Support pour l'exécution, en mode déconnecté, d'applications distribuées dans les environnements mobiles

Denis Conan, Bruno Bretelle, Sophie Chabridon et Guy Bernard
Institut National des Télécommunications
9, rue Charles Fourier
91011 Évry cedex, France
{Denis.Conan|Bruno.Bretelle|Sophie.Chabridon|Guy.Bernard}@int-evry.fr

Résumé

L'exécution d'applications réparties impliquant des terminaux mobiles et des serveurs fixes reliés par des réseaux sans fil nécessite la prise en compte de déconnexions, aussi bien involontaires lors de coupures réseau intempestives, que volontaires lorsque l'utilisateur souhaite diminuer les coûts de communication ou économiser des ressources. Cet article présente comment tirer parti de mécanismes standards de CORBA (objet par valeur et intercepteur portable) afin d'adapter des applications existantes pour qu'elles offrent une continuité de service en cas de déconnexion volontaire ou involontaire. Le premier mécanisme offre une solution simple pour la déconnexion volontaire en transférant sur le terminal une copie de l'objet manipulé, mais il n'est pas adapté à l'autre cas de déconnexion. Le deuxième mécanisme est plus puissant et permet de gérer les deux cas de déconnexion. La commutation du mode connecté au mode déconnecté se fait de manière transparente vis-à-vis de l'utilisateur avec seulement quelques modifications très localisées dans le code de l'application. L'approche proposée est illustrée et validée dans le cas de l'adaptation d'une application de messagerie électronique à un environnement sans fil.

Mots clés : CORBA, assistants personnels numériques, réseaux mobiles, opérations déconnectées.

1 Introduction

La communication sans fil, le traitement d'informations personnelles et les services d'informations réparties auront une importance stratégique dans l'avenir proche. Il existe déjà des terminaux PDA (*Personal Digital Assistant)* commercialement disponibles fournissant ces services. Toutefois, leur capacité informatique générale est encore assez modeste, la capacité de se relier aux services à distance et à l'Internet reste limitée et le nombre d'applications disponibles pour un système particulier est faible par rapport aux possibilités des ordinateurs personnels traditionnels.

Le travail présenté ici est réalisé dans le cadre du projet ITEA VIVIAN [Viv] concernant l'ouverture des plateformes mobiles pour le développement d'applications à base de composants. Les partenaires regroupent des industries européennes de premier plan dans le domaine des PDA et des communications mobiles telles que Nokia et Philips, des instituts de recherche (Université Technologique d'Helsinki, INRIA, INT) et des PME qui apportent au projet la connaissance métier dans des domaines aussi différents que les applications bancaires, les systèmes d'informations géographiques ou les logiciels linguistiques. L'objectif du projet VIVIAN est de fournir une plateforme logicielle adaptée aux terminaux mobiles de nouvelle génération permettant le développement d'applications logicielles par des tiers. Les applications visées sont du type client/serveur où les clients peuvent être mobiles et ont des ressources limitées en termes de taille mémoire et puissance notamment, et le serveur est distant et souvent relié à un réseau filaire par ailleurs. La plateforme VIVIAN repose sur la spécification CORBA de l'OMG et est conçue pour être extensible et évolutive afin de permettre la prise en compte des besoins variés des applications.

L'INT axe ses travaux sur le fonctionnement des applications en mode déconnecté. L'objectif est de fournir un support logiciel conforme aux spécifications CORBA 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.

La suite de l'article est divisé comme suit. La section 2 présente l'application de messagerie électronique prise comme exemple tout au long de l'article. La section 3 développe les deux mécanismes CORBA utilisés pour supporter les déconnexions volontaires et involontaires. L'implantation est réalisée avec ORBacus version 4.0.5 [Orb], dans le langage Java avec le JDK version 1.3.1, sous Linux et Windows. La plateforme matérielle utilisée est des ordinateurs portables connectés par un réseau Internet sans fil avec des cartes "IEEE 802.11b Lan". Les sections 4 et 5 concluent avec une présentation de travaux connexes, une évaluation et des perspectives.

