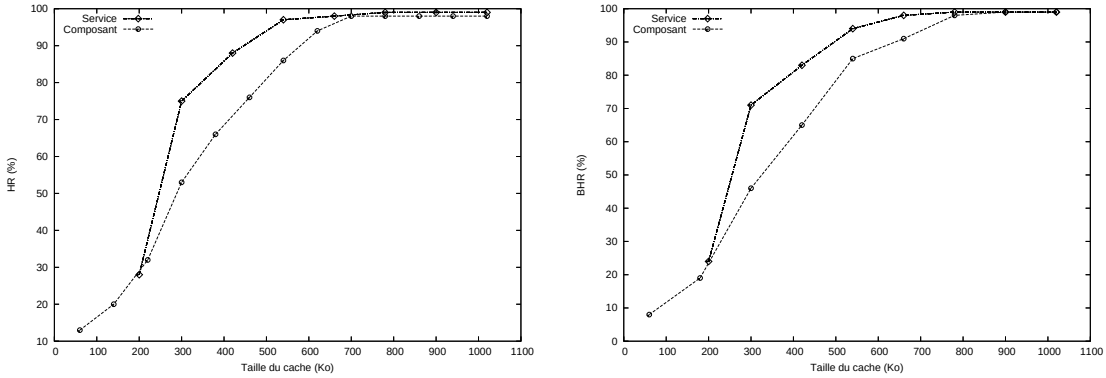
(a) HR *(b)* BHR

FIG. 6.5 – HR et BHR avec LRU comme politique d'ordonnancement



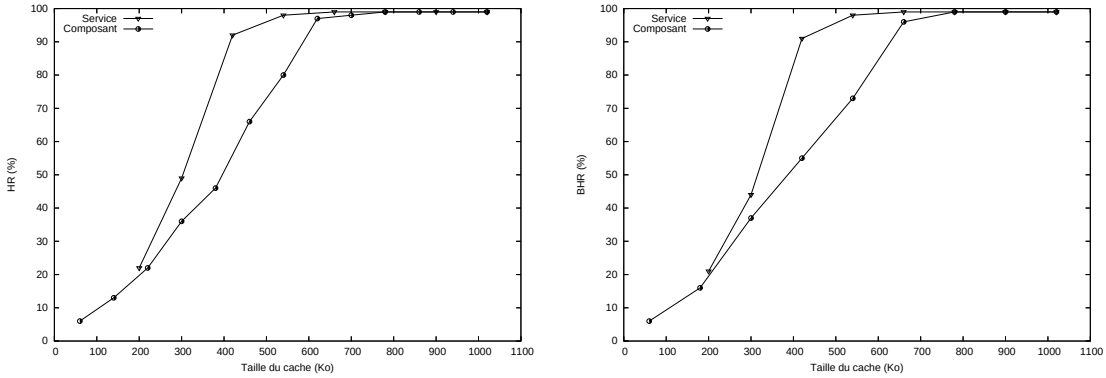
(a) HR *(b)* BHR

FIG. 6.6 – HR et BHR avec LFU comme politique d'ordonnancement



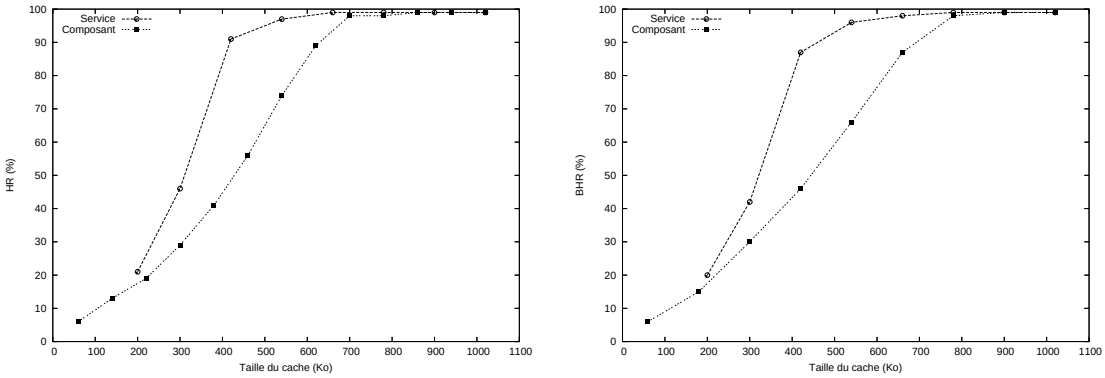
(a) HR *(b)* BHR

FIG. 6.7 – HR et BHR avec LFUPP comme politique d'ordonnancement



(a) HR *(b)* BHR

FIG. 6.8 – HR et BHR avec GDSF comme politique d'ordonnancement



(a) HR *(b)* BHR

FIG. 6.9 – HR et BHR avec SIZE comme politique d'ordonnancement

6.4 Mesures de batterie utilisée

L'objectif de ces tests est d'étudier l'impact de l'utilisation du service de gestion du cache de DOMINT sur la consommation de la batterie des terminaux mobiles. Nous décrivons dans la section 6.4.1 l'environnement du test. Ensuite, nous présentons dans la section 6.4.2 l'ensemble des résultats obtenus.

6.4.1 Environnement du test

L'environnement du test est constitué des deux machines décrites dans la section 6.3.1. Nous remplaçons le réseau filaire reliant ces machines par un réseau sans fil (Wi-Fi 802.11b) via une station de base. Nous n'avons pas pu mesurer la consommation de la batterie des PDA par manque de logiciels appropriés. En effet, la plupart des articles de la littérature présentant des mesures de consommation de la batterie des PDA utilisent des dispositifs électroniques afin de mesurer la consommation de la batterie, ce que ne pouvions faire.

La figure 6.10 présente l'interconnexion des deux machines. Le PC fixe est connecté au réseau filaire local du département Informatique de l'INT. Le PC portable est connecté à un réseau sans fil local via la station de base. Cette dernière sert aussi à faire le routage entre les deux réseaux.

Nous avons utilisé l'application *InternetTicket* (cf. chapitre 4) comme application de test. Tous les composants distants (7 composants dont 6 déconnectables) de cette application sont installés dans le PC fixe. Ce dernier a aussi le rôle de serveur de nommage. Une fois les composants distants installés sur le PC fixe, nous lançons sur le PC portable, d'abord le service de gestion du cache de DOMINT, ensuite le composant client. Au niveau du composant client, nous avons ajouté du code Java afin de générer une trace d'accès sur les services de l'application d'une manière aléatoire et indéfinie. Le temps séparant deux accès est 1 minute.

