Université d'Évry-Val d'Essonne
Institut National des Télécommunications

Rapport de Stage
DEA d'Informatique

MOBILITÉ DANS LES SYSTÈMES RÉPARTIS : LE DÉPLOIEMENT SUR LE TERMINAL MOBILE

Nabil KOUICI

Responsable de DEA : Jean-Marc DELOSME
Responsable de stage : Denis CONAN

Juillet 2002

Ce stage de DEA a été réalisé au sein du laboratoire **Systèmes Répartis** du département **Informatique** de l'**Institut National des Télécommunications**

Remerciements

Je tiens à remercier, Guy Bernard pour m'avoir accueilli dans l'équipe Système Répartie de l'Institut National des Télécommunications. La grande qualité de ses enseignements en DEA m'a convaincu de me consacrer aux systèmes répartis.

Je remercie Denis Conan pour avoir proposé ce stage et m'avoir encadré pendant ces cinq mois. Qu'il soit remercié pour son contact chaleureux, ses conseils et encouragements, son soutien permanent et pour la liberté de recherche qu'il a bien voulu me laisser. Je le remercie de m'avoir accordé le temps nécessaire pour s'entretenir avec moi et m'orienté vers le bon chemin dans la recherche.

Merci à tous ceux qui de près ou de loin m'ont aidé pendant cette période. En particulier, les personnes du département informatique et les étudiants du DEA d'Évry en stage à l'INT, avec qui les discussions sur les sujets de recherche de chacun ont apporté des idées et des points de vue différents.

Un grand MERCI aux thésards Érik Putrycz, Olivier Villin et Victor Budau dont l'aide a été toute aussi précieuse.

Je remercie enfin, mes parents, mes frères, mes sœurs et mes amis qui m'ont soutenu tout au long de ce stage.

Table des matières

1 Introduction **1**

2 Problématique et objectifs du stage **3**
2.1 Sujet du stage . . . 3
2.2 Résumé du contexte du stage . . . 4

3 État de l'art sur la mobilité **5**
3.1 Paradigmes des modèles client-seveur mobiles . . . 5
3.1.1 Adaptation des applications mobiles . . . 5
3.1.2 Extension des modèles client-serveur . . . 6
3.2 Étude de projets . . . 6
3.2.1 Coda . . . 7
3.2.1.1 Opération déconnectée et mode faiblement connecté . . . 7
3.2.1.2 Gestion du cache du client mobile . . . 7
3.2.1.3 Détection et résolution des conflits . . . 8
3.2.2 Odyssey . . . 8
3.2.2.1 Agilité . . . 9
3.2.2.2 Description d'Odyssey . . . 9
3.2.2.3 Architecture . . . 9
3.2.3 Bayou . . . 10
3.2.3.1 Architecture . . . 10
3.2.4 Rover . . . 11
3.2.4.1 Objets dynamiques ré-adressables et appel de procédure à distance non-bloquant . . . 11
3.2.4.2 Architecture . . . 12
3.2.5 Autre projets . . . 13
3.3 Déploiement . . . 13
3.3.1 Pas de cache . . . 13
3.3.2 Mise en cache systématique . . . 14
3.3.3 Mise en cache à la demande . . . 14
3.3.4 Pré-chargement . . . 14
3.4 Conclusion . . . 15

4 Architecture de Domint **17**
4.1 Les intercepteurs portables . . . 17
4.1.1 Intercepteur de requêtes . . . 17
4.1.1.1 Intercepteur côté client et côté serveur . . . 18
4.1.1.2 Informations sur la requête . . . 18

4.1.2 Intercepteur de références d'objets . . . 19
4.2 Plate-forme Domint . . . 19
4.2.1 Architecture . . . 20
4.2.2 Quelques détails sur les objets de Domint . . . 21
4.2.2.1 Gestionnaire d'objets déconnectés . . . 22
4.2.2.2 Gestionnaire de ressources . . . 22
4.2.2.3 Gestionnaire de connectivité . . . 22
4.2.2.4 Gestionnaire du journal d'opérations . . . 23

5 Protocole de déploiement d'objets **25**
5.1 Concept de criticité . . . 25
5.1.1 Introduction du concept . . . 25
5.1.2 Objets critiques et Objets non critiques . . . 27
5.1.3 Exemple de partitionnement des objets . . . 28
5.2 Déploiement des objets critiques et non critiques . . . 30
5.2.1 Déploiement suivant la criticité des objets . . . 30
5.2.2 Problème de connectivité . . . 30
5.3 Étapes de conception . . . 31
5.4 Déploiement dans Domint . . . 32
5.4.1 Architecture . . . 32
5.4.2 Gestion du cache . . . 34
5.4.2.1 Algorithmes de gestion du cache . . . 34
5.4.2.2 Remplacement des objets non critiques . . . 35
5.5 Discussion . . . 37

6 Réalisation **39**
6.1 Prototype de réalisation . . . 39
6.2 Application exemple . . . 40
6.3 Mise en œuvre . . . 40
6.4 Travail en cours . . . 42

7 Conclusion **43**
7.1 Perspectives . . . 44

Table des figures

2.1 Positionnement du stage 4

3.1 Les stratégies d'adaptation 6
3.2 Les états de Venus dans Coda 8
3.3 L'architecture d'Odyssey 10
3.4 L'architecture de Bayou 11
3.5 L'architecture de Rover 12

4.1 Les points d'interception 18
4.2 Le comportement de Domint en mode fortement connecté 20
4.3 Le comportement de Domint en mode partiellement connecté et déconnecté 21
4.4 L'hystérésis pour la gestion de la conectivité 23

5.1 Exemple de création des objets déconnectés 26
5.2 La criticité des objets 27
5.3 La propagation de la criticité entre classes 28
5.4 Le déploiement d'objets 30
5.5 Les étapes de conception 31
5.6 Le déploiement dans Domint au lancement de l'application 32
5.7 L'algorithme de Déploiement des objets critiques 33
5.8 L'interface de l'objet déconnecté 34
5.9 La gestion du cache dans Domint 35
5.10 La création d'un objet intermédiaire entre PI et RM 37
5.11 L'algorithme de gestion des objets non critiques 38

6.1 Le prototype d'implémentation 39
6.2 L'application de messagerie électronique 40

Liste des tableaux

3.1 La méthode de déploiement dans les différents systèmes 15

5.1 Un exemple de partitionnement d'objets pour l'application messagerie électronique . 29
5.2 Un exemple de séquences d'invocations des objet *DossierModel* et *CarnetAdresse-Personnel* 36

Chapitre 1

Introduction

Ces dernières années ont été marquées par une forte évolution des équipements utilisés dans les environnements répartis. Nous sommes successivement passés de réseaux locaux à des réseaux à grande échelle (Internet), puis à des réseaux sans fil inter-connectant des machines mobiles comme des téléphones portables ou des assistants personnels numériques (PDA). Cette évolution a abouti à la définition d'une nouvelle technologie, qui se base sur l'infrastructure des réseaux mobiles. Cette technologie est l'informatique mobile dans laquelle l'utilisateur peut continuer d'accéder à l'information fournie par une infrastructure distribuée, sans tenir compte de l'emplacement de l'utilisateur. Les techniques traditionnelles d'accès à l'information distribuée se basent sur deux hypothèses. Premièrement, la localisation des utilisateurs dans le système est stable. Deuxièmement, la connexion réseau est plutôt stable en terme de performance et de fiabilité. Dans les environnements mobiles, la validité de ces deux hypothèses est très rare. Le problème qui se pose n'est pas dans le matériel utilisé mais dans le logiciel qui utilise ce matériel, pour le développement des applications qui travaille en mode mobile.

L'informatique mobile se distingue des méthodes classiques d'accès à l'information par quatre contraintes :

- Par rapport aux éléments statiques, les éléments mobiles sont plus pauvres en ressources telles que la capacité de la mémoire et la vitesse d'exécution.
- Les éléments mobiles sont très exposés aux vols et l'endommagement.
- La connexion sans fil est très variable en terme de performance et de fiabilité.
- Les éléments mobiles possèdent des sources d'énergie finies.

Les machines mobiles devenant extrêmement courantes, leur utilisation doit devenir aussi naturelle que possible. En dépit de tous les problèmes mentionnés ci-dessus, un utilisateur mobile souhaite se déplacer librement et continuer à travailler le plus normalement possible avec son terminal mobile. Il est donc souhaitable de fournir une continuité de service malgré les déconnexions et les perturbations du réseau sans fil. Le besoin de continuer à travailler dans un environnement mobile soulève de nombreux problèmes. Premièrement, il est indispensable de gérer la disponibilité des données en présence de déconnexions. L'approche visant à résoudre ce problème passe par une réplication le plus souvent locale des données sur le terminal mobile. Deuxièment, il est également indispensable que le système et les applications soient réactifs aux changements de l'environnement mobile.

Le stage de DEA est effectué au sein de l'équipe MARGE (*Middleware pour Applications Réparties à Grande Échelle*) de l'*Institut National des Télécommunications* (INT) où des mécanismes ont été proposés pour répondre aux besoins de disponibilité des données et de réactivité aux changements de l'environnement mobile. Ces mécanismes sont mis en œuvre au sein d'une

plate-forme CORBA (*Objects Request Broker Architecture*) dénommée Domint. Les mécanismes principaux proposés par Domint sont, d'une part, le support de la réplication des objets, et d'autre part, le support des déconnexion volontaires et involontaires en utilisant des mécanismes CORBA tels que les objets par valeur et les intercepteurs portables.

Ce document présente l'ensemble du travail et des résultats obtenus au cours de ce stage. Il s'articule autour du plan suivant. Dans le chapitre 2, nous examinons la problématique et les objectifs du stage. Ensuite, le chapitre 3 présente un état de l'art sur les environnements mobiles où nous étudierons quelques grands projets qui traitent le problème de la mobilité des applications. Dans le chapitre 4 nous donnons une courte présentation de la plate-forme Domint. Ensuite, nous présentons l'approche de déploiement et de gestion du cache que nous avons intégrées dans Domint. Enfin, pour démontrer la faisabilité de notre approche, le chapitre 6 présente la réalisation de notre approche de déploiement et de gestion du cache sur une application de messagerie électronique.

Chapitre 2

Problématique et objectifs du stage

L'informatique mobile peut désigner la mobilité matérielle ou la mobilité logicielle. La mobilité matérielle est le déplacement physique d'un terminal tel qu'un ordinateur portable, un téléphone ou un assistant personnel numérique. La mobilité logicielle est le déplacement d'une entité logicielle entre deux terminaux physiques. L'entité logicielle mobile est alors appelée composant, agent, calcul ou application mobile. Dans la suite de notre travail, nous ne nous intéressons qu'à la mobilité logicielle que nous désignons par mobilité des applications. Dans ce chapitre, nous présentons dans un premier temps le sujet du stage. Ensuite, nous récapitulons le contexte de notre travail.

2.1 Sujet du stage

Ce stage de DEA s'insère dans une action plus globale réalisée dans le cadre du projet ITEA VIVIAN (*http ://www-nrc.nokia.com/Vivian*) concernant l'ouverture des plates-formes mobiles pour le développement d'applications à base de composants. La plate-forme VIVIAN repose sur la spécification CORBA de l'OMG (*Objects Management Group*) et est conçue pour être extensible et évolutive afin de permettre la prise en compte des besoins variés des applications.

L'équipe Systèmes Répartis de INT axe ses travaux sur le fonctionnement des applications en mode déconnecté. L'objectif est de fournir un support logiciel pour adapter des applications client-serveur existantes au fonctionnement déconnecté du terminal dans les réseaux sans fil. 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.

D'une manière générale, le projet est divisé en trois étapes, représentées dans la figure 2.1. La première étape consiste à traiter le problème du déploiement des objets dans le terminal mobile en répondant à quelques questions telles que :

- Quels sont les critères (inter-dépendance, taille, nombre) pour le choix des objets qu'il faut déployer localement sur le terminal mobile ?
- Comment définir cet ensemble d'objets locaux dits déconnectés ?
- Quand déployer cet ensemble d'objets ?

La deuxième partie du projet de l'INT consiste à traiter le problème de la création des objets déconnectés et la manière dont le client mobile travaille dans les différents modes de connectivités. La troisième partie du projet de l'INT traite de problème de la cohérence et de la réconciliation entre

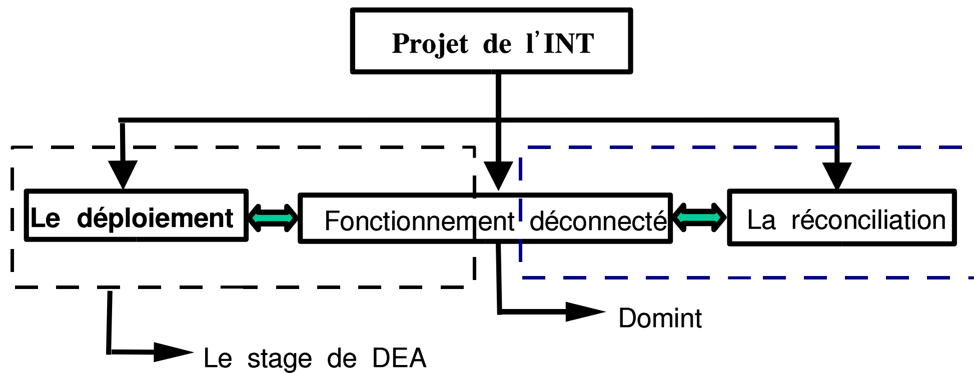

FIG. 2.1 – Positionnement du stage

les différentes objets distants et les objets déconnectés. Le prototype actuel de Domint ne s'intéresse qu'à la deuxième partie. Il définit trois mode de connectivité :

1. Mode fortement connecté : dans ce cas de figure, le mobile dispose d'une connexion normale au réseau, à la manière d'une station de travail classique. La connexion peut alors être réalisée soit par une liaison filaire, soit par une interface de communication sans fil, qui fournit en général un débit plus faible qu'une liaison câblée.
2. Mode partiellement connecté : le mobile ne dispose plus pour communiquer que d'un lien à faible débit. Cette perte de capacité peut être due à des perturbations physiques ou à des surcharge de traitements dans le réseau.
3. Mode déconnecté : dans ce mode le mobile est déconnecté soit parce qu'il n'y plus de liaison physique ou parce qu'il est impossible de maintenir une connexion sans fil.

L'objectif de ce stage et de fournir un protocole de déploiement des objets déconnectés sur le terminal mobile pour l'incorporer dans Domint. Ce protocole génère un autre problème au niveau du cache, puisque le prototype actuel de Domint ne tient pas compte de la présence de plusieurs objets dans le cache, qui dépend de la taille de la mémoire disponible. Ce problème nous conduit à proposer un mécanisme de gestion de cache au niveau client et pas au niveau de tout le système.

2.2 Résumé du contexte du stage

Les contributions de ce stage peuvent se résumer en deux grands points. Tous d'abord, nous proposons un protocole de déploiement des objets sur le terminal mobile basé sur la notion de criticité d'objet. Ensuite, nous proposons un mécanisme de gestion de cache basé sur le critére de la taille de la mémoire.

D'une manière générale, le contexte de notre travail peut se résumer dans les points suivants :
– Un modèle de conception orienté objets basé sur la spécification CORBA.
– Une architecture client-serveur flexible (dans le sens où le client peut être un serveur pour lui-même).
– Un fonctionnement des applications dans les trois modes de connectivité.

Chapitre 3

État de l'art sur la mobilité

Il existe de nombreux travaux liés à la mobilité dans les systèmes client-serveur, et plus récemment, sur les systèmes client-serveur orienté objets. C'est pourquoi cette étude commence par une présentation des domaines de la mobilité.

Dans ce chapitre et pour mieux comprendre le fonctionnement des applications mobiles, nous étudierons en premier lieu les paradigmes des modèles client-serveur mobiles (*Cf.* section 3.1), avant de parcourir les grands projets qui s'intéresse au problème de la mobilité (*Cf.* section 3.2). Enfin, la section 3.3 décrit les différentes techniques de déploiement de données sur les terminaux mobiles.

3.1 Paradigmes des modèles client-seveur mobiles

Les recherches qui existent sur les systèmes client-seveur mobiles peuvent se caractériser en deux grand paradigmes. Cette section présente ces deux paradigmes où nous verrons comment les applications client-seveur peuvent s'adapter au changement de l'environnement mobile et quelles sont les extensions à faire dans les modèles client-serveur classiques.

3.1.1 Adaptation des applications mobiles

Les systèmes client-serveur mobiles doivent avoir des réactions dynamiques dans l'interaction entre les clients mobiles et les machines statiques. D'après [Sat96a], l'intervalle des stratégies pour l'adaptation est délimité par deux extrémités (*Cf.* figure 3.1). Le premier extrême s'appelle "laisse-faire" : l'adaptation est entièrement de la responsabilité des applications. Il évite le besoin d'un support système, mais il manque un contrôleur central pour résoudre les demandes de ressources incompatibles entre différentes applications et pour imposer des limites sur l'utilisation des ressources. À l'autre extrémité, "transparence à l'application", l'adaptation est entièrement de la responsabilité du système. Donc, c'est au système de faire face aux problèmes de la mobilité des application, en particulier la variation de la bande passante du réseau. L'inconvénient de cette approche est qu'il peut y avoir des situations dans laquelle l'adaptation réalisée par le système est insatisfaisante ou même contre productive pour certaines applications. Entre ces deux extrêmes, il existe une autre stratégie d'adaptation appelée "collaboration entre l'application et le système", dans laquelle l'adaptation se fait en collaboration entre l'application et le système. Le système surveille les ressources et signale aux applications tout changement de niveaux de ressources. D'autre part, c'est le système qui fournit les mécanismes d'adaptation pour faire face aux problèmes de la mobilité des applications. L'application, quand à elle, fournit les politiques de cette adaptation, qui dépend de la sémantique de l'application.