2 L'application exemple

Pour illustrer l'utilisation des mécanismes CORBA pour les opérations en mode déconnecté, nous avons choisi une application de messagerie électronique. Elle offre des fonctionnalités simplifiées similaires à des logiciels courants tels que Netscape ou 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. L'objectif est de montrer comment l'application conçue pour les communications filaires peut être adaptée pour les communications sans fil. Trois versions différentes sont déclinées : centralisée, répartie et sans fil.

Dans la première version dite "centralisée", l'interface graphique pour l'utilisateur est exécutée dans la même entité d'exécution que l'objet qui gère la boîte aux lettres de l'utilisateur. Une deuxième entité d'exécution contient le gestionnaire de boîte aux lettres qui permet à un utilisateur privilégié d'ajouter et d'effacer des boîtes aux lettres. Les deux entités d'exécution sont situées sur le même ordinateur.

La version dite "répartie" est obtenue à partir de la version centralisée en séparant la partie cliente composée de l'interface graphique et la partie serveur constituée des boîtes aux lettres et du gestionnaire des boîtes aux lettres. La partie cliente est déportée sur l'hôte de l'utilisateur et reliée aux boîtes aux lettres et au gestionnaire de boîtes aux lettres par un réseau de communication filaire. Elle mémorise en local les informations parcourues par l'utilisateur. Les opérations de lecture, d'envoi et d'effacement de messages sont aussitôt effectuées sur la boîte aux lettres distante. Par contre, les opérations de modification d'informations déjà existantes localement sur le terminal mobile ne sont effectuées que localement afin d'éviter la transmission de trop nombreuses requêtes sur le réseau. Ce journal des dernières opérations locales depuis la précédente requête est inséré dans la requête qui suit. Ainsi, toute requête sur la boîte aux lettres commence par le traitement d'un journal d'opérations locales. Ce patron de conception est simple à appliquer et fonctionne correctement pour les applications réparties déterministes par morceaux [1] [JZ90]. Une deuxième manière de réduire le nombre de requêtes transmises sur le réseau est d'ajouter des opérations collectives qui lisent ou effacent des groupes de messages : par exemple, tous les messages non lus, tous les messages déjà lus ou tous les messages.

Pour diminuer encore la quantité d'informations transmises sur le réseau, la lecture des messages se fait en deux temps dans la version "sans fil" : d'abord les en-têtes puis les corps des messages. L'utilisateur navigue dans les en-têtes de messages et ne télécharge que les messages qu'il souhaite effectivement lire. Une deuxième raison justifiant ce choix est la faible quantité d'espace mémoire disponible sur le terminal mobile. Les figures 1, 2 et 3 illustrent le fonctionnement en mode connecté, puis en mode déconnecté, et ensuite, la reconnexion. Lorsque le terminal mobile est connecté au serveur où se trouve la boîte aux lettres du client (*Cf.* figure 1), les requêtes sont transmises directement à la boîte aux lettres. L'envoi d'un courrier électronique passe par la boîte aux lettres de l'émetteur qui demande l'adresse (la référence) du destinataire et fait suivre le courrier. Pour les lectures, le client s'adresse à sa boîte aux lettres. Avant la déconnexion (*Cf.* figure 2), une copie locale est créée sur le terminal mobile. Les requêtes sont traitées et journalisées par la copie locale. À la reconnexion (*Cf.* figure 3), les requêtes journalisées pendant la déconnexion sont transmises pour traitement par la boîte aux lettres distante. Dans les sections qui suivent, nous détaillons le support des déconnexions volontaires grâce aux objets par valeur, puis les déconnexions involontaires grâce aux intercepteurs portables.

[1] L'exécution de chaque entité d'exécution est divisée en intervalles. Chaque intervalle est une séquence déterministe d'actions commençant par la réception d'un message (au sens message utilisateur transmis sur le réseau, pas au sens courrier électronique) jusqu'à la réception du message suivant. L'exécution dans un intervalle est complètement déterminée par l'état de l'entité d'exécution au début de l'intervalle et l'état du message reçu au début de l'intervalle. Un nombre quelconque de messages peut être émis pendant l'intervalle. [JZ90]