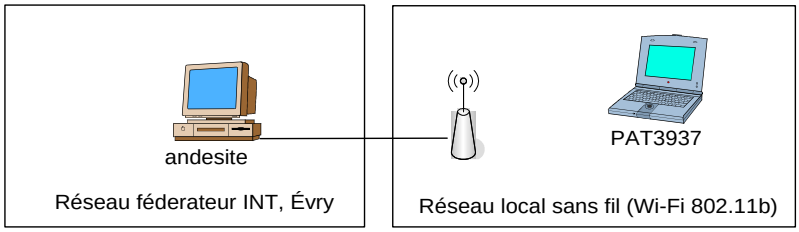


FIG. 6.10 – Machines utilisées dans les tests

Pour étudier l'impact de l'utilisation du service de gestion du cache sur la consommation de la batterie des terminaux mobiles, nous comparons la consommation de la batterie dans quatre scénarios : aucune application n'est lancée sur le PC portable, l'application *InternetTicket* est lancée sur le PC portable sans la gestion des déconnexions (sans le lancement du service de gestion du cache), et enfin, l'application *InternetTicket* est lancée sur le PC portable avec la gestion des déconnexions pour deux tailles du cache différentes. Cependant, comme précisé dans [106], l'analyse des caractéristiques des terminaux dans les environnements mobiles n'est pas une tâche triviale. En effet, il existe plusieurs facteurs pouvant influencer le comportement général des terminaux mobiles et du réseau sans fil (système d'exploitation, types de terminaux, température externe, nombre de terminaux présents, distances entre les terminaux et la station de base...). La consommation de la batterie peut être influencée par chacun de ces facteurs. Pour minimiser cette influence, nous fixons dans les quatre scénarios de tests les facteurs manipulables tels que le nombre de terminaux mobiles (un terminal), la distance entre le terminal est la station de base (environ deux mètres). Pour les facteurs non manipulables (température externe, nombre de processus s'exécutant sur le terminal mobile...), nous avons essayé d'avoir presque les mêmes conditions dans les quatre scénarios de tests.

Afin de simuler les variations de la bande passante, nous avons développé un simulateur qui permet de distribuer sur une période de temps donnée les différents modes de fonctionnement du terminal mobile (connecté, partiellement connecté et déconnecté). Cette distribution peut être paramétrée suivant les besoins. Par exemple, dans les tests que nous présentons dans cette section, sur 1 heure de fonctionnement, le simulateur partage ce temps en 70% connecté, 20% partiellement connecté et 10% déconnecté. Comme dans le simulateur utilisé dans SPREE [97], l'arriver des évènements des déconnexions suit la *loi de Poisson* et la duré de la période de fonctionnement (connecté, partiellement connecté et déconnecté) suit une distribution *Gaussienne*. Avant de lancer le service de gestion du cache et le composant client sur le PC portable, nous débranchons le câble d'alimentation électrique du chargeur de la batterie. Ensuite, chaque 6 minutes, nous récupérons le niveau de la batterie en utilisant le programme *BatteryMon* [11].

6.4.2 Résultats

La figure 6.11 présente les mesures de consommation de la batterie des quatre scénarios décrits dans la section 6.4.1. Dans le premier scénario, la batterie offre une autonomie de 132 minutes. Ce temps se réduit à 126 minutes dans le deuxième scénario, à 116 minutes dans le

troisième scénario et à 114 minutes dans le dernier scénario. Ces valeurs montrent que l'utilisation du service de gestion du cache dans le PC portable réduit l'autonomie de la batterie d'environ 8% par rapport à l'utilisation d'une application sans ce service. Nous expliquons cette perte par le surcoût en terme de charge engendré par le service de gestion du cache. La charge est essentiellement due au déploiement et au remplacement des services dans le cache.

Par ailleurs, la taille du cache utilisée dans le troisième scénario est de 700 Ko. Cette taille ne suffit qu'au déploiement de deux services (un nécessaire développeur et un nécessaire utilisateur). Les autres services sont déployés à la demande (remplacement du service nécessaire utilisateur). Par contre, la taille du cache utilisée dans le quatrième scénario est de 1 Mo. Cette taille permet la présence de trois services dans le cache, ce qui permet de réduire le nombre de remplacements. Ceci explique l'augmentation de l'autonomie de la batterie (2 minutes) dans le quatrième scénario par rapport au troisième scénario.

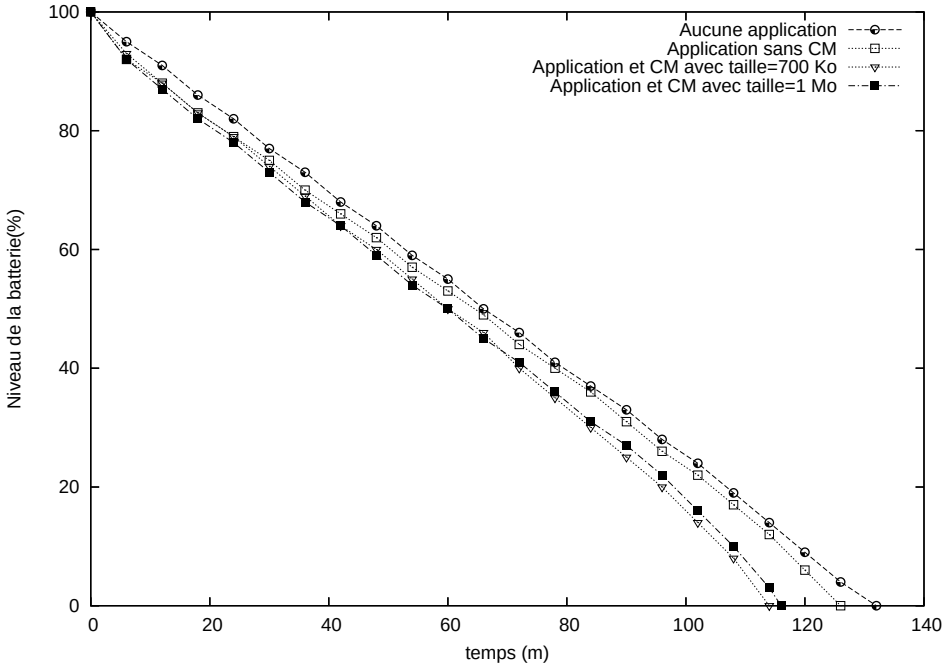


FIG. 6.11 – Mesures de la consommation de la batterie des terminaux mobiles

6.5 Synthèse

Ce chapitre a présenté les résultats de performances du service de gestion du cache de DOMINT. Dans un premier temps, nous avons étudié le temps de manipulation du graphe de dépendances dans les deux prototypes (prototype de base et prototype pour PDA). Nous avons constaté que le temps nécessaire pour la manipulation du graphe de dépendances est très faible en utilisant le prototype de base. Malheureusement, les mesures obtenues avec le prototype pour PDA sont beaucoup plus élevées par rapport aux mesures obtenues avec le prototype de base.