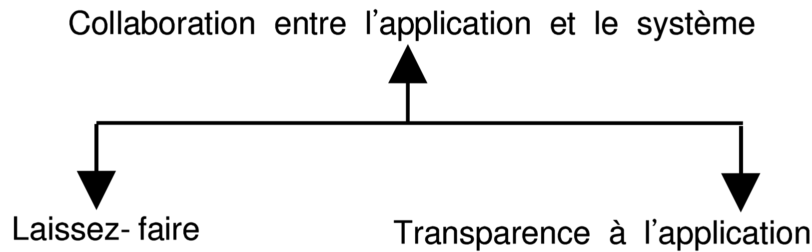

FIG. 3.1 – Les stratégies d'adaptation

3.1.2 Extension des modèles client-serveur

Une autre voie de caractérisation de l'impact de la mobilité est l'examen de son effet sur le modèle client-serveur classique. Le modèle client-serveur classique suppose que la localisation du client et du serveur ne peut pas changer et que la connexion entre ces machines ne peut pas changer non plus. En conséquence, les fonctionnalités entre le client et le serveur sont statiquement réparties.

Dans les environnements mobiles, la distinction entre le client et le serveur peut être temporairement flou [Sat96b]. Les ressources limitées d'un client peuvent satisfaire certaines opérations d'une manière normale et performante, mais il est besoin d'avoir de nouvelles ressources du serveur pour que le client manipule des données récentes. Par exemple, le client doit se débrouiller de fonctionner comme un serveur (pour lui ou pour d'autres machines), dans le cas où il existe des problèmes de connectivité, en créant des proxies dans la machine locale pour les données du serveur. D'après [JHE99], l'environnement mobile a généré de nouvelles techniques d'accès à l'information :

- Objets mobiles : ce sont des objets qui peuvent se déplacer d'une machine à une autre en exécutant une instruction du type "go". Ces objets donnent aux utilisateurs la possibilité d'exécuter des applications distantes dans leurs propres machines en téléchargeant les objets dans la machine locale. Dans notre travail, on ne s'intéresse pas à ce type d'objets puisque notre objectif et de créer des copies de ces objets dans les terminaux mobiles qui sont semblables aux objets distants et non pas la migration de ces objets.
- Mobilité virtuelle des serveurs : cette technique permet au client mobile de changer le serveur avec lequel il communique suivant l'endroit où il se trouve, ceci afin de réduire le temps de réponse. En conséquence, chaque serveur doit avoir une copie des données que le client manipule.
- Utilisation d'un proxy : un proxy sert d'un intermédiaire entre le client et le serveur. Il peut être local (dans la machine du client) ou distant (dans la machine du serveur ou sur une autre machine du réseau fixe). Le proxy permet de créer des copies locales des données du serveur pour augmenter la disponibilité des données dans le système. Cette technique est très utilisée dans les navigateurs Internet tel que WebExpress [HL96]. Le protocole de déploiement d'objets sur le terminal mobile que nous proposons se base sur l'utilisation d'un proxy dans la machine mobile.

3.2 Étude de projets

Cette section présente une étude de quatre prototypes de plates-formes qui permettent l'accès à l'information mobile. Ces systèmes servent à démontrer comment les paradigmes analysés dans les sections précédentes sont appliqués dans la pratique. Les quatre systèmes, à savoir Coda, Odyssey,

Rover et Bayou, démontrent quelques approches des nouveaux paradigmes pour les systèmes client-serveur mobiles.

3.2.1 Coda

Coda [Sat96b] est un système de gestion de fichiers. Son but est d'offrir au client la continuité d'accès aux données en cas d'arrêts des serveurs ou d'éventuelles déconnexions. Coda hérite de plusieurs caractéristiques d'utilisation et de conception de AFS (Andrew File System). Les clients visualisent Coda comme un simple système de gestion de fichiers partagé UNIX. L'espace de nommage de Coda est partagé sur plusieurs serveurs d'archivage qui construisent des sous-arbres appelés les volumes.

3.2.1.1 Opération déconnectée et mode faiblement connecté

L'opération de déconnectée est le premier concept introduit par Coda [KS91]. C'est une étape importante dans la gestion des déconnexions. Dans le mode déconnecté, le client continue d'avoir accès aux données dans son cache pendant les déconnexions intermittentes. La possibilité de fonctionner en mode déconnecté peut être utile même lorsque la connexion est disponible, par exemple, pour prolonger la vie de la batterie ou réduire les dépenses de transmission. La transparence est préservée du point de vue de l'application, puisque c'est le système qui porte la responsabilité de propager les modifications, de détecter les conflits et de faire les mises à jour quand la connexion est restaurée.

Dans les applications client-serveur mobiles, un client en mode déconnecté souffre de beaucoup de limitations telles que :

- La mise à jour n'est pas visible à tous les utilisateurs.
- L'absence des données dans le cache peut empêcher l'utilisateur de travailler sur son terminal mobile.
- La mise à jour est exposée au perte et à l'endommagement des données.
- Les conflits de mises à jour deviennent de plus en plus probables.

Pour traiter ces problèmes, Coda [Mum96] exploite la faible connectivité pour deux raisons principales, premièrement, faire une validation rapide du cache après une faute intermittente, deuxièmement, opérer une propagation des modifications "goutte à goutte" en tâche de fond pour ne pas dégrader les performances.

3.2.1.2 Gestion du cache du client mobile

Dans Coda, pour chaque client, un gestionnaire de cache appelé Venus. Il peut être dans l'état d'accumulation, d'émulation ou de réintégration. La figure 3.2 représente le diagramme de transition UML du gestionnaire de cache. Venus est normalement dans l'état accumulation : il se base sur les serveurs pour les opérations sur les fichiers mais reste toujours en alerte pour d'éventuelles déconnexions. 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. Á la reconnexion, Venus passe à l'état de réintégration pour réintégrer les modifications effectuées durant la déconnexion.

Dans l'algorithme de gestion de cache de Coda, Venus combine des sources de données implicite et explicite. Les données implicites se composent de l'historique du client, comme par exemple les fichiers les plus utilisés par l'utilisateur. Les données explicites prennent la forme d'une base de données spéciale au client *Hoard Data Base* (HDB). Le chargement des données explicites se fait au lancement de l'application. Un programme appelé *"Hoard"* permet à l'utilisateur de mettre à

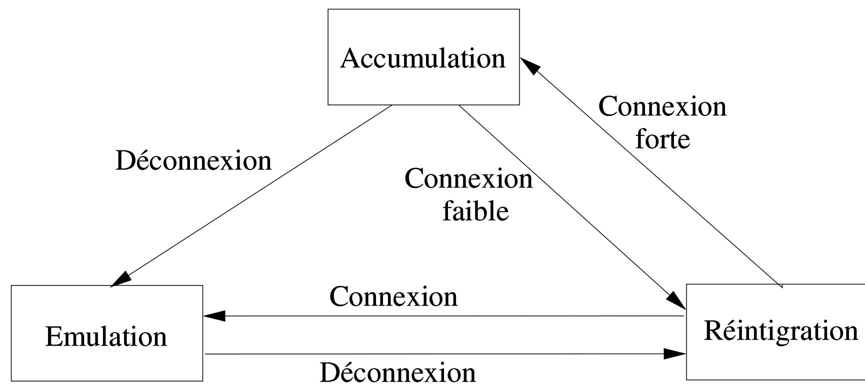

FIG. 3.2 – Les états de Venus dans Coda

jour la HDB directement ou via un script de commandes appelé *"Hoard Walking"*. L'absence de donnée dans le cache ne peut pas être masquée, elle apparaît comme erreur aux programmes et aux utilisateurs de l'application. La persistance des changements fait lors de la déconnexion est réalisée par des opérations de journalisation sur un journal d'opérations appelé le *Change Modify Log* (CML) . Venus met en application un certain nombre d'optimisations pour réduire la taille du CML. Avant qu'un enregistrement d'opération soit ajouté au CML, Venus contrôle s'il peut annuler l'effet de précédents enregistrements. Par exemple, si un client fait deux opérations d'écritures et une opération de suppression sur un fichier dans le cache, Venus ne gardera que l'opération de suppression dans le CML.

3.2.1.3 Détection et résolution des conflits

Coda traite le problème des mises à jour en utilisant une stratégie optimiste de contrôle des répliques. Ceci offre un degré plus élevé de disponibilité, puisque les données peuvent être mises à jour dans n'importe quel serveur. Lors de la réintégration, le système assure la détection des mises à jour contradictoires et fournit des mécanismes d'aide aux utilisateurs pour éliminer ces problèmes.

Quand un fichier est mis à jour, le serveur incrémente le numéro de version du fichier et celui du volume qui contient le fichier. Quand la connexion est restaurée, le client présente les numéros de volume pour la validation. Si le numéro de volume est valide alors Coda valide tous les fichiers cachés de ce volume. S'il est incorrect alors il doit valider fichier par fichier. D'autre part, si la validation individuelle d'un fichier n'est pas possible à cause de la faible connectivité, le client peut supposer que tous les éléments dans le volume sont incorrects ou reporter la validation jusqu'à la prochaine utilisation de l'élément.

Coda ne possède pas la connaissance sémantique suffisante pour résoudre des conflits de fichier, il offre un mécanisme pour installer et appeler d'une manière transparente un résolveur spécifique à l'application (ASR). L'ASR est un programme qui encapsule la connaissance détaillée et nécessaire à l'application pour distinguer les incohérences des différentes réconciliations. Si l'ASR ne peut pas résoudre les conflits, l'incohérence est exposée à l'utilisateur pour la réparation manuelle.

3.2.2 Odyssey

Avant de présenter Odyssey, nous présentons un facteur important dans la construction des systèmes réalisant une adaptation "collaboration entre l'application et le système" ; ce facteur est l'agilité [NSN+97].

3.2.2.1 Agilité

Prendre une meilleure décision exige une connaissance plus précise de la disponibilité de ressources. Dans le meilleur des cas, une application devrait toujours avoir la connaissance parfaite des niveaux de disponibilité de ressources. En d'autres termes, il ne devrait y avoir aucun délai entre un changement de valeur de la disponibilité et la détection de changement. L'agilité est la propriété d'un système mobile qui détermine l'environnement le plus turbulent dans lequel l'application peut fonctionner d'une manière acceptable. L'agilité peut avoir plusieurs niveaux de complexité : par exemple, un système peut être beaucoup plus sensible aux changements de la largeur de bande passante du réseau qu'aux changements du niveau de puissance de la batterie. Un autre point important dans l'agilité est l'existence de différentes origines des changements de la disponibilité des ressources. Le changement peut être provoqué par la variation dans l'approvisionnement en ressources due à la mobilité ou par les demandes concurrentes des applications. Ces deux types de changement peuvent être détectés par la mise en œuvre de mécanismes de contrôle de ressources.

3.2.2.2 Description d'Odyssey

Odyssey [Sat96a] est un projet de recherche de CMU mené par M. Satyanarayanan à la suite du projet Coda . Odyssey utilise dans la gestion des données un paramètre qui est la fidélité des données. La notion de fidélité dépend du type de données dans le système de fichiers. Dans Odyssey, pour chaque type de ressources et à tout instant, une fenêtre de tolérance est délimitée par une borne inférieure et une borne supérieure. Part conséquent, pour chaque ressource il existe plusieurs niveaux de fidélité. Par exemple, dans les applications multimedia qui traitent des fichiers de type image, nous donnons pour chaque image trois niveaux de fidélité suivant le nombre de couleurs disponible (noir et blanc, 256 couleurs, 16 millions couleurs). Le système de gestion de fichiers Odyssey est décomposé en plusieurs sous-espaces appelés des tomes. Les tomes sont conceptuellement semblables aux volumes dans Coda, mais ils ajoutent la notion de type, le type d'un tome détermine le type de ressources (image, texte, vidéo ...). En outre, un tome réside entièrement sur un seul serveur.

3.2.2.3 Architecture

L'architecture d'Odyssey est présentée dans la figure 3.3. D'après cette figure, Odyssey offre deux fonctionnalités. Premièrement, une fonctionnalité générique qui est implantée par un vice-roi (*"viceroy"* en anglais) qui a le rôle d'un contrôleur de ressources dans le système. Le vice-roi a un rôle central dans la négociation des ressources. Il surveille la disponibilité de la ressource et notifie l'application quand les bornes de la fenêtre de tolérance sont franchies. Deuxièment, Odyssey offre une fonctionnalité spécifique au type du tome. Cette fonctionnalité est implantée dans le gestionnaire de ressources qui s'appelle le gardien (*"Warden"* en anglais). Il y a un gardien pour chaque type de tome. En outre, l'architecture d'Odyssey permet de :

1. Faire des opérations sur des objets Odyssey : Odyssey est intégré dans NetBSD comme un nouveau VFS (Virtual File System). Le vice-roi et les gardiens sont implantés dans l'espace de l'utilisateur plutôt que dans le noyau du système. Les opérations sur les objets d'Odyssey sont réorientées vers le vice-roi par un intercepteur dans le noyau représenté dans la figure par le rectangle "Intercepteur". Tous les autres appels du système sont traités directement par NetBSD. Les gardiens sont statiquement liés avec le vice-roi, la communication entre eux se fait par des appels de procédures et des structures de données partagées. Les gardiens sont entièrement responsables de la communication avec les serveurs. L'application ne contacte jamais directement le serveur.

2. Demander des ressources : l'application demande les ressources à Odyssey en utilisant l'appel système suivant :

```
request (in path, in resource-descriptor, out request-id)
cancel (in request-id)
```

   L'appel prend un descripteur de ressources identifiant une ressource et indiquant une fenêtre de tolérance sur sa disponibilité. Cet appel exprime le désir de l'application d'être notifiée si la disponibilité de la ressource est en dehors des bornes de la fenêtre de tolérance.

3. Notifier les applications : quand le vice-roi découvre que la disponibilité d'une ressource est inférieure à la fenêtre de tolérance, il envoie une notification "upcall" à l'application correspondante pour ajuster sa fidélité selon la politique de l'application.

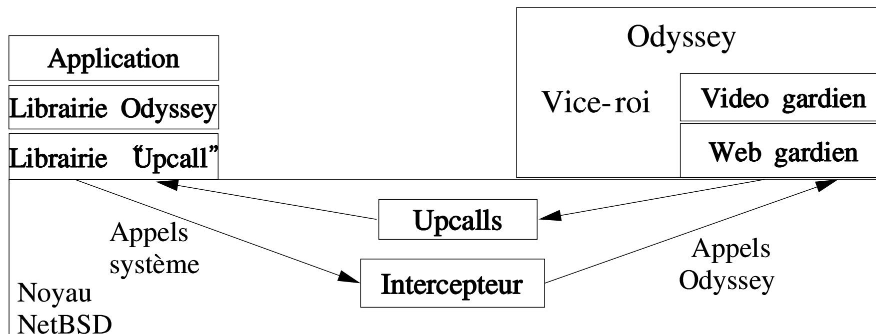

FIG. 3.3 – L'architecture d'Odyssey

3.2.3 Bayou

Bayou est un projet de Xerox PARC. Son but est de construire un support système pour partager des données entre utilisateurs mobiles via une communication point-à-point entre les terminaux (directement ou par l'intermédiaire d'autre machines) [TTP+95, PTT+94] .

Le client peut accéder à des données de n'importe quel serveur avec lequel il peut communiquer. Réciproquement, n'importe quelle machine y compris les machines mobiles, qui possède une copie d'une collection de données doit être disponibles pour des requêtes d'écriture ou de lecture pour d'autre machines. Donc, les machines mobiles peuvent être des serveurs pour certaines bases de données et des clients pour d'autres.

3.2.3.1 Architecture

Dans Bayou, chaque collection de données est répliquée dans plusieurs serveurs de données (*Cf.* figure 3.4). L'application cliente communique avec le serveur à travers une interface spécifique à Bayou. Cette interface est implantée dans la souche du client et elle fournit deux opérations de base de lecture et d'écriture.

Bayou réalise un schéma de réplication *Read-any/Write-any*, c'est-à-dire Bayou utilise un modèle de cohérence faible des répliques. Un problème de l'approche *Read-any/Write-any* est que les incohérences peuvent apparaître même lorsque seulement un utilisateur ou une application fait des modifications sur les données. Par exemple, un client écrit sur une base de données d'un serveur, et après un moment, lit des données d'un autre serveur. Le client peut voir des résultats

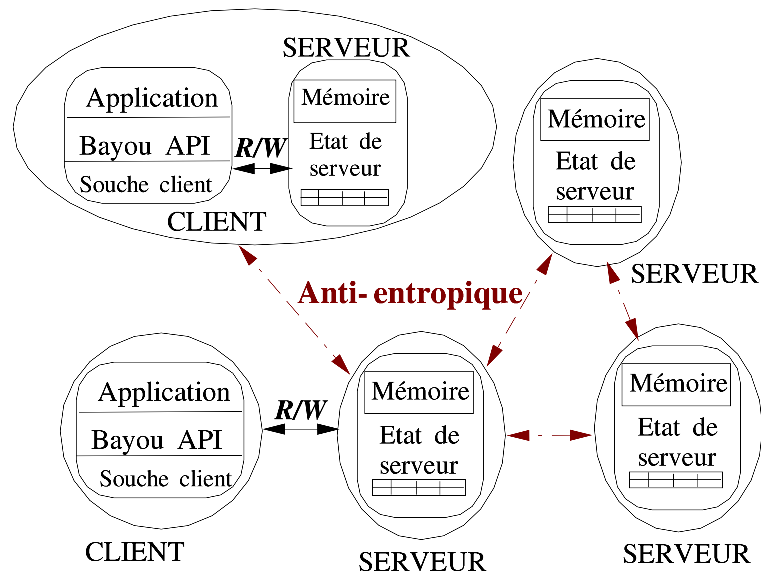

FIG. 3.4 – L'architecture de Bayou

contradictoires à moins que les deux serveurs aient mis à jour leurs bases de données de façon cohérente.