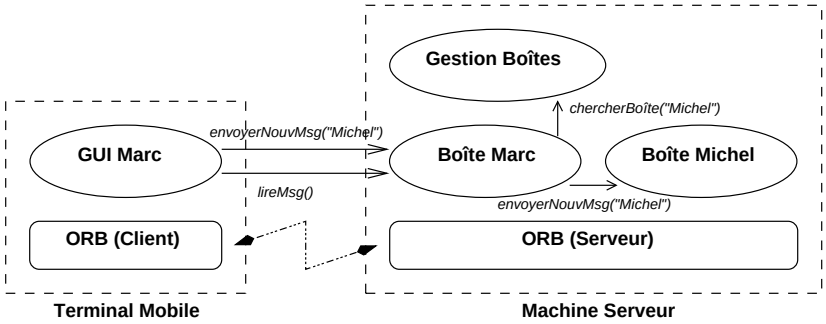

FIG. 1 – Fonctionnement en mode connecté

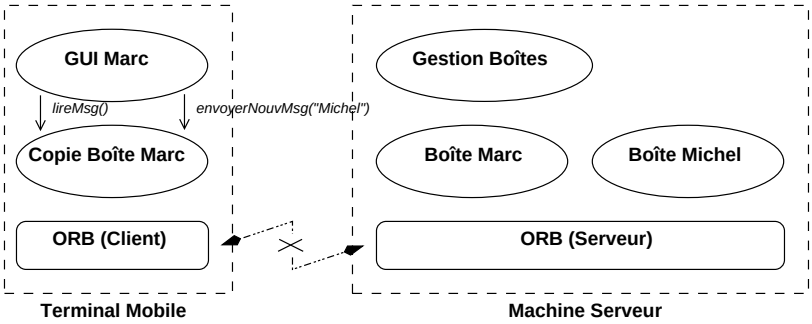

FIG. 2 – Fonctionnement en mode déconnecté

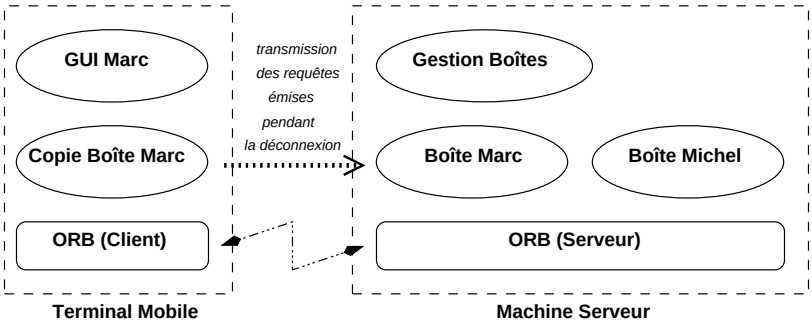

FIG. 3 – Reconnexion avec le serveur

3 Mise en œuvre des mécanismes CORBA

Dans les deux premières versions de l'application de lecture des courriers électroniques, l'interface graphique de l'utilisateur reste presqu'entièrement figée durant les déconnexions. Les seules opérations possibles sont la modification du statut (lu /non lu) des messages déjà téléchargés. Pour continuer son travail, l'utilisateur doit pouvoir accéder à une copie locale de sa boîte aux lettres. En outre, les déconnexions sont, soit volontaires suite à un ordre donné par l'utilisateur qui crée ou met à jour la copie locale, soit involontaires avec mise à jour régulière de la copie locale. Dans cette section, nous développons ces deux alternatives en mettant en œuvre deux mécanismes CORBA : le passage d'objets par valeur et les intercepteurs portables, respectivement. Dans la première solution, la gestion des déconnexions est effectuée par l'interface graphique qui choisit d'accéder à la copie locale ou à la boîte aux lettres distante. Par contre, l'objectif de la seconde solution est la gestion transparente [JHE99] des déconnexions involontaires et volontaires.

3.1 Mise en œuvre avec les objets par valeur

Avant de présenter la solution avec les objets par valeur (*Object By Value* ou OBV) dans la section 3.1.2, la section section 3.1.1 donne une rapide description de ce mécanisme.

