RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Adaptation d'une application répartie pour la disponibilité : Expérience et évaluation

Vania Marangozova, Daniel Hagimont

Projet SIRAC (INRIA-INPG-UJF),
INRIA Rhône Alpes, 655 av. de l'Europe, 38330 Montbonnot St Martin, France
Vania.Marangozova@inria.fr, Daniel.Hagimont@inria.fr

Résumé
Dans cet article nous présentons notre approche d'adaptation d'applications réparties existantes pour une exécution sous différentes contraintes réseau. Les applications, structurées en termes de composants, sont adaptées sans modification du code applicatif. Cette solution favorise la réutilisation de code et diminue les coûts de production de logiciel. Nous décrivons notre expérience d'adaptation d'une application existante dans le cadre précis de la garantie de disponibilité des applications en utilisant la duplication.

Mots-clés : adaptation, disponibilité, duplication

1. Introduction

Le besoin d'adaptation dans le cas des systèmes et des applications répartis se fait ressentir de plus en plus avec le progrès et la diversification vertigineux des réseaux, protocoles et dispositifs de calcul. Nombreux sont les cas où un même service doit être fourni dans différents environnements d'exécution : au sein d'un réseau local, d'un réseaux global, pour des entités mobiles. C'est particulièrement vrai avec le développement du marché des dispositifs mobiles comme les assistants personnels (PDA) ou les téléphones portables. Les concepteurs de services et applications répartis doivent faire face à un nombre sans cesse croissant de plates-formes et de contextes d'utilisation. Pour répondre aux besoins de réalisation de nouvelles versions de leurs services tout en suivant la vitesse de renouvellement du marché, la réponse est l'adaptation.

Dans cet article nous considérons l'adaptation d'applications réparties déjà existantes à des contextes réseaux différents (pannes et connectivité intermittente). En nous plaçant dans un contexte à composants, nous visons une adaptation sans modification du code fonctionnel de l'application. Cette approche évite le redéveloppement coûteux des applications réparties pour les besoins d'évolution. Pour répondre à cet objectif nous utilisons une plate-forme intergicielle nommée JavaPod[1], implémentée au sein de l'équipe SIRAC. Cette plate-forme fournit des mécanismes pour la séparation, la création et la composition de différents aspects du logiciel [9]. Les adaptations que nous avons réalisées assurent la disponibilité des applications. Elles intègrent et adaptent la gestion de la duplication et de la mise en cohérence suivant le contexte réseau considéré.

La suite de cet article est organisée comme suit. La deuxième section est consacrée à la problématique de l'adaptation et à sa déclinaison dans le cas de la disponibilité. La section 3 présente les scénarios que nous avons considérés et leurs besoins d'adaptations. La section 4 décrit la mise en œuvre de ces scénarios ainsi que la plate-forme JavaPod utilisée pour notre expérience. Une conclusion est proposée dans la section 5.

2. Adaptation pour la disponibilité

Nous présentons tout d'abord le problème de l'adaptation en général pour nous concentrer dans la suite sur le problème de la disponibilité.

2.1. Adaptation d'applications réparties

Nous considérons le problème d'adaptation dans le cadre d'applications à base de composants.

Une application répartie est un ensemble de composants distribués interconnectés entre eux. Les composants fournissent des services déclarés à l'aide d'interfaces. Malgré le fait que la plupart des modèles à composants utilisés sont des modèles à plat (CCM[11], EJB[14], COM[3]), nous choisissons de ne pas nous restreindre à un seul niveau de hiérarchie. Nous admettons par conséquent la possibilité d'existence de composants composites.

Nous définissons l'adaptation d'une application à base de composants comme un processus qui :

- ne modifie pas le code implémentant les fonctions fournies par le composant (*propriétés fonctionnelles*).
- effectue par conséquent des modifications sous forme d'extensions et de reconfigurations.

Pour réaliser ce type d'adaptation, il est souhaitable de pouvoir séparer les propriétés fonctionnelles du reste du code (appelé par opposition *propriétés non fonctionnelles*). Dans les applications réparties, les propriétés non fonctionnelles représentent la majorité du code. Elles concernent la gestion de la communication entre composants, l'hétérogénéité des plates-formes, la protection, la persistance, etc.

La séparation des propriétés fonctionnelles des propriétés non fonctionnelles dans le code[9] n'existe pas au niveau des réalisations "classiques" d'applications et de services répartis. Dans CORBA[12], par exemple, malgré l'effort de modularisation et de définition d'objets services (correspondant à la gestion de propriétés non fonctionnelles), ces objets sont utilisés explicitement par le programmeur dans le code. Un premier effort dans ce domaine a été fait par la spécification EJB qui considère les propriétés de transaction et de persistance. La gestion fournie est implicite et facilite beaucoup la construction d'applications EJB. Cependant, le programmeur est contraint de n'utiliser que ce qui est proposé par la plate-forme. Il n'a pas le choix entre plusieurs types de gestion pour une propriété donnée, ni la possibilité de choisir l'ensemble des propriétés non fonctionnelles considérées.

Nombreux sont les projets de recherche s'intéressant à la séparation et à la possibilité de configuration des aspects. Parmi ceux-ci nous trouvons AspectJ[10] qui reste au niveau langage et ne fait pas de distinction entre aspects fonctionnels et non fonctionnels. FlexiNet[6] s'intéresse à une architecture avec possibilité de configuration de la pile de protocoles de communication. Des travaux existent autour des adaptations pour le multimédia[4]. Cependant, rares sont les projets s'étant intéressés à des adaptations impliquant l'utilisation et la configuration des aspects de duplication et de mise en cohérence.