Pour atteindre le maximum de cohérence entre les différentes copies de la base de données, Bayou utilise un protocole dit anti-entropique pour la propagation des mises à jour [PTT+97]. Le protocole anti-entropique assure que toutes les copies d'une base de données sont convergentes vers le même état. En plus, le protocole anti-antropique permet à deux machines quelconques qui peuvent communiquer de propager périodiquement leurs mises à jour entre elles.

3.2.4 Rover

Rover est un projet de recherche au MIT (Massachusetts Institute of Technology) mené par Kaashoek [JTK97]. Rover offre un environnement pour supporter une adaptation transparente à l'application et une adaptation par collaboration entre l'application et le système pour des applications client-serveur mobiles. L'adaptation transparente à l'application est réalisée en développant des proxies pour les services du système. L'adaptation par collaboration entre l'application et le système introduit les concepts d'objet dynamique ré-adressables RDO (*"Relocatable Dynamic Objects"* en anglais) et d'appel de procédure à distance non-bloquant QRPC (*"Queued Remote Procedure Call"* en anglais).

3.2.4.1 Objets dynamiques ré-adressables et appel de procédure à distance non-bloquant

Un RDO est un objet qui peut être dynamiquement chargé sur le client à partir du serveur (ou vice versa) pour réduire les communications client-serveur et offrir une disponibilité de l'objet du côté client. Le client utilise ces objets même dans le cas où il est en mode fortement connecté. Dans le cycle de vie d'un RDO, son état peut être importé mais pas encore arrivé, présent dans l'environnement local, peut-être modifié localement par des invocations de méthodes, en attente pour être exporté vers le serveur, ou en attente dans le serveur pour être réconcilié.

Les QRPC est 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.

Dans ce rapport, on ne traite pas les appels (asynchrones) non-bloquants de CORBA.

3.2.4.2 Architecture

L'application Rover emploie des modèles check-in, check-out pour la manipulation des objets partagés [JTK95]. Dans la figure 3.5, l'application importe des RDO dans son espace d'adressage, invoque les méthodes fournit par ces objets et exporte ces objets vers le serveur. Rover stocke les objets dans le terminal mobile dans un cache qui est partagé par toutes les applications Rover de ce terminal. Les objets déployés sont une réplique des objets du serveur.

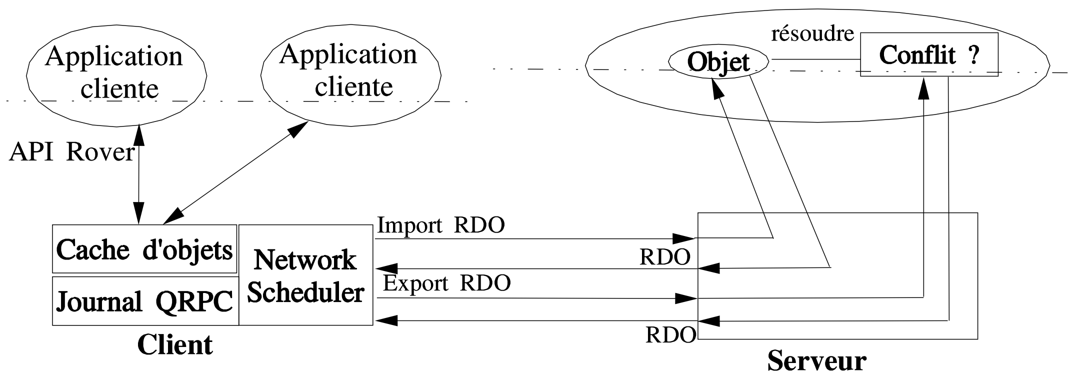

FIG. 3.5 – L'architecture de Rover

Lorsque l'application Rover invoque une méthode d'un objet, Rover contrôle d'abord le cache. Si l'objet réside dans le cache, Rover applique l'invocation directement sur l'objet du cache sans contacter le serveur, même s'il n'y a pas de problème de connexion. Ainsi, ces objets sont marqués provisoirement valides, ces objets seront marqués valides lorsque les invocations seront accomplies sur les objets serveurs. Dans ce cas, les applications peuvent continuer de fonctionner même dans le cas des déconnexions intermittentes. Si l'objet est non présent dans le cache, Rover cherche cet objet dans le serveur en utilisant des requêtes QRPC. Il sauvegarde ces QRPC dans un journal d'opération sur le client et retourne le contrôle à l'application. Lorsque le client mobile se reconnecte, le Network Scheduler (*Cf.* figure 3.5) transmet ces opérations vers le serveur. Le Network Scheduler peut délivrer les QRPC dans un autre ordre que l'ordre de leur émission (cela dépend de la priorité et du coût de communication avec le serveur). Lorsqu'une invocation de méthode modifie l'objet dans le cache, Rover met à jour la copie primaire (dans le serveur). Rover maintient un vecteur de version [RS96] de chaque objet pour que les méthodes puissent facilement détecter les changements sur l'objet. Les méthodes des RDO comportent du code de détection et de résolution des conflits qui dépend de la sémantique de l'application.

L'architecture de Rover est structuré en trois couches (application, support système, transport) et se compose de quatre composants. Le premier composant est le gestionnaire d'accès. Il est responsable du traitement de toutes les interactions entre les applications clientes et le serveur, et entre les applications clientes entre-elles. 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 contrôler le cache des objets. Le deuxième composant est le cache d'objets. Il fournit une mémoire stable pour les copies locales des objets importés, il 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 cache d'objets offre plusieurs option, pour maintenir la cohérence des objets avec la copie primaire :Verify-before-use, Verify-if-service-is-accessible, Expire-after-date, Service-callback "Rover essaie d'informer le client lorsque l'objet change".

Le troisième composant est le journal d'opérations. Il permet d'enregistrer les requêtes envoyées par le client vers les objets locaux pour les exécuter sur les objets distants. Le quatrième composant est le Network Scheduler. Il permet de mettre en œuvre un mécanisme de contrôle de ressources réseau pour choisir le moment de faire la mise à jour qui dépend de l'architecture de réseau et la largeur de la bande passante.

3.2.5 Autre projets

Dans le contexte de CORBA qui est le contexte de notre travail, deux projets, Π^2 [RSK00] et ALICE [Lyn99], traitent le problème de la mobilité matérielle ou physique dans le contexte de CORBA sans file (*"Wireless CORBA"*). Dans ce cas, la mobilité matérielle est provoquée par le changement de cellule (*"handover"*) et par conséquent des déconnexions courtes.

L'architecture de Π^2 se base sur la mise au point de deux proxies (un sur le terminal mobile et un autre dans le réseau filaire) dans le but de rendre les déconnexions involontaires transparentes à l'utilisateur.

ALICE utilise aussi un proxy. 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 au client. Le code nécessaire pour commuter vers le mode déconnecté doit être inclus dans le code de l'application.

3.3 Déploiement

«Un cache est un système qui se place entre les fournisseurs de données (les serveurs) et les consommateurs (les clients) »[Bag98]. Un grand nombre de travaux ont été réalisés ces dernières années sur les techniques et les méthodes de gestion du cache. Cette section présente une synthèse de l'état de l'art précédent sur la question précise des différentes stratégies de déploiement.

3.3.1 Pas de cache

Cette stratégie est utilisée pour les terminaux mobiles qui ne disposent pas d'un support spécifique à la mobilité. Dans ce cas, l'utilisateur déploie une copie de données manuellement pour travailler lors des déconnexions. Cette stratégie pose beaucoup de problèmes. Premièrement, elle est limitée à des applications qui traitent des fichiers ou des bases de données, car il est très difficile de décider quel objet charger sur le terminal puisque les objets ne sont pas manipulables directement par l'utilisateur. Deuxièmement, cette stratégie ne fonctionne pas pour les déconnexions involontaires.

Pour régler le problème des déconnexions involontaires, une technique a été proposée pour cette approche où l'application attend jusqu'à ce que l'état du réseau décroisse ou jusqu'à ce qu'une déconnexion peut se produire bientôt pour informer le client qui doit sauvegarder une copie des données qu'il manipule. Le problème avec cette approche est qu'un transfert peut ne pas se produire avec succès.

3.3.2 Mise en cache systématique

Dans cette stratégie, si l'application essaie d'accéder à l'objet distant, le système crée automatiquement une copie de cet objet dans le terminal mobile et le client utilise uniquement cette copie. Un avantage de cette technique est qu'elle offre une grande disponibilité des objets dans le cache. En revanche, le cache ne contient que les objets déjà utilisés. Donc, en mode déconnecté, le client n'a pas le droit d'invoquer de nouveaux objets. En outre, puisqu'à chaque fois qu'un objet est invoqué une copie locale est créée dans le cache, une politique de gestion de l'encombrement de la taille du cache est obligatoire. Bayou utilise cette approche qui se base sur la copie totale de la base de données dans les terminaux mobiles.

3.3.3 Mise en cache à la demande

Dans cette approche, la création d'une copie locale se fait à la demande de l'utilisateur. Cette approche est très utilisée dans les navigateurs Web tels que Exmh avec Rover [JHE99]. Le chargement des objets se fait à la première invocation. La différence avec la mise en cache systématique est que le système crée une copie locale uniquement pour les objets sélectionnés par l'utilisateur. Dans cette stratégie, le problème d'encombrement du cache est géré par le client et un objet non vu en cache ne sera pas disponible lors des déconnexions.

Le système SEER [Kue94] utilise une approche prédictive automatique de déploiement. Cette approche est basée sur l'idée qu'un système peut observer le comportement de l'utilisateur, faire des inférences sur les relations sémantiques entre les fichiers et utiliser ces inférences pour aider l'utilisateur à sélectionner les objets à mettre en cache. Un composant observateur observe le comportement de l'utilisateur et ses accès aux fichiers, classant chaque accès selon le type (de fichier : texte, vidéo, image ...). SEER [GG97] emploie un concept connu de distance sémantique pour mesurer l'intuition d'un utilisateur concernant les relations entre les fichiers. SEER définit plusieurs distance sémantique entre les fichiers :

- La distance sémantique temporelle est la durée qui sépare l'utilisation des deux fichiers.
- La distance sémantique à base de séquences est le nombre de références vers d'autre fichiers entre l'utilisation des 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'interventions d'ouvertures d'autres fichiers y compris l'ouverture du fichier B.

L'observateur fournit les résultats des observations à un composant de corrélation. Le composant de corrélation évalue les références des fichiers et calcule les distances sémantiques. Quand un nouveau contenu de cache doit être choisi, le composant de corrélation examine l'ensemble des fichiers pour trouver ceux qui sont actuellement en activité et choisit les fichiers ayant la plus grande distance sémantique jusqu'à ce que la taille maximum du cache soit atteinte.

3.3.4 Pré-chargement

Cette stratégie permet de charger des objets au lancement de l'application. La liste des objets a déployer sur le terminal mobile est pré-définie. Cette approche est utilisée dans plusieurs systèmes tels que Coda (*Cf.* section 3.2.1) et UPidata [AFM98] qui utilisent une approche de pré-chargement périodique avec notification des utilisateurs lorsque l'état de l'objet change.

Le pré-chargement peut être fait au moment de l'exécution de l'application. Par conséquent, la liste des objets a déployer sur le terminal mobile est dynamique. Dans ce cas, le pré-chargement consiste à charger des données dans le cache lorsque l'application prédit que l'utilisateur va effectuer une requête dans un futur proche. Si la prédiction est exacte, l'utilisateur gagne le temps de retrait du document. Si la prédiction est erronée, l'utilisateur a consommé des ressources du réseau et de stockage dans le cache pour rien. Par exemple, l'approche classique pour le pré-chargement dans les applications Web consiste à extraire les liens hyper-texte contenus dans les documents demandés par l'utilisateur et à commencer aussitôt à charger les documents sur lesquels ils pointent. Cette approche est coûteuse. Dans certains cas, cela peut mener à pré-charger un grand nombre de documents inutilement. Il est préférable d'essayer d'identifier parmi eux un petit nombre de documents davantages susceptibles d'être accédés.

Le tableau 3.1 donne un résumé sur les projets étudiés et leurs approches de déploiement.

Système	**Méthode de déploiement**
Coda	pré-chargement (une liste pré-définie par l'utilisateur)
Rover	à la demande (à la première invocation)
ALICE	à la demande (à la première invocation)
Bayou	systématique
SEER	prédictive automatique
UPidata	pré-chargement

TAB. 3.1 – La méthode de déploiement dans les différents systèmes

3.4 Conclusion

Dans ce chapitre, nous avons esquissé un état de l'art sur l'adaptation des applications client-serveur dans les environnements mobiles et les extensions à faire dans ces applications pour faire face à la mobilité. Ensuite, nous avons étudié quatre systèmes qui traitent le problème de la mobilité. Enfin, nous avons fait une synthèse sur les approches de déploiement utilisées dans les systèmes étudiés dans ce chapitre. Le prototype actuel de Domint utilise l'approche de mise en cache systématique pour le déploiement des objets dans le cache. Dans la suit de ce rapport, nous allons présenté notre approche de déploiement qui utilise une mise en cache systématique pour une catégorie d'objets, et un pré-chargement pour d'autre catégorie d'objets. Mais avant de décrire ces catégories d'objets, nous présentons dans le chapitre suivant l'architecture de Domint.

Chapitre 4

Architecture de Domint

Dans les applications distribuées classiques avec une forte connectivité, l'interface graphique de l'utilisateur GUI (*Graphical User Interface*) est chargée dans le terminal mobile, tandis que les objets du serveur résident sur une machine du réseau fixe. La continuité du service dans le cas où le terminal mobile se déconnecte est assurée par le déploiement des objets du serveur dans le terminal mobile avant la perte de connectivité. Dans le mode déconnecté, le terminal mobile journalise les opérations effectuées sur les objets locaux. La réconciliation entre les différentes copies est effectuée lorsque la connexion est restaurée.

Dans ce chapitre, nous présentons la plate-forme Domint (*Cf.* section 4.2), après avoir donner une courte présentation d'un mécanisme CORBA utilisé dans le développement de la plate-forme Domint : les intercepteurs portables (*Cf.* section 4.1). Nous signalons que l'étude de Domint et des intercepteurs portables est très importante pour comprendre notre approche de déploiement et de gestion du cache présentée dans le chapitre 5. En outre, des détails techniques sur Domint est les intercepteurs portables sont donnés dans ce chapitre.

4.1 Les intercepteurs portables

Un point d'interception est un point d'accrochage dans l'ORB (*"Object Request Broker"*) par lequel les services de l'ORB peuvent intercepter le flux normal d'exécution [OMG01]. Ces intercepteurs sont portables au sens où le service est utilisable dans tous les ORB. Donc, un intercepteur est construit indépendamment de l'ORB. Les intercepteurs portables sont des objets CORBA, leur interface est spécifiée dans le module *PortableInterceptor*.

CORBA définit deux groupes d'intercepteurs : les intercepteurs des requêtes et les intercepteurs d'IOR (*"Interoperable Object Reference"*). Les services de l'ORB peuvent utiliser ces intercepteurs pour faire des traitements sur les requêtes-réponses et échanger des informations en ajoutant un contexte de service aux requêtes et aux réponses.

4.1.1 Intercepteur de requêtes

L'intercepteur de requêtes est implanté pour intercepter le flux des requêtes-réponses à travers l'ORB en des points spécifiques appelés des points d'interception. Deux types d'intercepteur de requêtes sont définis : les intercepteurs côté client et les intercepteurs côté serveur.

4.1.1.1 Intercepteur côté client et côté serveur

Dans la figure 4.1, dans l'ORB côté client, il y a cinq points d'interceptions. Les points `Send_request` et `Send_poll` interceptent toutes les requêtes que le client émet vers le serveur. Les points d'interception `Receive_reply`, `Receive_exception` et `Receive_other` permettent d'intercepter toutes les réponses destinées au client. Similairement, il y a cinq points d'interception dans l'ORB côté serveur. Les points d'interception `Receive_request_service_contexts` et `Receive_request` interceptent toutes les requêtes destinées au serveur. Les points d'interceptions `Send_reply`, `Send_exeption` et `Send_other` interceptent toutes les requêtes du serveur vers le client.