3.1.1 Présentation d'OBV

Le concept d'objet par valeur a été introduit dans la version 2.3 de la norme CORBA [OMG01a] pour permettre de passer un objet par valeur et non plus par référence. Ceci implique que le client reçoit une copie complète de l'objet transmis ; cette copie est en fait une nouvelle instance de l'objet locale au client. Le type IDL correspondant à un OBV est `valuetype`. C'est un type mixte entre une interface et une structure. Un `valuetype` est transmis par valeur, comme le type `struct`, et il peut contenir des opérations, comme le type `interface`. La caractéristique essentielle d'un `valuetype` est qu'il est local, c.-à.d. son utilisation génère de simples appels de fonctions, pas des requêtes CORBA. Dans [Lyn99], le principal inconvénient invoqué pour l'utilisation des `valuetype` est les nombreuses modifications nécessaires dans le code du client et du serveur. Un `valuetype` avec état doit spécifier dans l'IDL un attribut pour chaque variable d'état définie dans l'interface qu'il supporte et doit donc également implanter les opérations d'interrogation et de modification correspondantes. Nous expliquons dans la prochaine section comment nous utilisons un `valuetype` personnalisé défini avec le mot-clé `custom` pour éviter de décrire l'état d'un `valuetype` dans l'IDL.

Les objets par valeur s'utilisent souvent, et c'est le cas dans notre solution, avec des interfaces abstraites [OMG01a]. Le concept d'interface abstraite, dont la description IDL commence par le mot-clé `abstract`, permet de choisir au moment de l'exécution si un paramètre doit être passé par objet ou par valeur. Une telle interface ne peut pas être instanciée directement, elle doit être héritée par une interface non abstraite ou "supportée" par un `valuetype` à l'aide du mot-clé `supports`.

3.1.2 Commutation non transparente et transfert d'état

Dans la mise en œuvre d'une solution à base d'objets par valeur pour le fonctionnement en mode déconnecté, nous avons cherché à minimiser l'impact sur l'application existante en réduisant les modifications nécessaires dans l'IDL et dans le code client. L'utilisation d'une interface abstraite permet de commuter très facilement dans le code client entre un objet distant présent sur le serveur et un `valuetype`, correspondant à une copie locale. Ceci offre donc une solution très simple pour la déconnexion volontaire avec un impact sur le code de l'application qui reste limité. Dans l'IDL, l'interface abstraite `AbstractMailBox` est définie avec l'ensemble des opérations pour la manipulation de messages (`send()`, `receive()`, `reply()`…). Cette interface abstraite est ensuite supportée par le `valuetype` `ValueMailBox` et est dérivée pour donner l'interface `MailBoxWirel`. Cette interface offre deux nouvelles opérations par rapport à l'interface d'origine `MailBox` : `disconnect()` est appelée par l'utilisateur lorqu'il souhaite se déconnecter ; `reconnect()` permet de revenir au mode connecté et donc d'appliquer sur l'objet `MailBoxWirel` les modifications effectuées en local pendant la déconnexion. Ci-dessous nous présentons un extrait de la description IDL correspondante. Dans le code de l'application cliente d'origine, la référence sur un objet boîte aux lettres doit être remplacée par une référence sur une interface abstraite. Celle-ci est ensuite renseignée avec soit l'IOR d'un objet de type `MailBoxWirel` dans le mode connecté, soit la référence d'une instance du `valuetype` `ValueMailBox` en mode déconnecté. La seule autre modification nécessaire dans le code client est l'appel aux fonctions `disconnect()` et `reconnect()`.

```
abstract interface AbstractMailBox {
   void sendNewMessage(...);
   Message receiveMessage(...);
   void forward(...);
   void reply(...);
   void deleteMessage(...);
   ...
};

// Forward declaration
interface MailBoxWirel;
```

```
custom valuetype ValueMailBox
             supports AbstractMailBox
{
    factory init(in any mailBoxWirelState);
    void reconnect(in MailBoxWirel mB,
        in LocalOperationLog log);
    };

interface MailBoxWirel : AbstractMailBox {
    ValueMailBox disconnect();
    void reconnect(in LocalOperationLog log);
};
```