Ces résultats requirent donc d'envisager une recherche d'optimisation de la librairie *lightGXL*. Par ailleurs, nous avons constaté que la notion de service introduite dans MADA n'ajoute qu'un surcoût très faible au déploiement dans le service de gestion du cache (prototype de base et prototype pour PDA).

Nous avons ensuite évalué les stratégies de déploiement et de remplacement du cache présentées dans le chapitre 5. Nous avons d'abord montré l'intérêt de l'utilisation du processus *optProcess* dans le déploiement au pré-chargement. Nous avons ensuite démontré que l'utilisation du service comme granularité de gestion du cache offre une meilleure disponibilité par rapport à la granularité composant.

Enfin, nous avons étudié l'impact de l'utilisation de notre service de gestion du cache sur la consommation de la batterie des terminaux mobiles. Les résultats obtenus montrent que le service de gestion du cache de DOMINT engendre une perte d'autonomie de moins de 8% par rapport à l'utilisation d'une application sans ce service. Ces expériences permettent d'envisager des études plus fines des critères liés à la consommation de la batterie dans notre service afin d'offrir une meilleure autonomie de la batterie.

Troisième partie

Conclusions et perspectives

Conclusions

Les environnements mobiles sont caractérisés par des contextes d'exécution très variables. Cette variabilité est essentiellement liée à la faible autonomie et aux capacités de stockage réduites des terminaux mobiles ainsi qu'à la connectivité intermittente dans les réseaux sans fil. Malgré les déconnexions réseau volontaires ou involontaires, cette variabilité ne doit pas être vue comme un risque de défaillance puisqu'elle est une conséquence de la mobilité des utilisateurs. Par ailleurs, les applications et les intergiciels conçus pour les environnements fixes, d'une part, sont inadaptés aux caractéristiques des environnements mobiles, et d'autre part, ne permettent pas d'exploiter les nouvelles fonctionnalités offertes par ces environnements. En effet, l'exécution d'une application répartie dans les environnements fixes présuppose que les entités réparties sont reliées entre elles par une connexion réseau de bonne qualité. Un effort de recherche considérable est donc nécessaire afin de concevoir des applications et des intergiciels adaptés au problème des déconnexions.

La constitution de l'état de l'art nous a permis de mieux comprendre la problématique du point de vue réseau et intergiciel. Nous avons entamé cet état de l'art par l'étude de l'informatique mobile. En effet, la bonne compréhension du problème des déconnexions nécessite de bien comprendre les notions de base des technologies de communications sans fil. Nous avons ensuite présenté le fonctionnement des intergiciels (synchrone et asynchrone). L'étude de ces intergiciels nous a permis de constater qu'ils ne sont pas en adéquation avec les caractéristiques des environnements mobiles. Nous avons ensuite abordé les différentes techniques de séparation des préoccupations fonctionnelles et extrafonctionnelles. Nous avons identifié cinq techniques : modélisation de l'architecture, paradigme composant/conteneur, réflexivité, programmation orientée aspects et mécanisme d'interception. Enfin, nous avons étudié la problématique de gestion des déconnexions avec plus de détails. Nous avons proposé un ensemble de critères permettant de classer les différentes solutions de gestion des déconnexions existantes dans la littérature. Ces critères sont : entité manipulée par l'application (fichier, objet. . .), type de déconnexion traitée (volontaire et involontaire), type de cache (local et réparti), modes de fonctionnement pris en compte (connecté, partiellement connecté et déconnecté), stratégie d'adaptation (laissez-faire, transparence et collaboration), type d'adaptation (statique, dynamique et auto-adaptation), stratégies de déploiement et de remplacement du cache, présence d'un mécanisme de commutation entre l'entité distante et l'entité déconnectée, utilisation d'un modèle de conception orienté déconnexions, et prise en compte des dépendances entre entités de l'application. À notre connaissance, la classification ainsi que les critères de classification sont en eux-mêmes une contribution au domaine

de la gestion des déconnexions, et plus précisément, de la gestion du cache pour les déconnexions.

Contributions

La plupart des solutions que nous avons étudiées dans l'état de l'art sur la gestion des déconnexions se base sur le principe des opérations déconnectées. L'étude de ce principe nous a montré qu'il dégage quatre problématiques : la gestion du cache du terminal mobile, la détection des déconnexions, la commutation entre les composants locaux et les composants distants, et la gestion de la cohérence des différents composants. Nos travaux ne traitent que les problématiques gestion du cache du terminal mobile et commutation entre les composants locaux et les composants distants. Nous nous sommes fixé deux propriétés que doit respecter notre solution : collaboration entre l'application, l'intergiciel et l'utilisateur pour la gestion des déconnexions, et séparation des préoccupations fonctionnelles et extrafonctionnelles.

L'objectif de notre travail était de fournir une solution au problème des déconnexions des applications réparties à base de composants en environnements mobiles. Du travail de recherche présenté dans cette thèse émergent plusieurs contributions. Nos contributions s'étendent de la modélisation de l'application à la mise en œuvre d'un service intergiciel pour la gestion d'un cache de composants.

En plus de l'état de l'art sur la gestion du cache pour les déconnexions, nous résumons ces contributions comme suit.