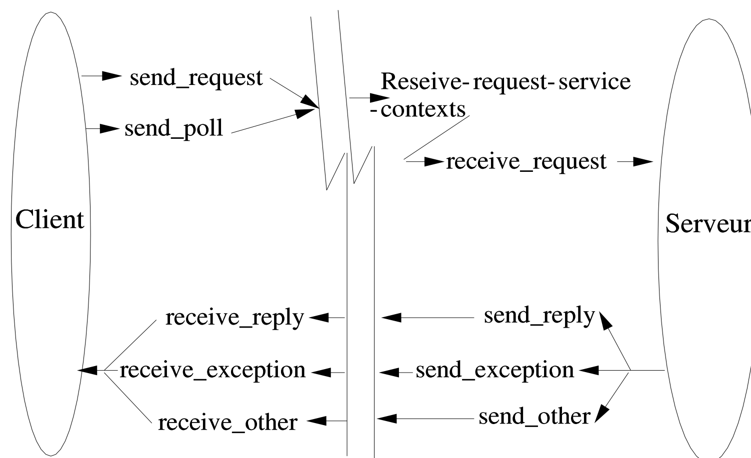

FIG. 4.1 – Les points d'interception

4.1.1.2 Informations sur la requête

À chaque point d'interception est donné un objet `ClientRequestInfo` côté client et un objet `ServerRequestInfo` côté serveur à travers lequel l'intercepteur portable peut accéder aux données de la requête. Il existe deux objets d'information qui sont : . Ces deux objets hérites de la même interface `RequestInfo` décrite de la façon suivante :

```
local interface RequestInfo {
        readonly attribute unsigned long request_id;
        readonly attribute string operation;
        readonly attribute Dynamic::ParameterList arguments;
        readonly attribute Dynamic::ExceptionList exceptions;
        readonly attribute Dynamic::ContextList contexts;
        readonly attribute Dynamic::RequestContext
                      operation_context;
        readonly attribute any result;
        readonly attribute boolean response_expected;
        readonly attribute Messaging::SyncScope sync_scope;
        readonly attribute ReplyStatus reply_status;
        readonly attribute Object forward_reference;
        any get_slot (in SlotId id) raises (InvalidSlot);
        IOP::ServiceContext get_request_service_context (
```

```
            in IOP::ServiceId id);
        IOP::ServiceContext get_reply_service_context (
            in IOP::ServiceId id);
};
```

4.1.2 Intercepteur de références d'objets

Dans plusieurs cas, l'ORB a besoin d'ajouter des composants aux références des objets pour permettre aux deux autres types d'intercepteurs de déterminer le traitement à effectuer. Cette fonctionnalité est réalisée à travers les interfaces `IORInterceptor` et `IORInfo` définies comme suit :

```
local interface IORInterceptor : Interceptor {
      void establish_components (in IORInfo info);
};

local interface IORInfo {
    CORBA::Policy get_effective_policy
        (in CORBA::PolicyType type);
    void add_ior_component
        (in IOP::TaggedComponent a_component);
    void add_ior_component_to_profile
        (in IOP::TaggedComponent a_component,
         in IOP::ProfileId profile_id);
};
```

4.2 Plate-forme Domint

Domint [CCVB02b, CCVB02c] est une plate-forme CORBA supportant le fonctionnement déconnecté. Dans Domint, CORBA est choisi pour sa capacité à être utilisé dans des domaines multiples et parce qu'il fournit des mécanismes tels que les intercepteurs portables pour établir une adaptation transparente à l'application.

Les principaux objectifs de Domint peuvent se résumer en :

- Supporter l'accès concurrent de plusieurs applications sur les objets locaux.
- Centraliser le gestionnaire de ressources et le gestionnaire du journal d'opérations, en d'autres termes, fournir un gestionnaire de ressources et un gestionnaire du journal d'opérations pour toutes les applications qui tournent sur le terminal mobile.

Le prototype actuel de Domint se base sur deux hypothèses [CCVB02a]. Premièrement, l'objet distant ne peut pas être accédé concurremment par plusieurs clients. Par conséquent, Domint ne gère pas le problème de cohérence des données. Deuxièmement, les requêtes du client ne comportent pas un mixage de paramètres de types *in, in-out* et *out*. Donc, le prototype actuel de Domint ne supporte que les requêtes qui soit renvoient des résultats soit acceptent des paramètres en entrée.

Afin de continuer à travailler même lors de déconnexion, l'idée principale de Domint est de créer sur le terminal mobile des objets proxies appelés les objets déconnectés (DO). Un DO est un objet CORBA qui est semblable dans la conception à l'objet du côté du serveur, mais spécifiquement construit pour faire face aux déconnexions et à la faible connectivité.

Dans les prochaines sections, nous décrivons l'architecture de Domint et le fonctionnement de tous les objets de Domint.

4.2.1 Architecture

L'architecture de Domint est représentée dans les figures 4.2 et 4.3 [CCVB02c]. Les deux figures présentent respectivement, des diagrammes de collaboration UML du client envoyant la première demande à un objet distant quand la connectivité est forte puis envoyant une demande dans le cas de faible connectivité. Dans le reste de la section, nous présentons les différentes entités de l'architecture de Domint.

Tous les rectangles sur les figures 4.2 et 4.3 représentent des objets CORBA. L'intercepteur portable PI est également un objet CORBA local, c'est-à-dire qu'il ne peut pas être appelé en dehors de son entité d'exécution (une entité d'exécution est attachée à un seul ORB). Toutes les requêtes du ou vers le client sont interceptées par PI. Sur les requêtes sortantes, PI agit en tant que commutateur entre le DO et l'objet distant. Sur la réception de la réponse, PI détecte les fautes de communication entre l'envoi de la demande et la réception de la réponse.

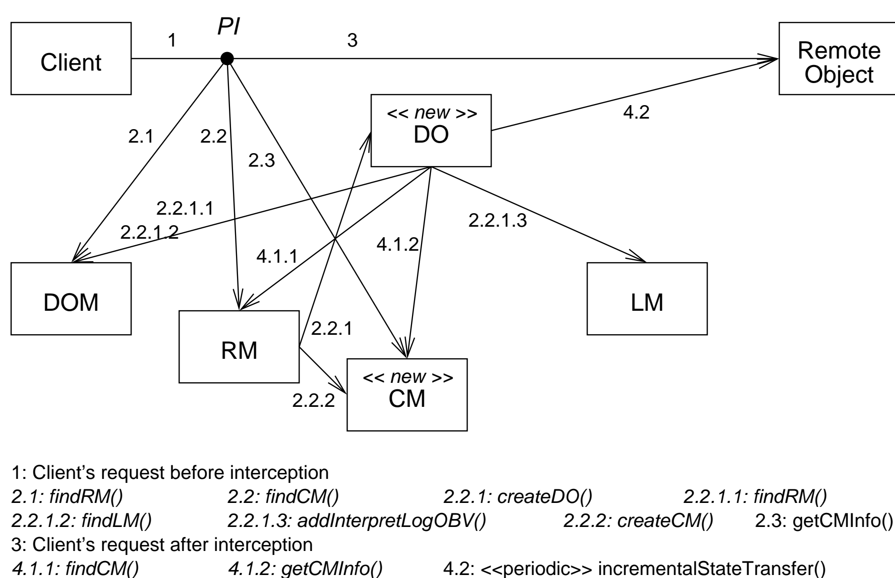

FIG. 4.2 – Le comportement de Domint en mode fortement connecté

L'interface du client (Client dans les diagrammes) et le PI appartiennent à la même entité d'exécution tandis que le gestionnaire d'objets déconnectés DOM, le gestionnaire de ressources RM, le gestionnaire de connectivité CM et le gestionnaire du journal LM sont groupés dans une autre entité d'exécution, également sur le terminal mobile. L'entité d'exécution des gestionnaires est lancée avant que le client ne manipule des applications dans son terminal. Le DOM est le point d'entrée pour trouver les autres gestionnaires. Le RM est une fabrique de CM. Dans Domint, à chaque DO correspond un CM .

Au première appel de l'objet distant, RM crée un DO et le CM correspondant sur la demande de PI. La figure 4.2 montre les interactions entre les différents objets de Domint lors du premier appel à un objet distant dans le cas de la forte connectivité. Dans le diagramme, chaque flèche est une requête CORBA. Les demandes 1 et 3 sont des cas particuliers : la requête 1 est interceptée par PI qui ne l'interprète pas mais laisse l'ORB la transmettre en tant que requête 3 à l'objet distant. Entre ces deux demandes, PI recherche la référence du CM associé à cet objet dans sa table interne et conclut que c'est le premier appel pour cet objet distant. Dans ce cas, PI demande au DOM (2.1) la référence du RM pour que ce dernier crée le DO (2.2.1) et le CM associé (2.2.2). Pendant sa construction, DO obtient les références de RM et de LM en appelant le DOM (2.2.1.1, 2.2.1.2). Ensuite, PI décide où

la requête du client va être envoyée (soit vers l'objet distant soit vers l'objet déconnecté). Dans le scénario où la connectivité est forte, les requêtes du client sont exécutées sur les objets distants (3). Enfin, l'objet déconnecté appelle périodiquement l'objet distant pour un transfert incrémental d'état (4.2). Cependant, il doit d'abord contrôler la connectivité en appelant le CM (4.1.2).

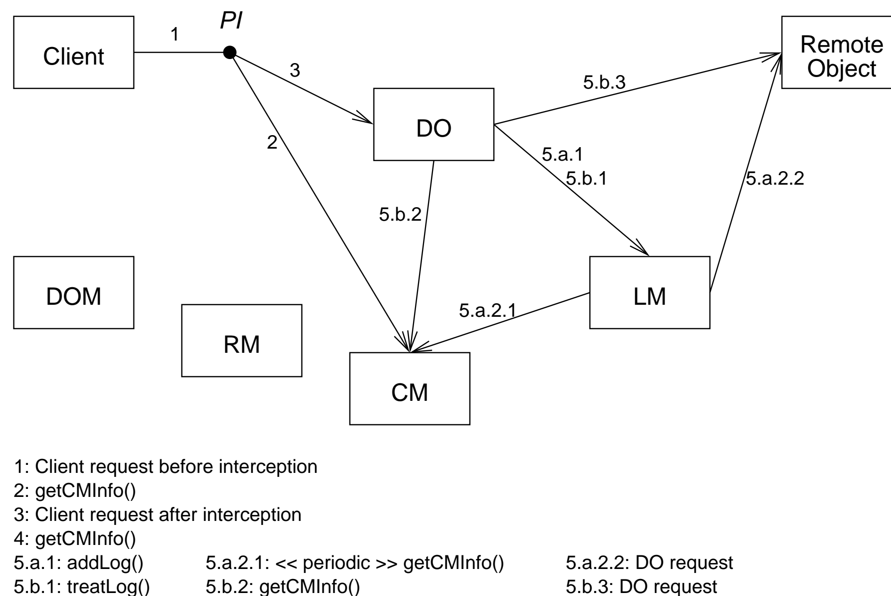

FIG. 4.3 – Le comportement de Domint en mode partiellement connecté et déconnecté

Quand la connectivité devient faible ou nulle, le client entre dans les modes partiellement connecté ou déconnecté respectivement. Ce cas est représenté par la figure 4.3. Les requêtes du clients sont interceptées par le PI (1). PI obtient l'information de connectivité du CM (2) et oblige l'ORB à transmettre les requêtes du client vers le DO (3). Domint donne deux scénarios possibles d'exécution dans ce mode : 5.a et 5.b. Si la requête du client est interprétée en tant que fournissant des informations à l'objet distant, le cas 5.a, DO met à jour son état et prépare une nouvelle requête, appelée une DO requête, pour l'objet distant. Le cas le plus simple est que la DO requête est équivalente en termes de types et de nombre de paramètres et en termes de contenu à la requête du client. Ensuite, DO encode la requête dans un Any CORBA [GAK00] et envoie cet Any au LM (5.a.1). Périodiquement, le LM décode la requête enregistrée, teste la connectivité (5.a.2.1), et si possible, envoie la requête à l'objet distant. Le deuxième cas 5.b se produit par exemple quand la requête du client est interprétée comme une requête qui renvoie des résultats au client. Si la taille du journal d'opération indiquée par le LM (5.b.1) est nulle et l'information sur la connectivité obtenue à partir du CM (5.b.2) permet la transmission sans fil avec l'objet distant, DO essaie d'abord de charger l'information demandée par le client de l'objet distant (5.b.3) et ensuite de mettre à jour son état avant de renvoyer l'information au client.

4.2.2 Quelques détails sur les objets de Domint

Cette section donne le rôle de chaque entité dans l'architecture de Domint, présente l'interface écrite dans CORBA IDL de ces entités et développe les éléments (attributs, exécutions) de l'interface de chaque entité.

4.2.2.1 Gestionnaire d'objets déconnectés

Comme déjà indiqué dans la dernière section, DOM est le point d'entrée du service de gestion des objets déconnectés, son nom d'interface est *DOManager*. L'interface fait partie du module nommé *dom* qui contient également les sous-modules *rm* et *lm* pour le gestionnaire de ressources et le gestionnaire du journal respectivement, et le sous-module *pi* qui décrit l'intercepteur portable. Les méthodes *FindLogManager()* et *findResourceManager()* servent à obtenir la référence du gestionnaire du journal et le gestionnaire de ressources.

```
module dom {
  interface DOManager {
    lm::LogManager findLogManager();
    rm::ResourceManager findResourceManager();
  };
  module rm; module lm; module pi;
};
```

4.2.2.2 Gestionnaire de ressources

Le gestionnaire de ressource centralise la commande de toutes les ressources dans le terminal mobile. Domint se focalise sur les ressources liées à la connectivité : l'activité du réseau, largeur de bande disponible, coût de transmission, charge de la batterie... L'intercepteur, l'objet déconnecté et le gestionnaire du journal obtiennent la référence de CM grâce à la méthode *findConnectivityManager()* du gestionnaire de ressource. Les gestionnaires de connectivité sont créés et détruits par l'intercepteur avec le *createConnectivityManager()* et le *destroyConnectivityManager()*.

```
module rm {
  interface ResourceManager {
    ConnectivityManager
      createConnectivityManager (in Object remoteObject);
    void destroyConnectivityManager
      (in Object remoteObject);
    ConnectivityManager
      findConnectivityManager (in Object remoteObject);
  };
  interface ConnectivityManager;
};
```

4.2.2.3 Gestionnaire de connectivité

Le gestionnaire de connectivité CM manipule une connexion logique entre un client sur le terminal mobile et un objet distant sur le terminal fixe. Domint définit un mécanisme dl'hystérésis (*Cf.* figure 4.4) qui permet de gérer la connectivité entre l'objet distant et DO.

Sur la figure 4.4, quand le niveau de ressource augmente et est inférieur au seuil *lowUp* (*resp.* *highUp*), le terminal mobile est déconnecté (*resp.* partiellement connecté). Quand le niveau de ressource diminue mais est plus élevé que la valeur *highDown* (*resp.* *lowDown*), le terminal mobile est connecté (*resp.* partiellement connecté). Sans la figure 4.4-2, il peut y avoir un effet de *"ping-pong"* autour de la valeur *highDown*. Ainsi, quand la connexion arrive dans l'état F en provenance de l'état E et quand le niveau de ressource dépasse la valeur *highDown*, le mode demeure "partiellement connecté" 'jusqu'à la valeur *highUp*.

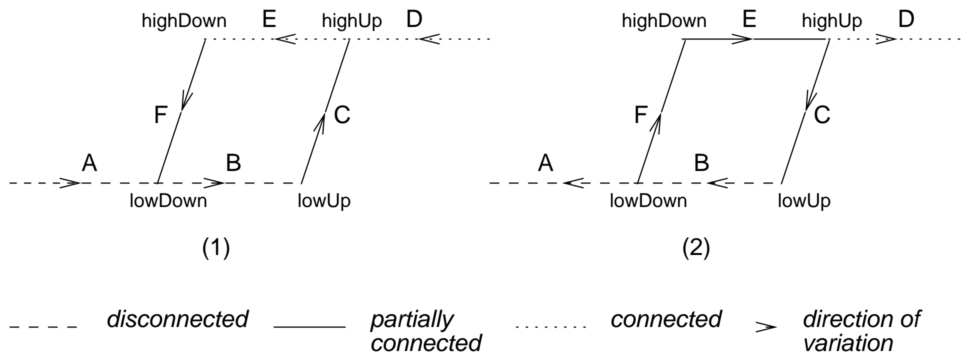

FIG. 4.4 – L'hystérésis pour la gestion de la conectivité

```
interface ConnectivityManager {
  attribute double lowDown;
  attribute double highDown;
  attribute double lowUp;
  attribute double highUp;
  readonly attribute double resLevel;
  readonly attribute char state;
  readonly attribute char mode;
  readonly attribute boolean forward;
  attribute boolean voluntaryDisc;
  readonly attribute Object remoteObject;
  readonly attribute Object localObject;
  string computeCI();
  string getConnectionStateInfo();
};
```

4.2.2.4 Gestionnaire du journal d'opérations

Le gestionnaire du journal d'opérations LM est responsable du contrôle des opérations enregistrées avec les méthodes `addLogRequest()`, `compactLog()`, `treatLog()`. Les opérations sont classées par objet distant. Le gestionnaire du journal ne peut pas interpréter les données enregistrées, mais les objets déconnectés fournissent le code qui permettra par exemple de les transmettre à l'objet distant.

```
module lm {
  abstract interface DORequest {
    void compactLog(in Object remoteObject);
    long treatLog();
  };
  interface LogManager {
    void addLogRequest(in any data);
    void compactLog(in Object remoteObject);
    long treatLog(in boolean optimistic);
    void receiveOBV(in DORequest applObject);
  };
};
```

Chapitre 5

Protocole de déploiement d'objets

Après avoir étudié les différentes approches de déploiement (*Cf.* section 3.3) et le fonctionnement général de l'architecture Domint (*Cf.* chapitre 4), nous présentons la conception de notre approche de déploiement qui sera implantée dans la plate-forme Domint. L'approche que nous proposons est une synthèse des approches étudiées dans le chapitre 3 en introduisant le concept de criticité des objets. La criticité permet de définir deux groupes d'objets : les objets critiques qui sont les objets nécessaires pour le fonctionnement des applications en mode déconnecté et les objets non critiques dont l'absence n'empêche pas le fonctionnement de l'application en mode déconnecté.