Nous décrivons maintenant la manière dont le transfert d'état est réalisé entre un objet de type `MailBoxWirel` et la copie dans un `valuetype ValueMailBox`. Un `valuetype` personnalisé défini avec le mot-clé `custom` dans l'IDL permet de décrire des mécanismes d'encodage et de décodage (opérations `marshal()` et `unmarshal()`) propres au `valuetype` pour la transmission entre le client et le serveur. Ces opérations sont automatiquement appelées par l'ORB du serveur et du client, respectivement. Pour permettre la création d'instances du `valuetype`, une fabrique doit être définie dans son implantation. Cette fabrique doit implanter l'opération `init()` déclarée dans l'IDL ainsi que la méthode `read_value()`. L'opération `disconnect()` du côté serveur appelle l'opération `init()` qui crée le `valuetype` du côté serveur. `init()` prend en entrée un paramètre de type `Any` qui doit contenir l'état de l'objet à recopier côté client. L'utilisation d'un type `Any` permet de transférer les données de manière opaque. Ainsi, aucun attribut d'état n'apparaît dans l'IDL. Lorsque l'opération `disconnect()` retourne le `valuetype`, l'ORB appelle automatiquement l'opération `marshal()`. Du côté client, l'ORB appelle automatiquement les opérations `read_value()` et `unmarshal()` au retour de l'opération `disconnect()`.

3.2 Mise en œuvre avec les intercepteurs portables

Avant de développer la solution dans la section 3.2.2, la section 3.2.1 donne une courte introduction aux intercepteurs portables CORBA.

3.2.1 Présentation des intercepteurs portables

Un point d'interception est un point d'accrochage dans l'ORB par lequel les services de l'ORB peuvent intercepter le flux normal d'exécution de l'ORB [OMG00]. Le support des opérations déconnectées est donc ici vu comme un service, au même titre que le support des transactions par exemple. La portabilité, au sens où le service est utilisable dans tous les ORB, est obtenue par la définition de trois types d'intercepteurs avec, pour chacun d'entre eux, un ensemble défini de points d'interception. Un intercepteur est construit indépendamment de l'ORB qu'il intercepte.

Les premiers points d'interception interviennent lors de la création des références des objets CORBA, nommées en CORBA "IOR". Ces points d'interception définis dans les intercepteurs d'IOR captent en fait les créations d'adaptateurs d'objets (POA) qui créent les références. Ils permettent d'ajouter de nouveaux composants aux IOR pour, dans notre cas, indiquer que tel objet supporte le mode déconnecté, et pour certains de ces objets, qu'il est possible de créer une copie locale sur le terminal mobile. Ces informations contenues dans les références servent aux deux autres types d'intercepteurs pour déterminer le type de traitement à effectuer.

Les deux autres types d'intercepteurs introduisent des points d'interception lors des envois et des réceptions des requêtes et des réponses par les clients et les serveurs. Les points d'interception créés du côté du client sont définis par les intercepteurs dits "côté client" et ceux créés du côté du serveur par les intercepteurs dits "côté serveur". La définition d'un tel intercepteur implique que toutes les requêtes et les réponses passant par cet ORB sont interceptées. Chacun de ces points d'interception analyse l'IOR de l'objet correspondant à la requête ou à la réponse. Si cette IOR possède le composant de la politique du mode déconnecté, un traitement spécifique est appliqué. Ces intercepteurs s'échangent des informations en ajoutant un contexte de service aux requêtes et aux réponses.

Pour supporter les déconnexions volontaires et involontaires, un intercepteur d'IOR est ajouté lors de l'initialisation de l'ORB sur lequel s'exécutent les boîtes aux lettres et le gestionnaire de boîtes aux lettres. En outre, un intercepteur côté client est ajouté à l'ORB de l'application sur le terminal mobile. Ces intercepteurs servent à effectuer la création de la copie locale sur le terminal mobile et à commuter de façon transparente entre l'objet distant et la copie locale.

3.2.2 Commutation transparente entre l'objet distant et la copie locale

Les deux intercepteurs (IOR et côté client) permettent de choisir dynamiquement entre la copie locale et la boîte aux lettres distante selon la qualité de la connexion. Pour simuler la gestion de la qualité de la communication filaire, une entité d'exécution "gestionnaire de connectivité" est ajoutée sur chaque machine. Périodiquement, le gestionnaire de connectivité sur le terminal mobile teste la connexion vers la boîte aux lettres distante par un aller/retour de messages avec une temporisation sur la réception. Un mécanisme d'hystérésis est conçu pour lisser les variations de la bande passante. L'hystérésis définit trois états de la connexion : bon, moyen et mauvais. La déconnexion correspond à l'état "mauvais". Ces trois états correspondent à trois modes de fonctionnement du côté client : respectivement, les requêtes ne sont exécutées que sur la boîte aux lettres distante ; les requêtes sont exécutées sur la copie locale et ensuite transmises à l'objet distant ; les requêtes ne sont exécutées que sur la copie locale.