Approche de conception MADA

Afin de prendre en compte la gestion des déconnexions dès la modélisation de l'application, nous avons proposé MADA, une approche de conception d'applications pouvant fonctionner en présence des déconnexions. Le but de l'approche MADA n'est pas de proposer une nouvelle méthode de conception d'applications réparties, mais plutôt de prendre en considération le problème de la gestion du cache pour les déconnexions dès la conception de l'application. Nous avons proposé dans MADA un profil de l'application basé sur un ensemble de méta-données. Ce profil permet de décrire toutes les entités de l'application : quelles sont les entités qui peuvent être dupliquées dans le terminal mobile (méta-donnée « déconnectabilité »), quelles sont les entités qui doivent être présentes sur le terminal mobile pour le fonctionnement en présence des déconnexions (méta-donnée « nécessité »), et quelle est la priorité entre ces entités (méta-donnée « priorité »). Nous avons défini la notion de service comme une composition logique de composants qui interagissent (interactions intra-services) entre eux pour accomplir une fonctionnalité de l'application. Ainsi, les fonctionnalités de l'application sont vues comme des services de l'application qui interagissent entre eux (interactions inter-services). Le profil de l'application est appliqué sur les services et les composants de l'application.

Nous avons ensuite proposé un profil UML pour la gestion des déconnexions nommé UML4CCMDisc qui permet d'étendre le processus de développement d'une application CCM

pour prendre en compte la gestion des déconnexions dès la modélisation de l'application. Enfin, nous avons proposé de modéliser les différentes interactions entre les entités de l'application (services et composants) dans un graphe de dépendances. Ce dernier fait apparaître explicitement les interactions intra-services et inter-services afin de les prendre en compte dans la gestion du cache. Nous avons proposé un ensemble de règles de propagation de la méta-donnée « nécessité » afin de rendre sa manipulation dynamique. Nous avons illustré l'approche MADA par une application de réservation et d'achat de billets d'avions par Internet (*InternetTicket*).

Contrairement aux solutions de gestion des déconnexions étudiées dans l'état de l'art, nous avons opté pour une adaptation « en collaboration » en introduisant l'utilisateur et l'architecte comme acteurs d'adaptation pour la gestion des déconnexions. La construction du profil de l'application par le développeur offre une meilleure attribution des méta-données aux entités de l'application, puisqu'il a la meilleure connaissance de la sémantique de l'application. L'utilisateur peut modifier ce profil et ainsi le personnaliser suivant ses besoins. Par ailleurs, l'approche MADA se distingue aussi des autres solutions (hormis les projets Seer et ACHILLES) par la prise en compte des dépendances entre les entités de l'application.

Service de gestion du cache de DOMINT

Nous avons proposé un service de gestion du cache pour le canevas logiciel DOMINT. Nous avons proposé dans ce service des stratégies de déploiement (quelles entités de l'application faut-il déployer localement, quand et pour quelle durée) et de remplacement (quelles entités doivent être supprimées lorsqu'il n'existe plus assez d'espace mémoire dans le cache). Ces stratégies se basent sur le profil de l'application et le graphe de dépendances proposés dans MADA. La stratégie de déploiement est assurée par une combinaison de trois étapes complémentaires de déploiement : pré-chargement, à la demande et à l'invocation. De même, la stratégie de remplacement est assurée par une combinaison de trois étapes complémentaires de remplacement : à la demande, explicite et périodique. Nous avons présenté la mise en œuvre Java de ce service pour des applications à base de composants CORBA. Nous avons utilisé le modèle de composant de Fractal pour la modélisation et la réalisation de ce service. Nous avons validé notre réalisation par deux prototypes : un prototype de base pour le fonctionnement du service de gestion du cache sur des machines standards (ordinateur personnel et ordinateur portable) et un prototype pour terminaux mobiles (assistants personnels numériques). Ces deux prototypes se basent sur le mécanisme de déploiement de la plateforme OpenCCM. Le portage de notre service sur des terminaux mobiles nous a donné un gain en taille mémoire de plus de 23%.

Modèle de conteneur pour la gestion des déconnexions

Nous avons proposé l'intégration de la gestion des déconnexions dans les conteneurs des composants. Nous avons proposé un modèle de conteneur de composant basé sur le principe des objets de contrôle. Le conteneur que nous proposons offre un lien entre un composant client, le service de gestion du cache, les composants locaux et les composants serveurs. Le conteneur permet de contrôler les requêtes du composant client, et de coordonner le déploiement et le

remplacement des composants dans le cache, suivant le niveau de disponibilité de la bande passante. En se basant sur le modèle du conteneur extensible (ECM), nous avons proposé une spécification et une réalisation Java/CCM de notre conteneur.

Comme décrit dans l'état de l'art, il existe peu de solutions qui abordent la gestion des déconnexions pour applications à base de composants. Notre solution se distingue des autres solutions par le fait qu'elle exploite le paradigme composant/conteneur pour la gestion des déconnexions. Ainsi, nous avons démontré avec MADA, le service de gestion du cache de DOMINT et notre modèle de conteneur, que le principe de séparation des préoccupations fonctionnelles et extrafonctionnelles dans le paradigme composant/conteneur convient aussi à la gestion des déconnexions.

Perspectives

Les perspectives de ce travail portent sur plusieurs points autant pour l'aspect architectural que pour l'aspect réalisation.

Concernant l'approche MADA, nous n'avons considéré la modélisation de la gestion des déconnexions que dans le modèle de composants CCM. Il serait intéressant d'étendre les différents mécanismes proposés dans MADA pour une modélisation au niveau PIM de MDA.

Un des mécanismes clés de l'approche MADA est la dynamicité de la méta-donnée « nécessité ». La dynamicité peut être aussi étendue à la méta-donnée « priorité ». En effet, rendre la méta-donnée « priorité » dynamique pourrait améliorer les performances des stratégies de déploiement et de remplacement.

En ce qui concerne le service de gestion du cache de DOMINT, la réalisation actuelle utilise tout le composant comme granularité de déploiement et de remplacement. Cependant, dans le modèle de composants CCM, l'implantation d'un composant peut être segmentée. Pour chaque segment, CCM génère un squelette et les segments sont activés indépendamment et possèdent un état. De plus, chaque segment est séparément identifié dans le système. Donc, il serait intéressant d'utiliser le segment comme granularité. Cependant, cette idée demande aussi à revoir l'approche MADA, en particulier, dans l'attribution des méta-données et la construction du graphe de dépendances. Nous avons déjà donné quelques pistes concernant l'utilisation du segment comme granularité dans [89].

Concernant la mise en œuvre de l'intégration de la gestion des déconnexions dans les conteneurs des composants, le travail qui reste à faire consiste d'une part à l'implantation du fonctionnement du conteneur en mode partiellement connecté, et d'autre part à trouver une solution au problème de redirection des requêtes au niveau du conteneur.