Dans ce chapitre, nous présentons le concept de criticité (*Cf.* section 5.1). Ensuite, nous répondons dans la section 5.2 à la question "quand déployer les objets ?". La section 5.3 développe les étapes à suivre pour la mise en œuvre du déploiement sur un système. Enfin, la section 5.4 décrit notre approche de déploiement dans Domint et le mécanisme de gestion du cache associé.

5.1 Concept de criticité

5.1.1 Introduction du concept

Dans la conception d'une application répartie CORBA, la première étape est la spécification du contrat sous forme d'interfaces écrites en IDL, qui permettent de définir tous les types d'objets. La plate-forme Domint décompose l'ensemble des objets de l'application en deux catégories :

1. Les objets autorisant un proxy pour le mode déconnecté : cette catégorie comporte les objets que l'application peut déployer dans le terminal mobile pour le fonctionnement en mode déconnecté. Dans Domint le choix de ces objets est fait par le programmeur de l'application. Ce choix dépend en particulier de la sémantique de l'application.
2. Les objets qui n'autorisent pas un proxy pour le mode déconnecté : contrairement à la première catégorie, cette catégorie contient les objets que l'utilisateur ne peut pas manipuler localement. De même que pour la première catégorie, le choix des objets qui n'autorisent pas un proxy pour le mode déconnecté se fait lors de la conception de l'application par le programmeur.

Comme décrit dans la section 4.2, l'objet proxy (objet déconnecté) est un objet CORBA qui est semblable à l'objet présent du côté du serveur. La première différence est que l'interface de l'objet déconnecté comporte des opérations supplémentaires ajoutées pour faire face au fonctionnement en mode déconnecté, notamment la sauvegarde d'opérations dans le journal et le transfert d'état avec l'objet distant. La deuxième différence est que le comportement des opérations communes avec

l'objet distant est différent : par exemple, le message électronique n'est pas transmis mais journalisé dans le journal d'opérations.

La figure 5.1 donne un exemple de création d'objets déconnectés dans une application de messagerie électronique. Les rectangles en pointillés représentent deux clients mobiles (A et B) et un serveur fixe qui contient les objets distants. L'application comporte deux objets. Le premier type est le *MailBoxManager* qui permet de créer, supprimer, renommer et déplacer les boîtes aux lettres. Cet objet n'est manipulable que par l'administrateur de l'application. Le deuxième type d'objet est le *MailBox*. Dans cette application, un objet de type *MailBox* est créé par utilisateur. La création s'effectue à partir de l'objet de type *MailBoxManager*. Dans la figure 5.1, l'objet de type *MailBox* est manipulé par le client à travers une interface graphique GUI.

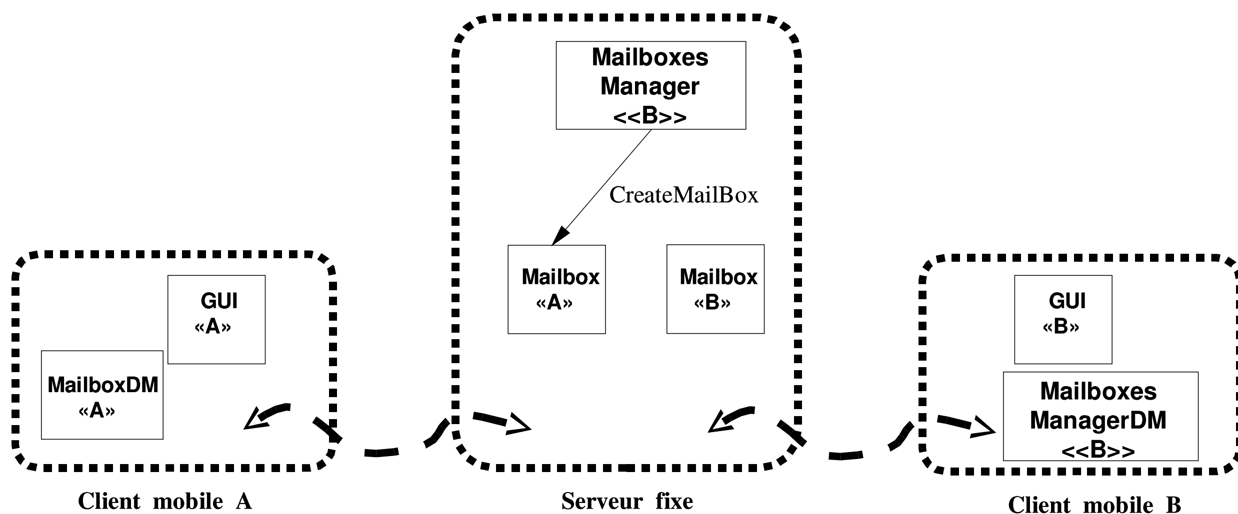

FIG. 5.1 – Exemple de création des objets déconnectés

Dans cette application, supposons que le client mobile B crée dans son terminal mobile un objet déconnecté *MailBoxManagerDM* proxy de l'objet *MailBoxManager* du serveur fixe. Dans le scénario où le client B a le droit de manipuler cet objet et crée une nouvelle boîte aux lettres pour un utilisateur C à partir de l'objet *MailBoxManagerDM*, l'utilisateur C ne pourra pas utiliser sa boîte aux lettres tant que le client B est déconnecté. Donc, dans ce cas de figure, l'objet *MailBoxManager* doit être classé par le programmeur de l'application comme étant un objet qui n'autorise pas le fonctionnement en mode déconnecté. Par contre, l'objet *MailBox* du serveur fixe peut avoir un objet déconnecté, puisque l'utilisation de cet objet par l'utilisateur en mode déconnecté permet quand même à l'objet resté sur le serveur fixe de recevoir les messages à destination de l'utilisateur. Cette utilisateur verra ses messages lors de la reconnexion. Donc, c'est à partir de la sémantique et du fonctionnement de l'application que le programmeur décide quels sont les objets qui autorisent la création d'un objet déconnecté pour le mode déconnecté. En conclusion, contrairement à Coda ou Rover, Domint ne traite pas tous les objets de l'application de la même manière et l'architecture de l'application est fortement liée à la sémantique de l'application.

Le problème avec cette décomposition est qu'elle ne traite pas le problème des objets proxy qui doivent absolument exister dans le terminal mobile pour le fonctionnement en mode déconnecté. Notre approche de déploiement commence par une réponse à ce problème avec l'introduction du concept de criticité. Le concept de criticité permet de donner un "poids" aux objets quand à leur présence dans le cache du terminal mobile. Ce concept nous amène à classer les objets qui autorisent

un proxy pour le mode déconnecté. Cette classification est présentée dans la figure 5.2. La catégorie des objets qui autorisent un proxy pour le mode déconnecté est partitionnée en deux classes : les objets critiques (OC) et les objets non critiques (ONC).

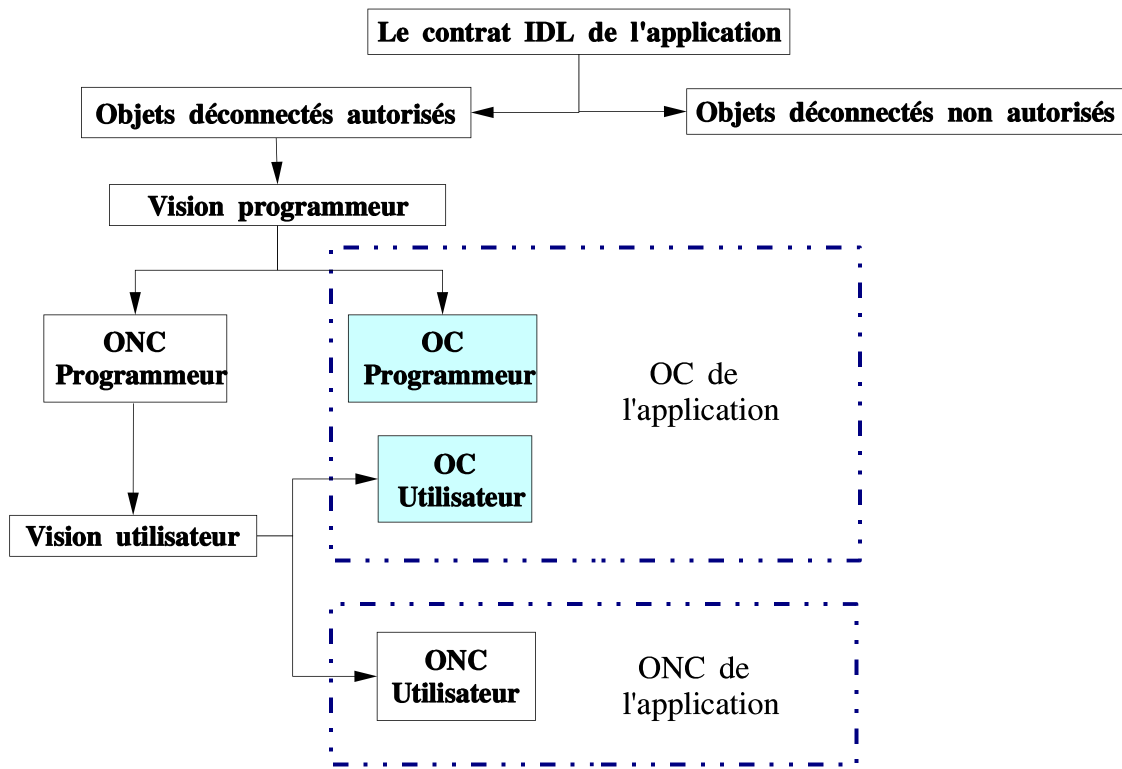

FIG. 5.2 – La criticité des objets

5.1.2 Objets critiques et Objets non critiques

Un objet critique est un objet dont la présence dans le cache est obligatoire pour le fonctionnement en mode déconnecté. L'utilisateur doit avoir un objet proxy pour le mode déconnecté dans son cache. Un objet non critique est un objet dont l'absence dans le cache ne peut pas empêcher le fonctionnement de l'application en mode déconnecté. L'application doit être construite de telle façon qu'un tel objet absent pendant une déconnexion n'empêche pas l'application de continuer à fonctionner. Cette classification doit être respectée dans la conception de l'application.

La conception orientée objets se base sur l'instanciation des objets à partir d'une classe. Nous envisageons deux méthodes possibles pour la classification de ces objets. La première méthode est de classifier à partir des classes d'objets. Dans ce cas, nous parlons d'attribut de classe : tous les objets instanciés à partir de cette classe possèdent la même criticité. Cette classe peut être une super-classe (classe de base) ou une classe héritée à partir d'autres classes. Cette solution pose un problème d'héritage de l'attribut `criticité` entre les différentes classes. Dans la figure 5.3, la classe *Classe 1* est déclarée comme étant une classe qui génère des objets critiques (marquée par une étoile dans la figure). La classe *Classe 3* hérite de la classe *Classe 1*. Donc, un problème se pose avec cet héritage, notamment avec la possibilité de propager l'attribut `criticité` entre les classes. La deuxième

méthode est la classification à partir des objets. Dans ce cas, nous parlons d'attribut d'instance : la criticité de l'objet est attribuée au moment de son instanciation.

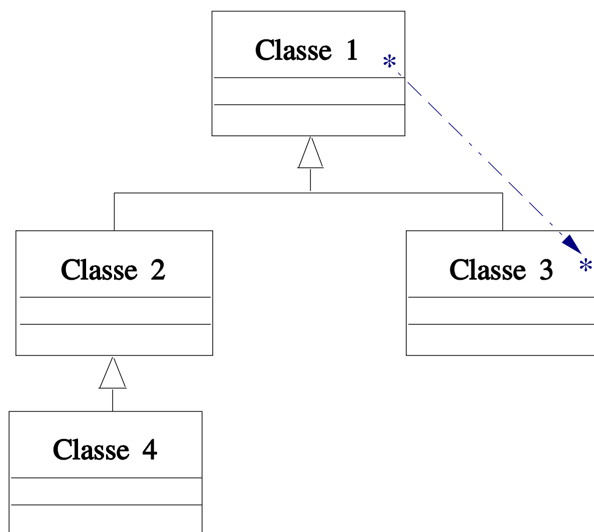

FIG. 5.3 – La propagation de la criticité entre classes

Pour déterminer qui partitionne les objets, nous envisageons deux solutions possibles. Premièrement, le partitionnement est la tâche du programmeur de l'application. Dans ce cas, le partitionnement se fait à partir des classes et/ou des objets de l'application par le programmeur. Deuxièmement, le partitionnement se fait en collaboration entre le programmeur et l'utilisateur de l'application. Cette solution est illustrée dans la figure 5.2. Au moment de la conception de l'application, le programmeur partitionne les classes et les objets qui autorisent un proxy pour le mode déconnecté en OC programmeur et ONC programmeur . Ensuite, à chaque lancement de l'application ou dans la configuration de l'application au moment de l'installation, l'utilisateur peut forcer un objet non critique du programmeur à devenir critique. Dans la figure 5.2, les rectangles grisés représentent les objets critiques de l'application, c'est-à-dire l'union des objets critiques programmeur et des objets critiques utilisateur. En outre, l'utilisateur ne peut pas forcer un objet critique programmeur à devenir un objet non critique utilisateur. Nous considérons que le programmeur connaît mieux que l'utilisateur quels sont les objets déconnectés qui doivent impérativement exister dans le terminal mobile pour le fonctionnement en mode déconnecté.

5.1.3 Exemple de partitionnement des objets

Afin de mieux comprendre notre concept de criticité, nous donnons dans cette section un exemple de classification des objets qui se basent sur la sémantique de l'application. L'exemple que nous traitons est l'application de messagerie électronique décrite dans la section 5.1.1 avec les deux classes *MailBox* et *MailBoxManager*. Cette fois-ci, nous ajoutons à notre application quatre autres objets :

- *CarnetAdressePersonnel* qui donne les adresses que le client utilise pour envoyer des messages. Cet objet offre une interface permettant au client d'ajouter, de supprimer et de récupérer des adresses de destinataires.
- *DossierMsgSupprime* qui contient tous les messages supprimés par le client de son *MailBox*. À partir de cet objet, le client peut récupérer un message qu'il a supprimé de sa boîte aux lettres.

– *DossierModel* qui contient un ensemble de messages pré-définis que le client peut utiliser pour éviter de réécrire les mêmes messages plusieurs fois. À partir de cet objet, l'utilisateur peut récupérer, ajouter ou supprimer des modèles de messages.
– *DossierMsgEnvoye* qui sauvegarde les messages envoyés par le client. Ces messages ne sont sauvegardés que lorsque le client sélectionne l'option correspondante. Le client peut lire, supprimer ou renvoyer un message de ce dossier vers d'autres destinataires.

Les objets *DossierMsgSupprime*, *DossierModel* et *DossierMsgEnvoye* sont instanciés de la même classe *Dossier*. La table 5.1 représente un partitionnement des objets. Cette table comporte quatre colonnes. La première colonne "classe" représente les classes utilisées par l'application. La deuxième colonne "objet" liste les objets de l'application. La troisième colonne "autorisant le mode déconnecté" donne la classification, par le programmeur, des objets en objets autorisant un proxy en mode déconnecté ("oui" dans la table) et ceux qui n'autorisent pas un proxy pour le mode déconnecté ("non" dans la table). La quatrième colonne"criticité" indique la criticité de chaque objet de l'application. Cette colonne est décomposée en trois sous-colonnes :
– "Choix P" : criticité des objets de l'application vue par le programmeur.
– "Choix U" : criticité vue par l'utilisateur sur les objets non critiques du programmeur.
– "Choix final" : union du choix du programmeur et de l'utilisateur.

Classe	Objet	Autorisant le mode déconnecté	Criticité		
			Choix P	Choix U	Choix final
MailBox	BoîtMessage	oui	OC	–	OC
MailBoxManager	Gestionnaire	non			
CarnetAdresse	CarnetAdressePersonnel	oui	ONC	**OC**	**OC**
Dossier	DossierMsgSupprime	oui	ONC	ONC	ONC
	DossierModel			ONC	ONC
	DossierMsgEnvoye			ONC	ONC

TAB. 5.1 – Un exemple de partitionnement d'objets pour l'application messagerie électronique

L'objet *Gestionnaire* instancié de la classe *MailBoxManager* est un objet qui n'autorise pas un proxy dans le terminal mobile en mode déconnecté. Par contre, les objets *BoîtMessage*, *CarnetAdressePersonnel*, *DossierMsgSupprime*, *DossierModel* et *DossierMsgEnvoye* autorisent un proxy pour le mode déconnecté. Puisque, la manipulation locale de ces objets n'affecte pas le fonctionnement de l'application. Dans la table 5.1, les quatre derniers objets sont classés comme étant des objets non critiques par le programmeur par le fait qu'ils sont instanciés de la classe *Dossier* considérée par le programmeur comme étant une classe qui génère des objets non critiques. Par contre, le programmeur classe l'objet *BoîtMessage* dans la catégorie des objets critiques. Donc, la présence de l'objet déconnecté qui lui correspond dans le terminal mobile est obligatoire pour le fonctionnement de l'application en mode déconnecté. Par ailleurs, l'utilisateur lance son application de messagerie électronique sur son terminal mobile et s'aperçoit qu'il a le droit de modifier la criticité des objets *CarnetAdressePersonnel*, *DossierMsgSupprime*, *DossierModel* et *DossierMsgEnvoye*. L'utilisateur pense qu'il va envoyer plusieurs messages à des destinataires différents ou qu'il ne connaît pas par cœur leur adresse. Par conséquent, il force la criticité de l'objet *CarnetAdressePersonnel*.

5.2 Déploiement des objets critiques et non critiques

Après avoir présenter notre vision du classement des objets dans une application répartie, nous allons décrire notre approche de déploiement qui se base sur la notion de criticité d'objet.