L'intercepteur d'IOR permet d'ajouter un composant "mode déconnecté" à toutes les références d'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 une copie locale sur le terminal mobile. Ce booléen est égal à faux pour le gestionnaire de boîtes aux lettres et à vrai pour les boîtes aux lettres auxquelles un terminal mobile accède. L'affectation de `localCopy` est faite, soit par le programmeur de l'application à travers l'ajout de la politique "mode déconnecté" aux adaptateurs d'objets, soit automatiquement selon un fichier de configuration spécifiant le nom de l'adaptateur d'objet. Une limite de l'approche est que tous les objets créés avec le même adaptateur d'objets possèdent la même politique avec la même valeur pour `localCopy`. Ce patron d'implantation simple à mettre en œuvre entraîne cependant des modifications mineures du code existant du côté du serveur. Cependant, il est raisonnable d'instancier différents adaptateurs d'objets pour des objets gérés différemment. Une solution pour obtenir une granularité au niveau des objets (plutôt qu'au niveau des adaptateurs d'objets) serait d'intercepter la première requête vers chaque objet et de modifier le composant "mode déconnecté" de la référence via un fichier de configuration.

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 vide. L'objet vide est "rempli" en appelant l'opération `disconnect()` sur la boîte aux lettres distante. Ensuite, cette mise à jour de la copie locale n'est effectuée que lorsque l'état de la connexion passe de "bon" à "moyen". Dans l'état "moyen" et "mauvais", la copie locale exécute toutes les opérations et est donc à jour. Pour chaque requête, 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 restransmet automatiquement la même requête sur la nouvelle destination. Si la destination est la copie locale, cette dernière lit l'état de la connexion et transmet une requête similaire à la boîte aux lettres distante si nécessaire (dans l'état "moyen"). Lors d'une reconnexion (passage de l'état "mauvais" à "moyen"), la copie locale transmet le journal des opérations locales et distantes effectuées pendant la déconnexion. Si ce journal ne contient que des opérations locales, la copie locale appelle l'opération `reconnect()` en donnant en argument le journal des opérations locales. Par conséquent, le code des objets susceptibles de posséder une copie locale sur le terminal est modifié pour contenir les opérations `disconnect()` et `reconnect()`. En outre, le code de l'objet correspondant à la copie locale est à écrire. Pour l'objet boîte aux lettres, il est très similaire au code de l'objet correspondant dans la version "répartie" de l'application.

4 Travaux connexes

Ces dernières années, de nombreux projets se sont intéressés à l'adaptation d'applications existantes à un environnement sans fil. Un panorama est présenté par [JHE99]. Nous ne donnons ici qu'une synthèse des points importants en regard de nos propres travaux. Le projet *Odyssey* [NSN+97] propose des mécanismes propriétaires de bas niveau pour une adaptation dynamique en fonction des variations de bande passante. Cependant, ceux-ci ne prennent pas en compte le mode déconnecté correspondant à une bande passante nulle. *Rover* [JTK97] décrit une architecture pour la gestion des variations de ressources et des déconnexions entre un terminal mobile et des serveurs fixes. Ce projet introduit le concept d'objet relogeable (*Relocatable Dynamic Object* ou RDO) pour le transfert d'objets entre un serveur et le terminal. Ceci implique que l'application soit implantée entièrement en termes de RDO, ce qui ne facilite pas l'adaptation de logiciels légataires.