Un autre aspect que nous aimerions explorer est le portage de notre service de gestion du cache et le modèle du conteneur vers le modèle de composant EJB. Tout d'abord, ceci permettrait de démontrer la généricité des mécanismes proposés. Ensuite, les réalisations en source libre de EJB (par exemple, JBoss ou Jonas) sont intéressantes pour la mise en œuvre de notre conteneur.

À long terme, nous souhaitons réutiliser les différentes contributions de cette thèse dans le cas d'un cache réparti. L'utilisation d'un cache réparti pourrait améliorer la qualité de service de

l’application en présence des déconnexions. En effet, un terminal mobile peut être déconnecté d’un site qui héberge des composants qui n’ont pas été déployés sur le terminal mobile, mais peut être connecté à un autre site qui dispose d’une copie de ces composants. Cependant, la répartition du cache nécessite des algorithmes de partitionnement de l’ensemble du système en sous-ensembles. Nous comptons réutiliser et adapter des algorithmes de gestion de groupes proposés dans la littérature.

Index

Bibliographie

[1] Adaptative Platform for Proactive Reconfigurable Systems. `http://www-inf.int-evry/AMPROS`, 2004.

[2] B. Andersen, E. Jul, F. Moura, and V. Guedes. File System Support for Semiconnected Operation in AMIGOS. In *2nd USENIX Symposium on Mobile and Location-Independent Computing*, Santa Cruz, December 1994.

[3] E. Anquetil and F. Bouteruche. Conception d'un micro-éditeur d'encre électronique et embarquement d'un système de reconnaissance d'écriture manuscrite sur un téléphone mobile. In *Actes de la 1ère Conférénce Francophone Mobilité et Ubiquité*, Nice, France, June 2004. ACM Press.

[4] Apertos. The Reflective Object-Oriented Operating System. `http://www.csl.sony.co.jp/project/Apertos/`, 1997.

[5] ASTRID. ASTRID home page. `http://www.astrid.be/`, 2004.

[6] H. Atzmon, R. Friedman, and R. Vitenberg. Replacement Policies for a Distributed Object Caching Service. In *International Symposium on Distributed Objects and Applications*.

[7] T. Aubonnet. *Du Réseau Intelligent aux Nouvelles Générations de Réseaux : création et qualité de service*. PhD thesis, École Nationale Supérieure des Télécommunications, Paris, France, January 2002.

[8] A. Baggio. *Objets Distribués Adaptables pour Environnements Mobiles*. PhD thesis, Université Paris 6, France, June 1999.

[9] S. Baker. Keynote : A2A, B2B - now we need M2M (middleware to middleware) technology. In *3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.

[10] G. Barish and K. Obraczke. World Wide Web Caching : Trends and Techniques. *IEEE Communications Magazine*, 38(5) :178–184, May 2000.

[11] BatteryMon. PassMark home page. `http://www.passmark.com`, 2004.

[12] J. Baumann, F. Hohl, K. Rothermel, and M. Straber. Mole - Concepts of a mobile agent system. *World Wide Web*, 1(3) :123–137, March 1998.

[13] F. Bennett, T. Richardson, and A. Harter. Teleporting - Making Applications Mobile. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, December 1994.

[14] G. Bernard, J. Ben-othman, L. Bouganim, G. Canals, S. Chabridon, B. Defude, S. Ferrié, J. Gançarski, R. Guerraoui, P. Molli, P. Pucheral, C. Roncancio, P. Serrano-Alvarado, and P. Valduriez. Mobile databases : a selection of open issues and research directions. *ACM SIGMOD Record*, 33(2) :78–83, June 2004.

[15] P. Bernstein. Middleware : a model for distributed system services. *Communications of the ACM*, 39(2) :86–98, Fubruary 1996.

[16] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer Society Press*, 32(7) :38–45, July 1999.

[17] G. Bieber and J. Carpenter. Introduction to Service-Oriented Programming. Technical report, `http://www.openwings.org`, December 2001.

[18] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1) :39–59, February 1984.

[19] Bluetooth home page. The Bluetooth specification. `http://www.bluetooth.org/spec/`, 2004.

[20] S. Bouchenak. *Mobilité et persistance des applications dans l'environnement Java*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, October 2001.

[21] D. Box. *Essential COM*. Addison Wesley Professional, 1997.

[22] E. Bruneton. *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 0ctober 2001.

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley and Sons, 1996.

[24] S. Campadello and K. Raatikainen. Agents in Personal Mobility. In *First International Workshop on Mobile Agents For Telecommunication Applications*, pages 359–374, Ottawa, Canada, October 1999.

[25] L. Capra, G. Blair, , C. Mascolo, W. Emmerich, and P. Grace. Exploiting Reflection in Mobile Computing Middleware. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4) :34–44, October 2002.

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

[27] CARISM. Composants Adaptables et Reconfigurables pour Intergiciels et Services Mobiles. `http://www.picolibre.org/projects/carism/`, 2004.

[28] U. Çetintemel, P. Keleher, B. Bhattacharjee, and M. Franklin. Deno : A Decentralized, Peer-to-Peer Object-Replication System for Weekly Connected Environments. 52(7) :943–959, July 2003.

[29] CCM. CORBA Component Model. OMG Document formal/02-06-65, Version 3.0, `http://www.omg.org`, June 2002.

[30] L. Chateigner. *Intergiciel extensible à base de composants adaptables pour l'informatique mobile : réplication optimiste et réconciliation.* PhD thesis, Institut National des Télécommunications, Évry, France, 2005. En cours de préparation.

[31] L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical report, HP Labs, Palo Alto, November 1998.

[32] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *2nd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 1–23, New York, USA, April 2000.

[33] COACH. Component Based Open Source Architecture for Distributed Telecom Applications. `http://coach.objectweb.org`, 2004.

[34] 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 technique, Institut National des Télécommunications, Évry, France, May 2003.

[35] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Domint : Weak Connectivity and Disconnected CORBA Objects on Hand-Held Devices. Technical report, Institut National des Télécommunications, Évry, France, January 2003.

[36] D. Conan, S. Chabridon, O. Villin, G. Bernard, A. Kotchanov, and T. Saridakis. Handling Network Roaming and Long Disconnections at Middleware Level. In *Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices (in conjunction with EDOC'2002)*, Lausanne, Switzerland, September 2002.