5.2.1 Déploiement suivant la criticité des objets

La figure 5.4 présente notre approche de déploiement. Cette approche dépend de la nature des objets à déployer. D'après la figure 5.4, pour les objets critiques, le déploiement se fait au lancement de l'application. Contrairement à Coda qui utilise le déploiement au lancement de l'application, les objets à déployer peuvent ne pas être instanciés (présents en mémoir) dans le serveur. Par exemple, une application qui utilise un activateur des serviteurs (*Servant Activator* en anglais) [GAK00], l'objet serveur est créé à la première invocation d'un client à cet objet. L'activation des objets au moment de l'invocation ne pose pas de problèmes pour le déploiement, puisque l'objet déconnecté est créé indépendamment de l'objet distant. Une fois l'objet déconnecté est créé, il doit faire un transfert d'état avec l'objet distant. Donc, l'objet distant va être activé par la requête du transfert d'état.

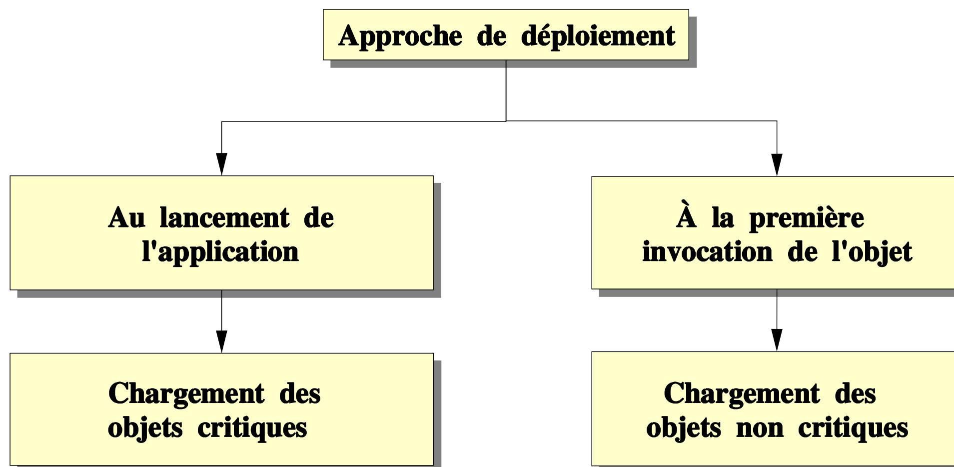

FIG. 5.4 – Le déploiement d'objets

Pour les objets non critiques, la création des objets déconnectés se fait à la première invocation. Quand l'application invoque un objet non critique, le système vérifie si l'objet existe dans le cache. Si ce n'est pas le cas, le système crée automatiquement un objet proxy dans le cache.

5.2.2 Problème de connectivité

Notre approche se base sur l'hypothèse que le déploiement ne se fait que dans le mode fortement connecté. Or, le problème avec cette hypothèse est que le déploiement peut être interrompu si la connectivité passe au mode faiblement connecté voir déconnecté. Nous prévoyons deux cas de figure à ce problème selon que l'objet invoqué est critique ou non.

Premièrement, supposons qu'il y a une déconnexion involontaire au moment du chargement des objets critiques. La première solution à ce problème est d'attendre une bonne connectivité pour terminer le chargement des objets critiques. Cette solution est en contradiction avec notre approche qui se base sur le principe qu'un objet critique doit être présent dans le cache. Pour éviter ce problème,

nous proposons une deuxième solution où nous avons décider de créer des objets déconnectés sans états ("objets déconnectés vides") pour les objets non déployés. Un objet déconnecté vide est tout simplement un objet qui respecte l'interface de l'objet déconnecté mais sans faire un transfert d'état avec l'objet distant. Une conséquence à cette solution est que l'utilisateur ne verra pas l'état actuel de ces objets. Enfin, pour éviter les déconnexions volontaires au moment du chargement des objets critiques, nous proposons d'interdire les déconnexions volontaires au moment du chargement des objets critiques.

Deuxièmement, supposons que la première invocation sur un objet non critique survient dans le mode déconnecté ou partiellement connecté. Contrairement au premier cas où nous créons un objet déconnecté vide, nous utilisons la même approche que Rover qui sauvegarde les requêtes dans un journal d'opérations. Au passage au mode connecté ou partiellement connecté, la requête est renvoyée vers l'objet serveur. Dans ce cas, nous évitons la création d'un objet déconnecté vide parce que la requête de l'application peut être destinée à modifier l'état de l'objet ou à récupérer des données.

5.3 Étapes de conception

Le figure 5.5 représente les deux problèmes à traiter pour la mise en œuvre de notre approche de déploiement. Premièrement, au moment de la programmation de l'application, le programmeur doit choisir s'il propage l'attribut «criticité» entres les différentes classes de l'application. Deuxièmement, au moment de la configuration de l'application sur le choix des types d'objets, nous envisageons deux visions possibles :

- Une vision programmeur où le partitionnement des objets se fait au moment de la programmation. Ce partitionnement est statique dans le sens où la criticité de l'objet ne peut pas être modifiée une fois qu'elle est fixée par le programmeur.
- Une vision programmeur plus utilisateur dans le sens où l'utilisateur peut changer la criticité d'un objets.

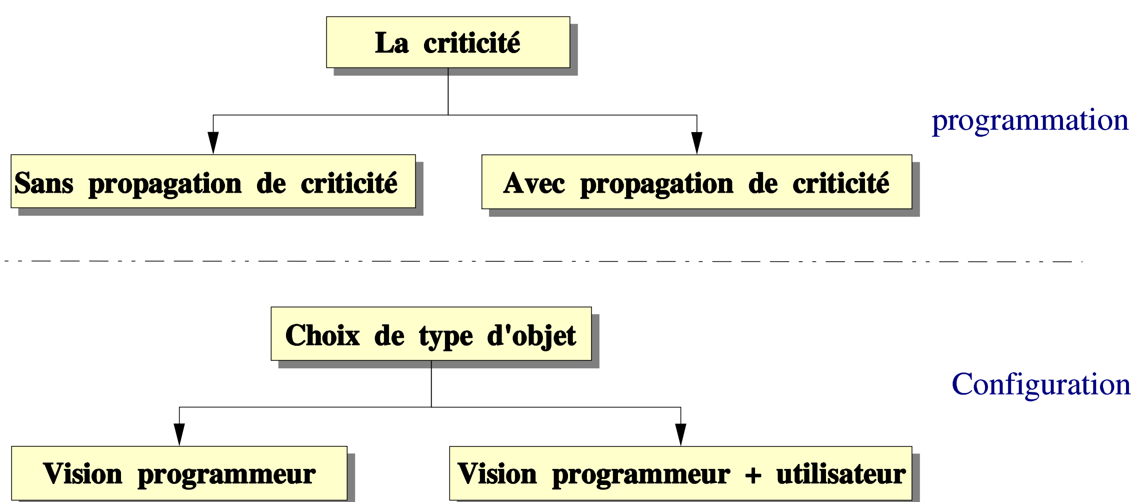

FIG. 5.5 – Les étapes de conception

5.4 Déploiement dans Domint

L'architecture de déploiement que nous avons étudiée jusqu'à maintenant n'est liée à aucune plate-forme. Dans cette section, nous présentons le fonctionnement de notre approche dans la plate-forme Domint. Nous traitons le problème de partitionnement des objets en objets critiques est objets non critiques (*Cf.* section 5.4.1). Ensuite, dans la section 5.4.2 nous détaillons notre mécanisme de gestion du cache.

5.4.1 Architecture

L'architecture de déploiement dans Domint est représentée dans la figure 5.6. Cette figure présente un diagramme de collaboration UML, il décrit l'algorithme de déploiement des objets critiques présenté dans la figure 5.7. Les information sur la criticité des objets sont contenus dans un fichier de configuration. Chaque entrée de ce fichier comporte deux champs :

1. **Nom logique**
   Le champ "nom logique" représente la référence de l'objet dans le système. Dans le contexte CORBA, le nom logique de l'objet peut être récupéré à partir du service de nommage ou à partir d'un fichier de désignation. Ce champ est rempli par le programmeur de l'application qui connaît comment trouver la référence de chaque objets.
2. **Type**
   Le champ "type" représente la criticité de l'objet. Dans le cas où l'objet est critique, ce champ est égal à "OC". Dans le cas contraire, ce champ est égal à "ONC".

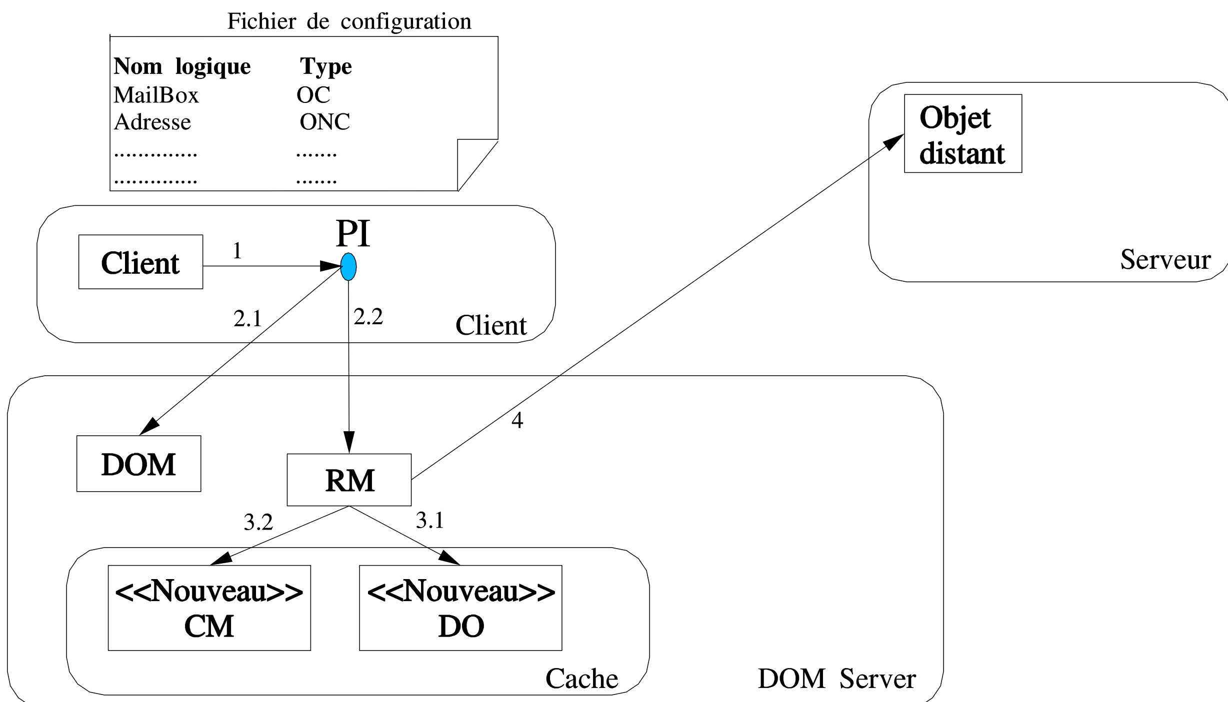

FIG. 5.6 – Le déploiement dans Domint au lancement de l'application

Le fichier de configuration est construit par le programmeur de l'application durant la phase de programmation. Au lancement de l'application ou au moment de la configuration de l'application,

l'utilisateur peut modifier ce fichier pour choisir parmis les objets non critiques du programmeur ceux qui deviennent critiques.

Au lancement de l'application et après la mise à jour du fichier de configuration par le client (Client dans la figure 5.6), l'intercepteur portable (PI) demande à DOM (2.1) la référence de RM. Nous rappelons que l'unité d'exécution de DOM n'est pas la même que celle du client. Puisque le client et le DOM se trouvent dans la même machine et selon l'hypothèse que l'application cliente connaît le port d'entrée pour trouver le DOMServer, PI peut construire la référence de DOM à partir de ces informations.

Une fois la référence de RM obtenue, PI invoque l'objet RM (2.2) via la méthode `Deployer()` présentée dans la figure 5.7 en lui passant l'adresse du fichier de configuration. Dans cet algorithme, RM parcourt le fichier de configuration, et pour chaque entrée, effectue les opérations suivantes. D'abord, il vérifie le type de l'objet à travers le champ "type". Si l'objet est un objet critique, RM crée l'objet déconnecté DO de cet objet et le gestionnaire de connectivité CM associé (3.1, 3.2) de la même manière que dans Domint. Ensuite, RM invoque l'objet distant (4) pour faire un transfert d'état. Si la création de l'objet déconnecté échoue suite à une défaillance de la connexion, l'objet déconnecté reste vide. Le transfert d'état entre l'objet serveur et l'objet déconnecté se fait quand la connexion redevient bonne. Ensuite, DO essaie périodiquement de contacter l'objet distant pour faire un transfert d'état.

```
Algorithme Deployer(chaine fichierConf) {
  file  fichier;
  objet do;
  info  objet;
  file = lireFichier(fichierConf);
  Tant que (non fin de fichier(file))  {
     info = recupererInfoTable();
     nomObjet = info.nom;
     typeObjet = info.type;
     Si (typeObject == "OC")   {
         do = creerobjetDeconnecteEtCM(nomObjet);
         Si (do != null)  {
         do.incrementalStateTransfer();
         }
      }
  }
}
```

FIG. 5.7 – L'algorithme de Déploiement des objets critiques

Si le type de l'objet est "non critique", RM ne fait rien puisque le déploiement de ces objets se fait à la première invocation. Lors de l'interception d'une requête, PI recherche la référence du CM associé à l'objet distant dans sa table interne. S'il ne la trouve pas, il conclut que c'est la première invocation pour cet objet distant. Dans ce cas, PI appelle RM via la méthode `createConnectivityManager()` pour que ce dernier crée l'objet déconnecté DO et le gestionnaire de connectivité CM associé dans le cache. L'objet déconnecté que RM crée dans le terminal mobile respecte l'interface donné dans la figure 5.8. la méthode `incrementalStateTransfer()` permet de faire le transfert d'état avec l'objet distant, tandis

que les méthodes `disconnect()` et `reconnect()` sont utilisées pour commuter entre l'objet distant et l'objet déconnecté dans le cas de la déconnexion volontaire.

```
interface ObjetDM {
//les mêmes méthodes que l'objet distant
.......................
.......................
.......................
//ajouter pour gérer la déconnexion
void incrementalStateTransfer();
void disconnect();
void reconnect();
}
```

FIG. 5.8 – L'interface de l'objet déconnecté

5.4.2 Gestion du cache

Pour résoudre le problème d'encombrement du cache, nous décrivons dans cette section le mécanisme de gestion du cache que nous proposons pour Domint (*Cf.* section 5.4.2.2) après avoir présenté quelques algorithmes classiques de la littérature (*Cf.* section 5.4.2.1).

5.4.2.1 Algorithmes de gestion du cache

De nombreuses recherches ont été consacrées à la gestion de la mémoire virtuelle. Entre autres aspects, la majorité de ces recherches se sont intéressées aux algorithmes de remplacement ou de suppression des données. Si nous considérons que la taille du cache est très inférieure à la taille des données accessibles (généralement c'est le cas de figure), il devient essentiel de ne conserver en cache que les objets les plus utilisés. Il est donc vital pour la performance du cache de choisir le meilleur algorithme de remplassement possible. Parmi les algorithmes de la littérature les plus utilisés dans la gestion du cache nous citons les algorithmes suivants [SG98] :

- FIFO (*First in, First Out*) : c'est l'algorithme le plus simple qui soit. Les objets sont expulsés du cache dans le même ordre qu'ils y sont entrés. Cependant, cet algorithme n'est pas performant : un objet jugé utile par l'utilisateur devrait avoir une chance de rester longtemps en cache, tandis qu'un objet peu utile devrait être expulsée rapidement. L'algorithme FIFO, qui ne tient pas compte de la popularité des objets, ne répond pas à ce cahier des charges.
- LRU (*Least Recently Used*) : dans cette algorithme, chaque fois qu'un objet doit être expulsé, l'algorithme choisit l'objet qui n'a pas été accédé depuis le plus longtemps. Cette politique est nettement plus efficace que FIFO car elle permet aux objets de rester dans le cache tant qu'ils sont très utiles. Cependant, LRU est plus compliqué à mettre en œuvre que FIFO.
- LFU (*Least Frequently Used*) : cet algorithme consiste à expulser l'objet qui a été accédé avec la plus faible fréquence. Cet algorithme est plus compliqué à implanter que LRU. Son inconvénient est qu'il a tendance à ne rien oublier : si un objet est extrêmement utile pendant un moment, puis cesse brusquement d'être utilisé, il restera en cache longtemps après. Une variante qui répond à ce problème est d'utiliser un paramètre de vieillissement des objets dans le cache.

5.4.2.2 Remplacement des objets non critiques

Dans notre approche, il est évident que les objets à choisir pour le remplacement appartiennent au groupe des objets non critiques, puisque le principe de notre approche de déploiement se base sur l'hypothèse qu'un objet critique doit toujours être présent dans le cache. Dans notre approche, nous supposons que la taille du cache est supérieure à la somme des tailles des objets critiques. Cette hypothèse nous assure que l'existence de tous les objets critiques dans le cache ne génère pas un encombrement du cache.

L'approche de gestion du cache que nous proposons pour la gestion des objets non critiques se base sur l'algorithme LFU. Cette approche est représentée dans la figure 5.9.