Dans le contexte CORBA, le principal problème étudié est celui de la mobilité de l'utilisateur. La spécification *Wireless* CORBA [OMG01b] définit une extension de la couche transport pour rendre la mobilité du terminal client complètement transparente à l'application serveur. Seules les déconnexions involontaires de très courtes durées sont prises en compte, comme lors du passage d'une cellule du réseau sans fil à une autre. L'utilisation d'un ORB respectant

cette norme permettra à la plateforme VIVIAN de bénéficier de cette facilité. Le projet Π^2 [RSK00] utilise la notion de proxy pour rendre les déconnexions intempestives transparentes à l'utilisateur. Un proxy doit être installé sur le terminal et un autre sur le réseau filaire relié au serveur. Notre approche ne nécessite pas l'utilisation de proxy et permet en plus la gestion des déconnexions volontaires. Toujours dans le contexte CORBA, le projet *ALICE* [Lyn99] est également basé sur des proxies et prend en compte à la fois les déconnexions volontaires et involontaires en utilisant les OBV mais au prix d'une importante modification des applications existantes. Dans notre approche, l'utilisation des intercepteurs portables permet de limiter l'impact sur le code de l'application.

Enfin, dans le domaine des composants, *SIRAC* [CB01] propose un mécanisme pour la duplication des *beans* d'un serveur basé sur les Enterprise Java Beans vers un terminal. Indépendamment de la technologie utilisée, EJB *vs.* CORBA, cette approche est très proche de la nôtre.

5 Évaluations et perspectives

Le mécanisme CORBA des objets par valeur permet de créer une nouvelle instance d'objet avec le même état. Il est utilisé pour transférer sur le terminal mobile toutes les données nécessaires au fonctionnement en mode déconnecté. Les modifications dans l'IDL de l'application sont limitées et son utilisation est simple dans le code de l'application cliente exécutée sur le terminal mobile. Les inconvénients sont la non-transparence pour le programmeur de l'application et l'impossibilité de gérer la commutation automatique sans intervention de l'utilisateur, entre les modes connecté et déconnecté.

Le mécanisme CORBA des intercepteurs portables rend possible l'insertion de traitements dans le flot normal d'exécution de l'ORB, à la création des adaptateurs d'objets du côté serveur et à l'émission et à la réception des requêtes et des réponses du côté client et du côté serveur. Il est utilisé pour commuter de façon transparente entre les modes connecté et déconnecté en fonction des résultats de l'observation des communications sans fil. Les déconnexions volontaires et involontaires sont supportées de la même façon. Les modifications dans l'IDL de l'application sont encore plus limitées que pour les objets par valeur. En outre, aucune modification n'est nécessaire dans le code de l'application cliente exécutée sur le terminal mobile. Les inconvénients encore présents dans le prototype illustré dans cet article sont la nécessité de modifications mineures du côté du serveur (l'objet à déporter sur le terminal mobile doit posséder les opérations `disconnect()` et `reconnect()`) et la non-généricité des intercepteurs construits.

En conclusion, nous avons montré dans cet article que les mécanismes CORBA peuvent être utilisés afin d'adapter les applications au fonctionnement en mode déconnecté. Nous travaillons actuellement à minimiser les modifications à effectuer dans le code des applications légataires, plus précisément, sur la non-modification de la partie serveur, et sur la généricité des intercepteurs portables conçus pour les déconnexions. Une autre tâche en cours est le portage de la partie cliente sur un PDA.

Références

[CB01] S. Chassande-Barrioz. Adaptation à la mobilité par la duplication de Bean dans un serveur EJB. 2001 samoa workshop, `http ://sirac.imag.fr/SAMOA/samoa-workshop01/wkshp2001.html`, March 21-23, 2001.

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

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

[JZ90] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

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

[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. Sixteenth ACM Symposium on Operating System Principles (SOSP97)*, Saint-Malo, France, October 5-8, 1997.

[OMG00] OMG. Portable Interceptors. Interceptors Finalization Task Force. Published draft, Object Management Group, April 2000.

[OMG01a] OMG. The Common Object Request Broker - Architecture and Specifications. Revision 2.4.2. OMG Document formal/01-02-01, Object Management Group, February 2001.

[OMG01b] OMG. Wireless Access and Terminal Mobility in CORBA Specification. OMG Document dtc/01-06-02, Object Management Group, June 2001.

[Orb] ORBacus for C++ and Java - Vesion 4.0.5. `http ://www.ooc.com`.

[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.

[Viv] The ITEA Vivian project web site. `http ://www-nrc.nokia.com/Vivian`.