[37] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET) : Routing Protocol Performance Issues and Evaluation Considerations. Network Working Group, RFC : 2501, January 1999. `http://www.ietf.org/rfc/rfc2501.txt`.

[38] CWM. Common Warehouse Metamodel Specification. Omg document ad/01-02-01, Object Management Group, February 2001. Version 1.0.

[39] DOMINT. DOMINT home page. `http://picolibre.int-evry.fr`, 2005.

[40] D. Dwyer and V. Bharghavan. A mobility-aware file system for partially connected operation. *CM SIGOPS Operating Systems Review*, 31(1) :24–30, January 1997.

[41] EDOC. UML Profile for Enterprise Distributed Object Computing Specification. Omg document ptc/02-02-05, Object Management Group, February 2002.

[42] EJCCM. EJCCM home page. `http://www.cpi.com/ejccm`, 2004.

[43] T. Ernst. *Le Support des Réseaux Mobiles dans IPv6*. PhD thesis, Université Joseph Fourier, Grenoble, France, October 2001.

[44] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2) :114–131, June 2003.

[45] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies : potential and performance. *ACM SIGMETRICS Performance Evaluation Review*, 27(1) :178–187, June 1999.

[46] J. Ferber. Computational reflection in class based object-oriented languages. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 317–326, New Orleans, Louisiana, USA, October 1989.

[47] F. Flinn, D. Narayanan, and s. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. In *Eighth Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[48] J. Flinn, S. Park, and M. Satyanarayanan. Balancing Performance, Energy, and Quality in Pervasive Computing. In *International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[49] G. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4) :38–47, April 1994.

[50] Fractal. Fractal home page. `http://fractal.objectweb.org`, 2004.

[51] M. J. Franklin. Transactional Client-Server Cache Consistency : Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3) :315–363, september 1997.

[52] K. Froese and R. Bunt. Cache Management for Mobile File Service. *The Computer Journal*, 42(6) :442–454, 1999.

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

[54] M. Gardner, C. Gibbs, and T. van Do. Is the Future Really Always-on ? From Always-on Networks to Always-on Sessions. *BT Technology Journal*, 20(1) :91–102, January 2002.

[55] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1) :80–112, 1985.

[56] C. Gransart, J. Geib, and P. Merle. *CORBA : des concepts à la pratique*. DUNOD, 1999.

[57] S. Gurun, C. Krintz, and R. Wolski. NWSLite : A Light-Weight Prediction Utility for Mobile Devices. In *International Conference on Mobile Systems, Applications, and Services*, pages 2–11, Boston, Massachusetts, USA, June 2004.

[58] R. Harbus. Dynamic process migration : To migrate or not to migrate. Technical report csri-42, University of Toronto, Canada, July 1986.

[59] Hewlett-Packard. HP home page. `http://www.hp.com`, 2004.

[60] R. Holt, A. Schurr, S. Elliott, and A. Winter. GXL home page. `http://www.gupro.de/GXL`, 2002.

[61] B. Housel, G. Samaras, and D. Lindquist. WebExpress : a client/intercept based system for optimizing Web browsing in a wireless environment. *ACM Mobile Networks and Applications*, 3(4) :419–431, December 1998.

[62] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1) :51–81, February 1988.

[63] C. Hérault and S. Lecomte. Adaptabilité des Services Techniques dans un Modèle à Composants. In *Conférence Française sur les systèmes d'exploitation, CFSE*, pages 488–499, La Colle sur Loup, France, october 2003.

[64] L. Huston and P. Honeyman. Disconnected Operation for AFS. In *First USENIX Symposium on Mobile and Location-Independent Computing*, pages 1–10, Capbridge, MA, April 1993.

[65] L. Huston and P. Honeyman. Partially Connected Operation. In *2nd Symposium on Mobile and Location-Independent Computing*, pages 91–98, Ann Arbor, Michigan, USA, April 1995.

[66] IETF. Session Initiation Protocol. `http://www.ietf.org`, 2004.

[67] Intel. Intel home page. `http://www.intel.com`, 2004.

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

[69] J2EE. Java 2 Platform Enterprise Edition Specification. `http://java.sun.com/j2ee/`, 2002. Version 1.4.

[70] C. Janneteau and H. Lach. *Wireless IP NEtwork as a Generic platform for Location Aware Service Support*. WINEGLASS IST Project, `http://wingelass.tilab.com`, July 2000. Deliverable N°4.

[71] JavaSpaces. JavaSpace Service Specification. `http://www.sun.com`, 2003.

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

[73] JMS. Java Messaging Service Specification. `http://java.sun.com/jms/`, April 2002. Version 1.1.

[74] JNDI. Java Naming and Directory Interface. Technical report, Sun Microsystem, 1999. Version 1.2.

[75] Jonathan. Jonathan home page. `http://jonathan.objectweb.org`, 2004.

[76] A. Joseph, J. Tauber, and M. Kaashoek. Rover : A Toolkit for Mobile Information Access. In *16th Symposium on Operating Systems Principles*, Copper Mountain, Colorado, USA, December 1995.

[77] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *ACM Transactions on Computers*, 46(3) :337–352, 1997.

[78] K2CCM. K2 CCM home page. `http://www.icmgworld.com`, 2004.

[79] M. Kaddour and L. Pautet. A Middleware for Supporting Disconnections and Multi-Network Access in Mobile Environments. In *Proceedings of the 2nd Conference On Pervasive Computing (PerCom)*, Orlando, Florida, USA, March 2004.

[80] D. Katsaros and Y. Manolopoulos. Web Caching in Broadcast Mobile Wireless Environments. *IEEE Internet Computing*, 8(3) :37–45, May 2004.

[81] R. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Computing Magazine*, 1(1) :6–17, 1994.

[82] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 327–353, Budapest, Hungary, June 2001.

[83] G. Kiczales, J. Lamping, M. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Jyväskylä, Finland, June 1997.

[84] G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[85] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *13th ACM Symposium on Operating Systems Principles*, pages 213–225, Pacific Grove, USA, May 1991.

[86] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1) :3–25, February 1992.

[87] G. Kortuem, S. Fickas, and Z. Segall. On-Demand Delivery of Software in Mobile Environments. In *Nomadic Computing Workshop, 11th International Parallel Processing Symposium*, April 1-5 1997.