Notre approche définit deux nouveaux paramètres spécifiés par l'utilisateur qui sont le seuil de disponibilité du cache et la période sur laquelle l'algorithme LFU calcule la fréquence d'utilisation de chaque objet. Le seuil de disponibilité du cache est défini comme étant la taille minimale disponible du cache au dessous de laquelle RM doit faire des suppressions d'objets dans le cache. La période permet de définir la périodicité du calcul de la fréquence d'utilisation de chaque objet. Par conséquence, dans notre cas, la fréquence d'utilisation de chaque objet est égale au nombre d'invocations pendant la dernière période divisée par la période

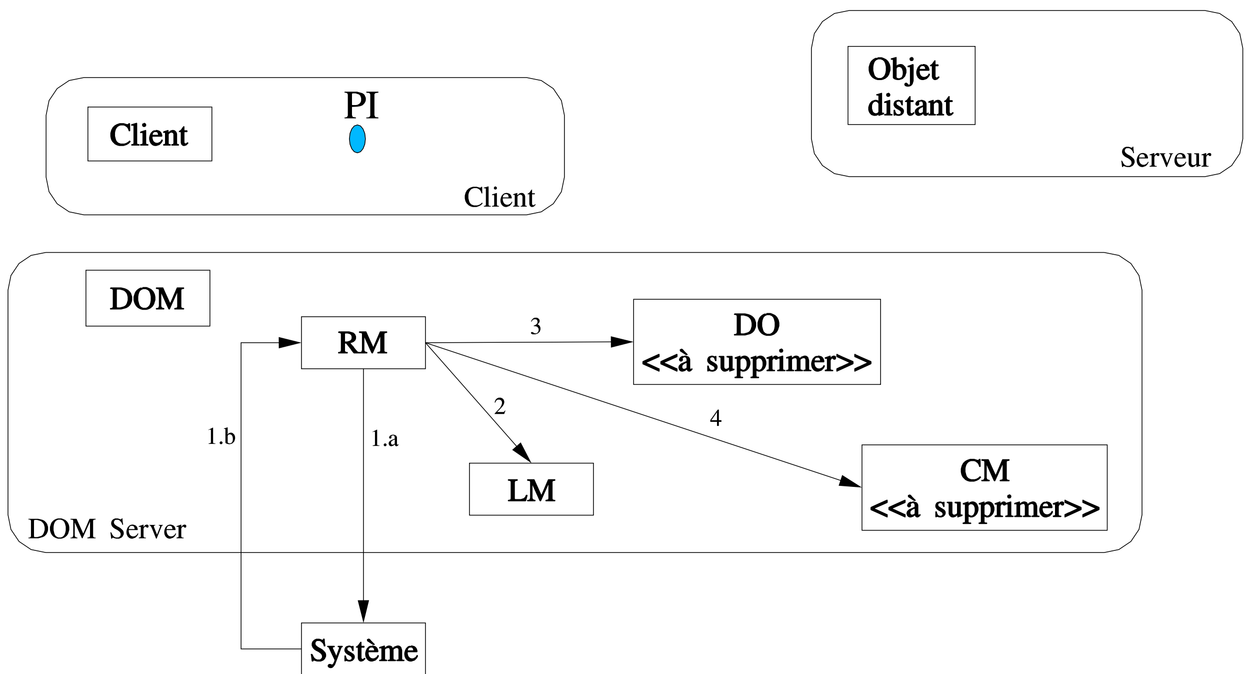

FIG. 5.9 – La gestion du cache dans Domint

La table 5.2 représente un exemple d'instants d'invocation des objets *CarnetAdressePersonnel* et *DossierModel* dans l'application de messagerie électronique vue dans la section 5.1.3. La première colonne de cette table donne les deux objets étudiés. La deuxième colonne donne les instants d'invocation pour chaque objet. Dans notre exemple, nous supposons que l'heure d'invocation est donnée en seconde. Par exemple, l'objet *CarnetAdressePersonnel* est invoqué aux instants 2, 3,...., 26 et 29. Si nous appliquons l'algorithme LFU sur cet exemple, l'objet *CarnetAdressePersonnel* est invoqué dix fois et l'objet *DossierModel* est invoqué onze fois. Par conséquence, c'est l'objet *CarnetAdressePersonnel* qui va être choisi dans l'algorithme pour le remplacement. Or, cet objet est plus fréquemment accédé que l'objet *DossierModel* dans les dix dernières secondes. Dans l'algo-

rithme que nous proposons,si la période est par exemple ègale à dix secondes, le calcul du nombre d'invocations pour chaque objet commence à partir de la vingtième seconde. Avec cet algorithme, la fréquence de l'objet *DossierModel* est zéro et celle de *CarnetAdressePersonnel* est cinq. Par conséquence, c'est l'objet *DossierModel* qui doit être choisi par l'algorithme pour le remplacement.

Objet	Instant d'invocation
CarnetAdressePersonnel	2 , 3 , 7 , 9 , 12 , 21 , 23 , 25 , 26 , 29
DossierModel	1 , 2 , 4 , 6 , 10 , 13 , 15 , 16 , 17 , 18 , 19

TAB. 5.2 – Un exemple de séquences d'invocations des objet *DossierModel* et *CarnetAdressePersonnel*

Nous envisageons deux solutions possibles pour concevoir notre algorithme de gestion du cache. La première solution est que RM dispose d'une table interne qui permet d'enregistrer le nombre d'invocations du client vers chaque objet non critique. Cette fonctionnalité est implantée dans la méthode `incrementeNumber()` ajoutée dans l'interface *ResourceManager*. Cette méthode est appelée par PI à chaque invocation d'un objet non critique. Lorsqu'un client invoque un objet non critique, RM incrémente le nombre d'invocations de cet objet dans sa table interne. Pour que le problème de la première invocation d'un objet non critique n'apparaisse pas, la création de l'objet déconnecté et l'appel de la méthode `incrementeNumber()` sont faits après que RM ajoute une entrée pour cet objet dans sa table interne. Le nombre d'invocations d'un nouvel objet dans le cache est initialisé à zéro. Périodiquement, RM ordonne cette table suivant la fréquence d'invocations sur les objets (du moins invoqué vers le plus invoqué pendant la dernière période).

Quand RM découvre que la taille restant du cache est inférieure au seuil de disponibilité, il prend la première entrée de sa table interne et contrôle si l'objet sélectionné ne comporte pas des requêtes sauvegardées dans le journal d'opérations (2) dans la figure 5.9. Dans le cas où le journal d'opérations ne contient pas d'entrées spécifiques à l'objet sélectionné, RM supprime du cache le DO et le CM (3, 4). RM exécute la même procédure avec la prochaine entrée de la table interne jusqu'à ce que la taille du cache restante devienne supérieure au seuil de disponibilité.

L'inconvénient de cette solution est qu'elle génère un nombre très important de requêtes, puisqu'à chaque fois que le client invoque un objet non critique l'intercepteur portable invoque RM pour que ce dernier mette à jour sa table interne.

La deuxième solution que nous proposons est de mettre la table d'invocations dans l'intercepteur portable. Dans cette solution, PI met à jour la table d'invocations à chaque fois que le client invoque un objet non critique. Quand RM s'aperçoit que la taille du cache restante est inférieure au seuil, il demande à PI de lui fournir la table d'invocations. Quand RM reçoit la table d'invocations, il effectue le même traitement que dans la première solution. Cette solution pose le problème d'invocation de PI, puisque PI est un objet CORBA local qui ne peut pas être invoqué de l'extérieur de son unité d'exécution. Nous proposons deux solutions à ce problème. La première solution présentée dans la figure 5.10 est de créer un objet corba (non local) PITable dans l'unité d'exécution de PI qui comporte une table interne d'invocation. Chaque fois que PI intercepte un invocation vers un objet non critique, il transmet les références (nom de l'objet, heure d'invocation) de cette requête à PITable. Si RM s'aperçoit que la taille du cache restante est inférieure au seuil de disponibilité, il demande la table d'invocation de PITable. La deuxième solution est de créer un socket TCP ou UDP dans PI pour qu'il puisse être invoqué par RM. Nous préférons a priori la solution à base d'objet CORBA.

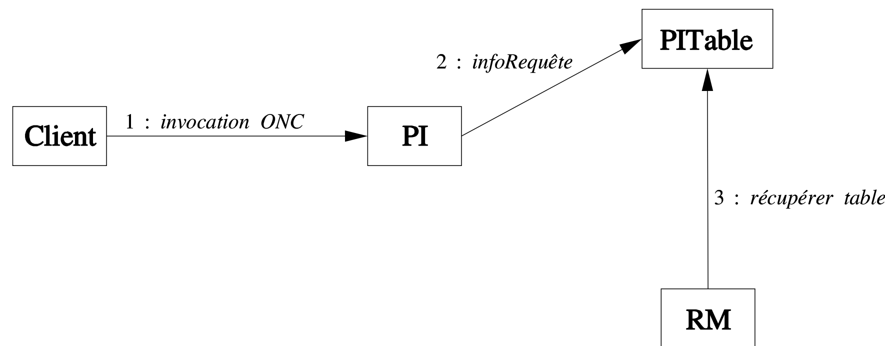

FIG. 5.10 – La création d'un objet intermédiaire entre PI et RM

Pour que RM sache que la taille du cache est inférieure au seuil, nous envisageons deux solutions. La première solution (1.a dans la figure 5.9) consiste à utiliser des notifications du système (*UpCall* en anglais). Dans cette solution, le système doit notifier RM à chaque fois que la taille restante dans le cache est inférieure au seuil de disponibilité. Cependant, cette solution est très difficile à mettre en œuvre puisqu'elle demande beaucoup de programmation système. En outre, la plupart des systèmes n'offrent pas ce type de notification et si un système offre cette notification, cela reste très limités : par exemple, les signaux dans UNIX sont des notifications au plus une fois et n'acceptant pas d'argument. La deuxième solution possible est que RM interroge le système périodiquement pour récupérer la taille du cache restante (1.b dans la figure 5.9). Cette solution est plus simple à implanter que la première solution et c'est la solution que nous choisissons dans notre approche de gestion du cache. L'algorithme de gestion du cache est donné dans la figure 5.11. Dans le cas où le journal d'opérations contient des entrées spécifiques à l'objet sélectionné, RM passe à l'entrée suivante jusqu'à ce qu'il trouve un objet qui ne comporte pas de requêtes sauvegardées dans le journal d'opérations. Le problème est que le journal d'opérations peut contenir des entrées pour tous les objets déconnectés. En conséquence, d'après l'algorithme de la figure 5.11, un bouclage à l'infini sur la table interne du RM peut avoir lieu. Dans Domint, LM essaie périodiquement de vider le journal d'opérations (*Cf.* section 4.2). Donc, la procédure de traitement du journal d'opérations converge vers la transmission de toutes les requêtes sauvegardées dans le journal d'opérations. Par conséquent, RM tend à trouver un objet qui ne comporte pas d'opérations sauvegardées dans le journal d'opérations.

5.5 Discussion

La méthode de déploiement proposée dans ce chapitre peut se résumer en deux points. D'abord, nous avons développé un nouveau concept qui est la criticité des objets dans une application répartie et les différentes algorithmes de déploiement de chaque catégorie d'objets. Ensuite, nous avons proposé un algorithme de gestion du cache. Un point fort de notre approche de déploiement est que, à tout moment, le cache du terminal mobile contient au minimum tous les objets critiques.

Par rapport à Coda qui utilise une approche de pré-chargement par le client pour le déploiement, dans notre approche, le choix des objets à déployer sur le terminal mobile se fait entre le programmeur de l'application et le client. En conséquence, dans Coda, l'application cliente en mode déconnecté peut ne pas fonctionner si le client fait un mauvais choix.

En outre, notre approche de partitionnement des objets de l'application en objets critiques et objets non critiques est très similaire à celle de Coda qui partitionne les fichiers de l'utilisateur

```
Algoritme remplacerobjetNonCritique() {
  réel taillerestanteCache;
  objetCible objet;
  boolean aucun;
  taillerestanteCache = Système.tailleRestante(cache);
   tant que (taille != seuil)  {
    objetCible = tableInterne.suivant();
    objet = objetCible.nom;
    aucun = LM.verifierExistenceOperationJournalisee(objet);
            Si (aucun == vrai) {
             // pas d'entrées dans le log pour cet objet
             SupprimerDO-et-CM-associé();
            }
            Sinon {  attentePassivement(durée);
                       }
          }
    }
}
```

FIG. 5.11 – L'algorithme de gestion des objets non critiques

en données explicites qui représentent le choix du client et données implicites qui se composent de l'historique du client. Le déploiement des données implicites se fait suivant un algorithme de chargement (les plus récemment utilisés). Donc, un fichier non choisi par l'utilisateur peut ne pas exister dans le cache même si le client utilise ce fichier. Par contre, dans notre approche, si un client invoque un objet non critique, un objet déconnecté est créé automatiquement dans le cache. Donc si le client se déconnecte juste après l'invocation de cet objet, il peut faire des traitements sur l'objet déconnecté créé dans son cache.

Un autre point important dans notre approche est que lorsqu'un client mobile se déconnecte volontairement au lancement de l'application, il peut travailler puisque son cache contient au minimum tous les objets critiques. Dans Rover et Coda, si un client se déconnecte volontairement au lancement de l'application, il y a peu de chance que ce client puisse travailler en mode déconnecté.

L'inconvénient de notre approche de déploiement est que le déploiement de tous les objets critiques au lancement de l'application peut prendre du temps et est conditionné par la taille du cache.

Notre protocole de déploiement peut être dans une situation où tous les objets de l'application existent dans le cache, en particulier, si l'application invoque tous les objets non critiques ou si l'utilisateur choisit tous les objets comme étant des objets critiques. Cette situation génère le problème d'encombrement du cache, notamment avec les terminaux mobiles. Pour régler ce problème, nous avons proposée un protocole de gestion de cache qui est présenté dans la section 5.4.2. L'avantage principale de ce protocole est qu'il ne supprime que les objets non utilisés dernièrement par le client. De plus, les objets que l'algorithme peut supprimer sont les objets non critiques de l'application. L'inconvénient majeur est que la taille du cache doit être inférieure à la somme de toutes les tailles des objets critiques. Or, cette hypothèse ne tient pas compte de la croissance des objets critiques.

Chapitre 6

Réalisation

Dans ce chapitre, nous illustrons et validons l'approche de déploiement et de gestion du cache que nous avons étudiée dans le chapitre 5. Cette validation se fait dans le cadre d'une application de messagerie électronique dans un environnement sans fil.

6.1 Prototype de réalisation

La réalisation est effectuée sur la plate-forme Domint.Dans cette réalisation, nous utilisons l'ORB ORBacus 4.1.0 de O.O.C. (*Object Oriented Concept*). Il est conforme à la norme CORBA 2.4 qui fournit plusieurs mécanismes, en particulier, les intercepteurs portables. En outre, nous utilisons le langage JAVA.

Cette réalisation respecte l'architecture présentée dans la figure 6.1. Dans cette architecture, chaque terminal mobile dispose d'un service DOM (gestionnaire des objets déconnectés) qui permet de gérer les déconnexions de l'utilisateur et faire la commutation transparente entre l'objet distant et l'objet déconnecté, suivant le niveau de connectivité donné par l'hystérésis.

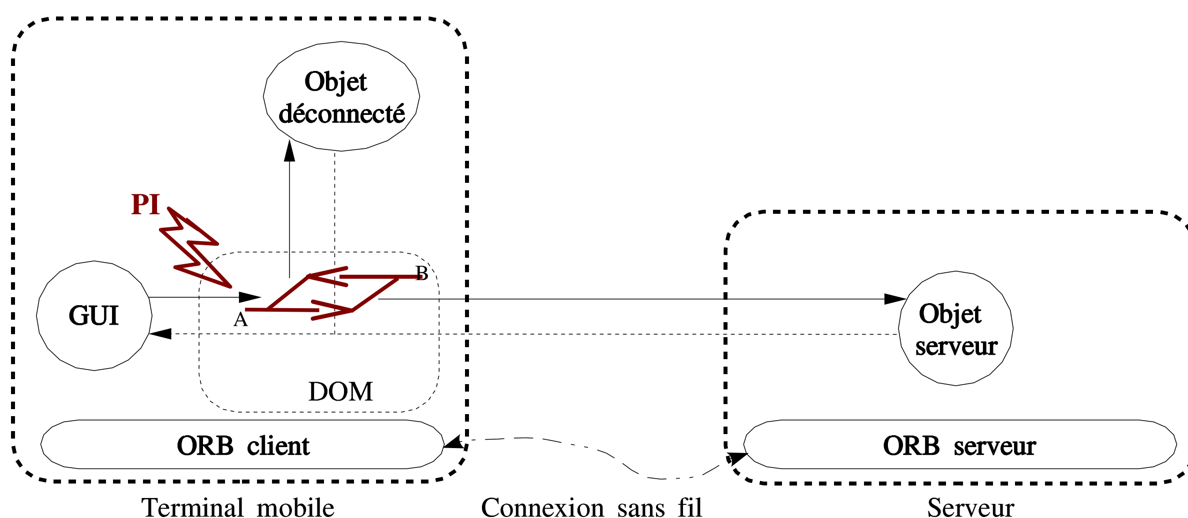

FIG. 6.1 – Le prototype d'implémentation

6.2 Application exemple

Pour illustrer l'utilisation de l'approche de déploiement et de gestion du cache que nous avons développées dans ce rapport, nous avons choisi d'implanter cette approche sur l'application de messagerie électronique vue dans la section 5.1.3. Nous rappelons que cette application est la même que celle développée par l'équipe MARGE de l'INT pour illustrer l'architecture de Domint. Cette application offre des fonctionnalités simplifiées similaires à des logiciels courants tels que Netscape messenger ou Microsoft IE. L'utilisateur manipule des messages consistant en un corps et un en-tête composé de l'identifiant du message (un entier), les noms de l'émetteur et du destinataire, le sujet, la date d'émission et l'état du message (lu ou non lu). Les principales opérations accessibles par l'interface graphique sont l'envoi d'un nouveau message, d'une réponse ou d'un message à suivre, la réception d'un message et l'effacement d'un message.