[88] N. Kouici, L. Chateigner, D. Conan, S. Chabridon, and G. Bernard. Gestion de déconnexion pour applications réparties à base de composants dans un environnement mobile. In *7th International Symposium on Programming and Systems*, Algiers, Algeria, May 2005.

[89] N. Kouici, D. Conan, and G. Bernard. Adaptation des applications réparties à base de composants aux terminaux mobiles en environnement sans fil. In *Manifestation des jeunes chercheurs STIC, MAJECSTIC*, Marseille, France, October 2003.

[90] 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*, pages 358–364, Las Vegas, Nevada, USA, June 2003.

[91] S. Krakowiak. Patrons et canevas pour l'intergiciel. Présentation, Quatrième École d'été sur les Intergiciels et sur la Construction d'Applications Réparties, Autrans, France, August 2003.

[92] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *The USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 1997.

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

[94] G. Kuenning. Design of the SEER predictive caching schema. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, USA, 1994.

[95] G. Kuenning. *Seer : Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, University of California, Los Angeles, USA, May 1997.

[96] G. Kuenning and G. Popek. Automated Hoarding for Mobile Computers. In *The 16th ACM Symposium on Operating Systems Principles*, pages 264–275, Saint Malo, France, October 1997.

[97] K. Kvilekval and A. Singh. SPREE : Object Prefetching for Mobile Computers. In *International Symposium on Distributed Objects and Applications, DOA*, pages 1340–1357, Larnaca, Cyprus, October 2004.

[98] P. Laumay. *Configuration et déploiement d'intergiciel asynchrone sur système hétérogène à grande échelle*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, March 2004.

[99] T. Lehman, S. Laughry, and P. Wyckoff. TSpaces : The next Wave. In *32nd International Conference on System Sciences*, Maui, Hawaii, January 1999.

[100] A. Lelah and K. Grein-Cochard. Une définition à la mobilité. In *Actes de la 1ère Conférénce Francophone Mobilité et Ubiquité*, pages 13–16, Nice, France, June 2004.

[101] N. Lynch. Supporting Disconnected Operation in Mobile CORBA. Master's thesis, Trinity College Dublin, Ireland, September 1999.

[102] C. Maltzahn, K. Richardson, D. Grunwald, and J. Martin. On Bandwidth Smoothing. In *4th International Web Caching Workshop*, San Diego, California, USA, April 1999.

[103] D. Mandato, E. Kovacs, F. Hohl, and H. Amir-Alikhani. CAMP : a Context-Aware Mobile Portal. *IEEE Communications Magazine*, 40(1) :90–97, January 2002.

[104] P. Maniatis, M. Roussopoulos, E. Swierk, G. Lai, K.and Appenzeller, X. Zhao, and M. Baker. The Mobile People Architecture. *ACM Mobile Computing and Communications Review*, 3(3) :36–42, July 1999.

[105] V. Marangozova. *Duplication et cohérence configurables dans les applications réparties à base de composants*. PhD thesis, Université Joseph Fourier, Grenoble, France, July 2003.

[106] C. Marchand. *Mise au point d'algorithmes répartis dans un environnement fortement variable, et expérimentation dans le contexte des pico-réseaux*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, December 2004.

[107] C. Mascolo, L. Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced Lectures on Networking*, Pisa, Italy, 2002.

[108] MDA. MDA Guide. Omg document formal/2003-06-01, Object Management Group, June 2003. Version 1.0.1.

[109] MicoCCM. MicoCCM home page. `http://www.fpx.de/MicoCCM/`, 2004.

[110] S. Microsystems. Object Serialization. `http://java.sun.com/j2se/`, 2004.

[111] J. Miller and J. Mukerji. Model Driven Architecture (MDA). Omg document ormsc/2001-07-01, Object Management Group, July 2001. White Paper.

[112] MOF. Meta-Object Facility Specification. Omg document formal/2002-04-03, Object Management Group, April 2002. Version 1.4.

[113] R. Monson-Haefel. *Enterprise JavaBeans*. O'REILLY, 2004.

[114] MQSeries. MQSeries home page. `http://www.mqseries.net`, 2004.

[115] L. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Carnegie Mellon, USA, September 1996.

[116] B. Myers. Using Handhelds and PCs Together. *Communications of the ACM*, 44(11) :34–41, November 2001.

[117] nanoxml. nanoxml home page. `http://nanoxml.cyberelf.be`, 2003.

[118] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *16th ACM Symposium on Operating System Principles*, Saint-Malo, France, October 1997.

[119] ObjectWeb. ObjectWeb home page. `http://www.objectweb.org`, 2004.

[120] OMG. The Common Object Request Broker - Architecture and Specifications. Revision 2.6. OMG Document formal/01-12-01, http://www.omg.org, year = "December 2001".

[121] OMG. Discussion of the Object Management Architecture : OMA Guide. Omg document formal/00-06-41, http://www.omg.org, January 1997.

[122] OMG. Portable Interceptor. Omg document ptc/2001-03-04, Object Management Group, March 2001. Published Draft.

[123] OMG. UML Profile for CORBA Specification. Omg document formal/02-04-01, Object Management Group, April 2002. Version 1.0.

[124] OMG. Object Management Group home page. http://www.omg.org, 2005.

[125] OMG. UML Profile for CORBA Components. Omg specification ptc/2004-11-05, Object Management Group, January 2005.

[126] OMG. Wireless Access and Terminal Mobility in CORBA. OMG Document Formal/04-04-02, http://www.omg.org, April 2004.

[127] OpenCCM. OpenCCM home page. http://openccm.objectweb.org, 2005.

[128] OSMOSE. Open Source Middleware for Open Systems in Europe. http://www.itea-osmose.org/, 2004.

[129] V. Padmanabhan and J. Mogul. Using predictive prefetching to improve World Wide Web latency. *ACM SIGCOMM Computer Communication Review*, 26(3) :22–36, July 1996.

[130] M. Pellegrini, O. Potonnièe, R. Marvie, S. Jean, and M. Riveill. CESURE : une plate-forme d'applications adaptables et sécurisées pour usagers mobiles. Technical report, CESURE RNRT Project, http://www.inria.fr/recherche, 2000.

[131] Perseus. Perseus home page. http://perseus.objectweb.org, 2003.

[132] N. Pessemier, L. Seinturier, L. Duchien, and O. Barais. Une extension de Fractal pour l'AOP. In *Première journée Francophone sur le Développement de Logiciels par Aspects, JFDLPA*, Paris, France, September 2004.

[133] K. Petersen, D. Terry, M. Theimer, A. Demers, and M. Spreitzer. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating Systems Principles*.

[134] K. Petersen, D. Terry, M. Theimer, A. Demers, M. Spreitzer, and B. Welch. The Bayou Architecture : Support for Data Sharing among Mobile Users. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, CA, USA, December 1994.

[135] S. Pierre. *Réseaux et systèmes informatiques mobiles, fondements, architectures et applications*. Presses Internationales Polytechnique, 2003.

[136] E. Pitoura and B. Bhargava. Building Information Systems for Mobile Environments. In *Third ACM International Conference on Information and knowledge Management*, pages 371–378, Gaithersburg, Maryland, USA, November 1994.

[137] Qedo. Qedo home page. http://qedo.berlios.de, 2004.

[138] A. Redkar, C. Walzer, and S. Boyd. *Pro Msmq : Microsoft Message Queue Programming*. Apress, 2004.

[139] M. Rodrig and A. LaMarca. Oasis : An Architecture for Simplified Data Management and Disconnected Operation. *Personal and Ubiquitous Computing Journal*, 9(2), March 2005.

[140] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning. The Remote Processing Framework for Portable Computer Power Saving. In *ACM Symposium on Applied Computing*, pages 365–372, San Antonio, Texas, USA, February 1999.

[141] R. Ruggaber, J. Seitz, and M. Knapp. π^2 - A Generic Proxy Platform for Wireless Access and Mobility. In *19th ACM Symposium on Principles of Distributed Computing*, pages 191–198, Portland, Oregon, USA, July July 2000.

[142] N. Sabri. *Une Architecture à base de Composants CORBA pour des Services Personnalisés*. PhD thesis, Université d'Évry Val d'Essonne, Évry, France, June 2003.

[143] SARDES. SARDES home page. `http://sci-serv.inrialpes.fr`, 2004.

[144] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *15th Symposium on Principles of Distributed Computing*, pages 1–7, May 1996.

[145] M. Satyanarayanan. Pervasive Computing : Vision and Challenges. *IEEE Personal Communications*, 8(4) :10–17, August 2001.

[146] M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, (2) :85–124, May 2002.

[147] M. Satyanarayanan. From the Editor in Chief : The Many Faces of Adaptation. *IEEE Pervasive Computing*, 3(3) :4–5, July 2004.

[148] M. Satyanarayanan, J. Kistler, L. Mummert, M. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *First USENIX Symposium on Mobile and Location-Independent Computing*, pages 11–28, Capbridge, MA, April 1993.

[149] ScanSoft. ScanSoft home page. `http://www.scansoft.com`, 2004.

[150] D. Schmidt and F. Kuhns. An Overview of the Real-Time CORBA Specification. *IEEE Computer Society Press*, 33(6) :56–63, June 2000.

[151] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture : Pattern for Concurrent and Networked Objects*. John Wiley and Sons, 2000. volume 2.

[152] R. Sessions. *COM+ and the Battle for the Middle Tier*. Wiley, 2000.

[153] B. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, USA, 1982.

[154] Sun. UML Profile for EJB Specification. Public draft, Sun Microsystem, June 2001.

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

[156] A. Tanenbaum. *Réseaux*. Pearson Education, 4 edition, 2003.

[157] X. N. W. Team. Nomadicity in the NII. Technical report, Cross-Industry Working Team, 1995.

[158] L. Temal and D. Conan. Détections de défaillances, de connectivité et de déconnexions. In *Actes de la 1ère Conférénce Francophone Mobilité et Ubiquité*, pages 90–97, Nice, France, June 2004. ACM International Conference Proceedings Series.

[159] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou : A Weakly Connected Replicated Storage System. In *15th ACM Symposium on Operating Systems Principles*, pages 172–182, Copper Mountain, Colorado, USA, December 1995.

[160] T. Thai and H. Lam. *.NET Framework Essentials*. O'REILLY, 2001.

[161] D. Thanh, S. Steensen, and J. Audestad. Mobility Management and Roaming with Mobile Agents. In *Proceedings of the IFIP-TC6/European Commission International Workshop on Mobile and Wireless Communication Networks*, pages 123–137, Paris, France, May 2000.

[162] TRAVLER home page. `http://fmg-www.cs.ucla.edu/travler98/`, 1998.

[163] N. Tripathi, J. Reed, and H. VanLandinoham. Handoff in Cellular Systems. *IEEE Personal Communications*, 5(6) :26–37, December 1998.

[164] UML. UML Home Page. http ://www.uml.org, 2005.

[165] M. Vadet. *Un Modèle de Services Logiciels pour la Spécialisation des Intergiciels à Composants*. PhD thesis, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, Lille, France, November 2004.

[166] M. Vadet and P. Merle. Les conteneurs ouverts dans les plates-formes à composants. In *Proc. Journée Théme Émergent Composants*, Besançon, France, october 2001.

[167] O. Villin. *Gestion de la qualité de service de bout en bout dans les systèmes répartis : approche gestion des ressources* . PhD thesis, Université d'Évry Val d'Essonne, Évry, France, April 2002.

[168] VisiBroker. VisiBroker home page. `http://www.borland.com/visibroker/`, 2005.

[169] VIVIAN. Opening Mobile Platforms for the Development of Component-based Applications. `http://www-nrc.nokia.com/Vivian/`, 2001.

[170] M. Völter. Server-Side Components - A Pattern Language. In *Sixth European Conference On Pattern Languages of Programs*, Irsee, Germany, 4-8 July 2001.

[171] WAPForum. WAP home page. `http://www.wapforum.org/`, 2003.

[172] J. Warmer and A. Kleppe. *The Object Constraint Language : Getting Your Models Ready for Mda*. Addison-Wesley, 2004.

[173] S. Williams, M. Abrams, C. Standridge, A. G., and E. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Procedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[174] S. Wilson and J. Kesselman. *Java Platform Performance : Strategies and Tactics*. Sun microsystems, 2000.

[175] XMI. XML Metadata Interchange Specification. Omg document formal/02-01-01, Object Management Group, January 2002. Version 1.2.

[176] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *The First International Conference on Mobile Systems, Applications, and Services*, pages 129–144, San Francisco, CA, USA, 5-8 May 2003.