Pour implanter notre mécanisme de déploiement et celui de gestion du cache, nous avons ajouté quatre objets à l'application de messagerie électronique. Ces objets sont présentés dans la figure 6.2 et sont les mêmes étudiés dans la section 5.1.3.

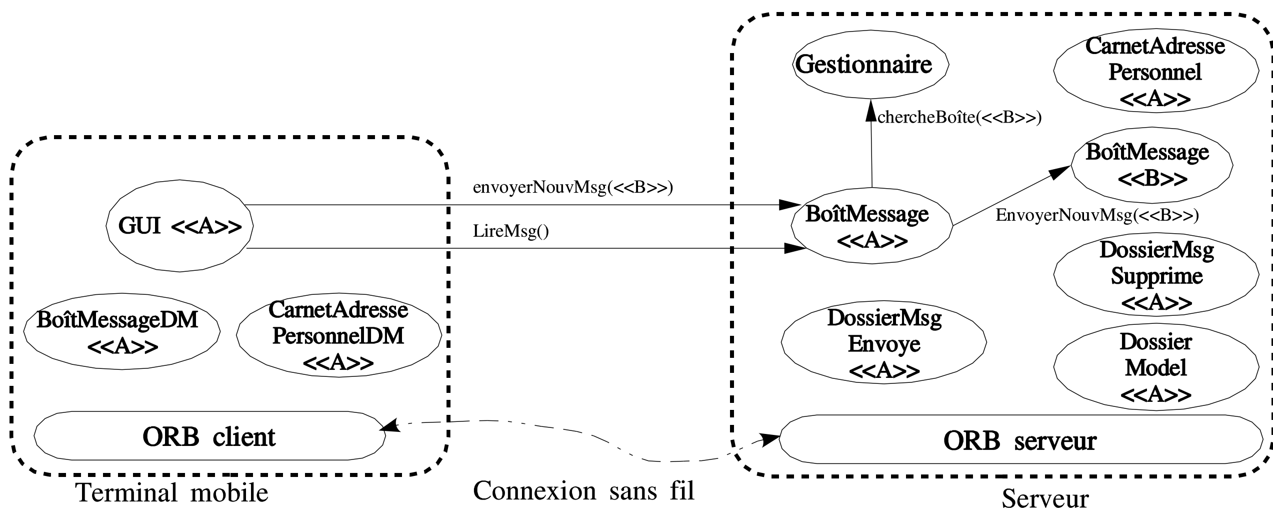

FIG. 6.2 – L'application de messagerie électronique

Tous les interfaces du serveur sont spécifiées dans le module IDL `Mail`, tandis que les objets déconnectés sont spécifiés dans le module IDL `MailDM`. L'objet *Gestionnaire* est créé au lancement de l'application côté serveur. Par contre, les autres objets sont créés à partir de l'objet *Gestionnaire*. En plus, dans notre application, tous les objets sont enregistrés dans le service de nommage.

6.3 Mise en œuvre

Dans l'application de messagerie électronique, les IOR des objets qui permettent le mode déconnecté sont "taguées" par la politique de déconnexion ajoutés par les intercepteurs d'IOR (*Cf.* section 4.1.2). L'intercepteur d'IOR permet d'ajouter un composant "mode déconnecté" à toutes les références des objets d'une application demandant le support des opérations déconnectées. Le nouveau composant contient un booléen `localCopy` indiquant si l'objet référencé doit posséder ou non un objet déconnecté sur le terminal mobile. Ce booléen est égal à faux pour les objets qui n'autorisent pas un objet déconnecté pour le mode déconnecté et à vrai si c'est le cas. L'implantation de cette nouvelle politique se fait dans la classes *DOMPolicyFactory*. Donc, l'intercepteur portable (PI)

ne demande de créer des objets déconnectés que pour les objets dont l'IOR comporte la politique de déconnexion. Pour implanter cette politique, l'application de messagerie électronique comporte le module *pi* donné par la déclaration IDL suivante :

```
 module pi {
    /**
     * Objet politique pour la gestion des déconnexion
     * Cet objet sera associe au IOR de l'objet distant
     */
    const CORBA::PolicyType DOM_POLICY_ID = 1010;

    /**
     * Ajouter le booléen "localCopy" dans CORBA::Policy
     * pour indiquer si l'ORB autorise  les objets
     * déconnectés sur les terminaux mobiles.
     */
    local interface DOMPolicy : CORBA::Policy {
readonly attribute boolean localCopy;
    };
    const IOP::ComponentId DOM_TAG_COMPONENT = 1010;
}
```

Le fonctionnement du l'intercepteur portable est donné dans la classe `CRIDisconnect`. Cette classe implante l'interface `ClientRequestInterceptor` vue dans la section 4.1.1.

Toutes les requêtes et les réponses sont interceptées du côté du terminal mobile. Lors de l'envoi de la première requête à un objet dont la référence contient un composant avec `localCopy` à vrai, l'intercepteur crée un adaptateur puis un objet déconnecté vide. L'objet vide est "rempli" en appelant l'opération `incrementalStateTransfert()` sur l'objet distant. Ensuite, cette mise à jour de l'objet déconnecté est effectuée périodiquement à condition que la connectivité soit bonne. Pour chaque requête du client, l'intercepteur côté client décide quel objet recevra la requête. Si la destination effective doit changer, le point d'interception exécuté lors de l'envoi lève l'exception CORBA `RequestForward`. Cette exception contient l'IOR de la nouvelle destination. L'ORB retransmet automatiquement la même requête sur la nouvelle destination.

Le mécanisme de déploiement que nous avons développé se base sur le partitionnement des objets en objets critiques et objets non critiques. La première idée que nous avons eu pour ce partitionnement est d'ajouter un composant "type d'objet" à tous les IOR des objets. Le nouveau composant contient un booléen `ObjetCritique` indiquant si l'objet est critique ou non. Cette solution n'est opérationnelle que pour les applications pour lesquelles les objets du serveur ne sont manipulables que par un seul client, puisque chaque objet contient une seul politique de criticité. La deuxième solution est de travailler avec un fichier de configuration qui contient le nom logique et la criticité de l'objet. Chaque utilisateur peut ainsi posséder son propre fichier de configuration. Dans cette réalisation, c'est la deuxième solution que nous choisissons. Un exemple de partitionnement des objets de l'application de messagerie électronique est donné dans la table 5.1.

La création des objets déconnectés pour les objets non critiques se fait dans la méthode `deployer()` de l'objet *ResourceManager*. Cette méthode prend en entrée deux chaînes de caractères :

– `fichier` qui représente le chemin d'accès du fichier de configuration.
– `userName` qui représente le nom de l'utilisateur de la boîte au lettres.

L'opération `deployer()` est appelée par l'intercepteur portable dans le constructeur de l'objet *CRIDisconnect*. L'intercepteur portable trouve le point d'entrée de DOM qui permet de trouver les objets *ResourceManager* et *LogManage* en appelant les méthodes `findResourceManager` et `findLogManager` respectivement.

Le traitement à faire dans l'opération `deployer()` peut se résumer comme suit. D'abord, l'opération `deployer()` récupère le fichier de configuration dont le chemin d'accès est donné par le paramètre d'entrée `fichier`. Ensuite, pour chaque entrée de ce fichier, elle contrôle le type de criticité de cet objet via la champ "type" du fichier de configuration. Si l'objet est critique, l'opération `deployer()` récupère la référence de l'objet distant du service de nommage et crée l'objet déconnecté qui sera attaché à l'ORB de DOM. Une fois l'objet déconnecté activé, l'opération `deployer()` crée un gestionnaire de connectivité (CM) pour l'objet déconnecté qui sera attaché au même ORB. Enfin, une entrée est ajoutée dans la table interne de l'intercepteur portable qui comporte la référence de l'objet distant et la référence du gestionnaire de connectivité associé.

Comme décrit dans le chapitre 5, le déploiement des objets non critiques se fait à la première invocation sur cet objet. Au lancement de l'application, l'intercepteur portable récupère le fichier de configuration que l'utilisateur a modifié. À l'interception d'une requête, l'ORB appelle la méthode `send_request()` de l'objet *ClientRequestInterceptor* implanté dans la classe *CRIDisconnect*. Cet appel prend comme paramètre un objet de type *ClientRequestInfo* qui donne toutes les informations sur la requête, en particulier, la référence de l'objet invoqué, l'opération invoquée et les paramètres de cette invocation (*Cf.* section 4.1.1.2). Après avoir récupéré la référence de l'objet invoqué, l'intercepteur portable vérifie le type de criticité de cet objet à partir du fichier de configuration. Dans le cas où l'objet est non critique, l'intercepteur portable parcourt sa table interne pour vérifier si l'objet existe dans le cache. Si la table interne de l'intercepteur portable ne comporte pas d'entrée pour cet objet, l'intercepteur portable appelle la méthode `createConnectivityManager()` de l'objet gestionnaire de ressources RM. Cette méthode prend en paramètre la référence de l'objet invoqué et elle permet de créer l'objet déconnecté et le gestionnaire de connectivité associé. Une fois l'objet déconnecté créé, l'intercepteur portable ajoute une entrée pour cet objet dans sa table interne.

6.4 Travail en cours

À l'instant où nous écrivons ces lignes, nous n'avons implanté que le mécanisme de déploiement que nous avons proposé. Le travail en cours est d'implanter notre approche de gestion du cache. Ensuite, nous envisageons de faire des tests de performance sur le déploiement et la gestion du cache.

Chapitre 7

Conclusion

Le domaine des systèmes répartis est en pleine évolution, en particulier les applications réparties qui relient des terminaux mobiles et des serveurs fixes. Or, la plupart de ces applications sont rarement optimales. En effet, de trop nombreux paramètres influent sur la performance de ces applications. Il est donc souhaitable de fournir une continuité de service malgré les déconnexions et les perturbations du réseau sans fil, notamment sur les terminaux mobiles. Pour fournir la continuité de service, le système doit avoir un mécanisme de déploiement des données dans le terminal mobile pour traiter le problème de déconnexion.

L'étude de la littérature sur la mobilité dans les systèmes répartis a permis de souligner les besoins en déploiement pour les applications fonctionnant en mode déconnecté. Cette étude de la littérature nous a permis de classer les différents mécanismes de déploiement qui existent en quatre approches : pas de cache local au terminal mobile, mise en cache systématique des données, mise en cache des données à la demande et pré-chargement des données.

Nous avons proposé des algorithmes pour, d'une part, développer un mécanisme de déploiement des objets dans les terminaux mobiles pour le fonctionnement en mode déconnecté, d'autre part, gérer la mémoire virtuelle du terminal mobile.

Nous avons étendu la plate-forme Domint pour traiter le problème du déploiement et de la gestion du cache. Bien que la période du stage ne soit pas suffisante pour réalisation entièrement de notre approche, nous avons complété une application de messagerie électronique sur la plate-forme Domint et nous avons implanté les parties concernant le déploiement.

L'approche défendue dans ce rapport repose sur la notion de criticité d'objet. La première étape du déploiement consiste à partitionner les objets de l'application en objets critiques et objets non critiques. Les objets critiques sont les objets nécessaires pour le fonctionnement de l'application en mode déconnecté, tandis que les objets non critiques sont les objets dont l'absences pendant une déconnexion n'empêche pas l'application de continuer à fonctionner. Le déploiement des objets critiques se fait au lancement de l'application. Par contre, le déploiement des objets non critiques se fait à la première invocation par l'application. L'algorithme de gestion du cache proposé se base sur l'algorithme LFU. Dans cet algorithme, le remplacement des objets n'est effectué que sur les objets non critiques.

L'architecture de déploiement et de gestion du cache peut être implantée sur n'importe quel système qui offre le mécanisme d'interception des requêtes. Par ailleurs, le cache du terminal mobile contient au minimum tous les objets nécessaires pour le fonctionnement de l'application, ce qui donne une meilleur disponibilité du service applicatif en mode déconnecté.

Cependant, l'architecture est très liée à la sémantique de l'application, ce qui rend l'application non transparente à l'utilisateur. En outre, la mise en œuvre de notre approche de déploiement et de gestion du cache est liée au mécanisme des intercepteurs non disponibles dans tous les systèmes.

7.1 Perspectives

L'architecture décrite dans ce rapport permet aux programmeurs et aux utilisateurs de configurer facilement leur système de déploiement et de gestion du cache. Cependant, la démarche n'est pas totalement intuitive. Pour pouvoir être utilisée le plus largement, il faudra donc la simplifier au maximum.

Plusieurs extensions du travail présenté dans ce rapport peuvent être envisagées en termes de faciliter d'utilisation, d'enrichissement et d'application à d'autres types de systèmes.

La première extension possible pour le concept de criticité des objets est de définir une criticité dynamique des objets. Cette idée est semblable à celle utilisée dans le système SEER : l'application doit prédire le future besoin de l'utilisateur pour mettre à jour la liste des objets qui doivent exister dans le cache. La deuxième extension possible est l'ajout du rôle de l'administrateur de l'application dans le partitionnement des objets de l'application et de traiter le problème de propagation de la criticité entre les classes de l'application.

La collecte d'informations sur le cache est basée sur l'algorithme LFU. Or, cet algorithme est très limité en termes de performances. Une idée de départ pour améliorer notre approche de gestion du cache est de définir une fonction de coût sur les objets de l'application. Cette fonction peut tenire en comte la criticité de l'objet, le type de réseau à partir duquel se fait la communication avec l'objet distant et la taille de l'objet. La forme de cette fonction reste un sujet de recherche dans les années à venir.

Bibliographie

[AFM98] A.P. Afonso, S.R. Francisco, and J.S Mário. UbiData : An Adaptable Framework for Information Dissemination to Mobile Users. In *Proc ECOOP 98 Workshop on Mobility and Replication*, DI Campo Grande, Lisboa Portugal, 1998.

[Bag98] A. Baggio. Replication and Caching Strategies in Cadmium. Technical Report 3409, INRIA France, Avril 1998.

[CCVB02a] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Disconnected Operations in Mobile Environments. In *Proc. 2nd IPDPS Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, Ft. Lauderdale, USA, April 2002.

[CCVB02b] D. Conan, S. Chabridon, O. Villin, and G. Bernard. A Disconnected Object Management Service for Mobile Environments. Technical report, INT France, Avril 2002.

[CCVB02c] D. Conan, S. Chabridon, O. Villin, and G. Bernard. Weak Connectivity and Disconnected CORBA Objects. *submitted*, Avril 2002.

[GAK00] B. Gerald, V. Andreas, and D. Keith. *Java Programming with CORBA*. Wiley Computer Publishing, 2000.

[GG97] H.K. Geoffrey and J.P. Gerald. Automated Hoarding for Mobile Computers. In *Proc 16th Symposium on Operating Systems Principles (SOSP'97)*, pages 264–275, 1997.

[HL96] B.C. Housel and D.B. Lindquist. WebExpress : A system for optimizing Web Browsing in a Wirless Environment. In *Proc. ACM Symposium on Principles of Distributed Computing (MOBICOM'96)*, NY USA, November1996.

[JHE99] J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2), June 1999.

[JTK95] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek. Rover : A toolkit for mobile information access. *Proc 16th Symposium on Operating Systems Principles (SOSP'95)*, December 1995.

[JTK97] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek. Mobile computing with the Rover toolkit. *ACM Transactions on Computers*, 46(3), 1997.

[KS91] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc 13th ACM Symposium on Operating Systems Principles*, number 5, pages 213–225, Pacific Grove, USA, 1991.

[Kue94] G. Kuenning. Design of the SEER predictive caching schema. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, U.S.A, 1994.

[Lyn99] N. Lynch. Supporting Disconnected Operation in Mobile CORBA. M.sc. thesis, Trinity College Dublin, 1999.

[Mum96] L.B. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, September 1996.

[NSN+97] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proc. 16th ACM Symposium on Operating System Principles (SOSP97)*, Saint-Malo, France, October 5-8, 1997.

[OMG01] OMG. Portable Interceptors. Interceptors Finalization Task Force. Published draft, Object Management Group, September 2001.

[PTT+94] K. Petersen, D.B. Terry, M.M. Theimer, A.J. Demers, J. Spreitzer, and B. Welch. The Bayou Architecture : Support for Data Sharing among Mobile Users. In *Proc of the 1994 Workshop on Mobile Computing Systems and Applications*, 1994.

[PTT+97] K. Petersen, D.B. Terry, M.M. Theimer, A.J. Demers, and J. Spreitzer. Flexible Update Propagation for Weakly Consistent Replication. *Proc 16th Symposium on Operating Systems Principles (SOSP'97)*, 1997.

[RS96] M. Raynal and M. Singhal. Capturing Causality in Distributed Systems. *Computer IEEE*, February 1996.

[RSK00] R. Ruggaber, J. Seitz, and M. Knapp. π^2 - A Generic Proxy Platform for Wireless Access and Mobility. In *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, Portland, Oregon, July 2000.

[Sat96a] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc 15th Symposium on Principles of Distributed Computing (PODC'96)*, pages 1–7, 1996.

[Sat96b] M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1), 1996.

[SG98] A. Silberschatz and P.A. Galvin. *"Principes des systèmes d'exploitation"*. Addison-Wesley, 1998.

[TTP+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing Update Conflicts in Bayou : A Weakly connected Replicated Storage System. *Proc 15th Symposium on Operating Systems Principles (SOSP'95)*, 1995.