2.2. Application dans le cadre de la disponibilité

Dans le cadre de la problématique d'adaptation, nous choisissons de considérer les adaptations nécessaires pour assurer la disponibilité des applications sous différentes contraintes réseau. La solution classique pour la disponibilité implique l'utilisation des aspects de duplication et de mise en cohérence. Nous considérerons ici deux cas où la disponibilité est mise en cause : le premier concerne le problème classique de tolérance aux pannes alors que le deuxième touche à la gestion des usagers mobiles. Le rôle de la duplication (et la mise en cohérence) est présenté dans les deux cas.

2.2.1. Disponibilité et tolérance aux pannes

La tolérance aux pannes d'un service réparti garantit la capacité du service de fournir un fonctionnement cohérent malgré les défaillances de machines ou du réseau. La disponibilité dans ce cas est classiquement assurée par la duplication des composants du service sur plusieurs machines. Un problème inséparable de la duplication est la cohérence. Il est clair que si une défaillance rend indisponible un composant du service considéré, la copie qui doit se substituer à ce composant doit fournir le même comportement. Les copies du composant doivent donc être maintenues cohérentes entre elles. Une solution classique est la garantie de la cohérence forte[15] par exécution transactionnelle sur les serveurs dupliqués.

2.2.2. Disponibilité et mobilité

L'activité des usagers mobiles se déroule dans un environnement à connections intermittentes. Du fait de la non fiabilité ou du coût des communications, les entités mobiles sont obligées de faire face au phénomène de déconnexion. La disponibilité d'un service, même en mode dégradé, est de nouveau assurée

par la duplication. Selon qu'il s'agit de déconnexion totale (aucune connexion réseau) ou locale, le service sera dupliqué sur l'entité mobile ou éventuellement sur une machine de proximité. Les processus de duplication et de mise en cohérence sont différents du cas précédent. A la place d'une exécution atomique il est éventuellement plus intéressant d'avoir une communication différée comme dans Rover[8]. En ce qui concerne la cohérence, Bayou[2] montre qu'il peut être intéressant d'utiliser une cohérence faible avec un protocole de réconciliation de conflits spécifique à l'application.

2.2.3. Solutions pour la duplication et la mise en cohérence

Les deux cas extrêmes de solutions de duplication et de mise en cohérence que nous avons brièvement présentés, ne sont qu'une petite partie de la panoplie de solutions possibles. Les schémas de duplication diffèrent par les rôles des copies impliquées (maître esclave ou importance égale), la façon dont ces copies sont créées (création initiale par administrateur, création à la volée lors des communications entre entités, etc.), la façon dont les interactions entre copies sont gérées (pouvant inclure des transactions, la gestion de groupe, gestionnaire de localisation, etc.). Les modèles de cohérence sont également nombreux. Il y a les approches de cohérence basées sur l'image mémoire d'une entité ou sur la sémantique applicative de son état.

A notre connaissance, parmi les systèmes existants et prenant en compte les aspects de duplication et de mise en cohérence, il n'existe pas de solutions orientées composant et proposant des mécanismes d'adaptation ou de choix entre les différentes politiques de gestion. Les solutions standards consistent à définir des protocoles qui sont implicitement utilisés par le système et ne sont pas modifiables. C'est le cas des approches classiques dans le domaine des bases de données ou des systèmes de fichiers. Par opposition à une approche de duplication de composants sémantiquement riches, ces protocoles ne concernent que la duplication de données (en termes de pages, blocks mémoire). Avec le succès des dispositifs mobiles, les schémas classiques de duplication et de mise en cohérence ont été remis en question. Des projets comme Rover[8], Bayou[2], etc. ont proposé des solutions incluant des schémas de duplication point à point (*peer-to-peer*), la cohérence relâchée et des mécanismes de réconciliation spécifiques à l'application. Cependant, ces travaux sont motivés par la recherche de solutions dans le contexte spécifique de la mobilité et beaucoup de leurs choix sont figés.

Nous proposons donc d'étudier l'intégration et l'adaptation de la gestion de la duplication et de la mise en cohérence pour la garantie de la disponibilité. Nous présentons dans le section suivante les scénarios d'adaptation considérés.

3. Scénarios d'adaptation d'une application

Dans cette section nous illustrons les idées présentées précédemment sur l'adaptation pour de la disponibilité. Nous considérons un scénario d'application et deux adaptations possibles. Les deux adaptations correspondent aux cas discutés dans la section 2 et notamment la tolérance aux pannes et les déconnexions des mobiles.

3.1. La réservation de matériel: version de base

L'application que nous considérons est une application de réservation du matériel disponible dans l'établissement de l'INRIA Rhône-Alpes. Les personnes travaillant à l'institut peuvent se connecter par l'intranet à cette application, consulter les disponibilités d'un outil et faire la réservation pour une période donnée. L'application est architecturée suivant un modèle standard client-serveur. Le client ne dispose que d'une interface graphique qui est une appliquette Java. Toute action de l'utilisateur sur cette appliquette est transformée en requête vers le serveur. Le serveur unique gère toute création, suppression ou modification de réservation, ainsi que la base de données (base relationnelle) correspondante. Les requêtes des différents clients sont centralisés et mises en séquence sur le serveur ce qui évite les réservations conflictuelles.

3.2. Adaptation pour la tolérance aux pannes

Les pannes que nous considérons dans ce scénario sont les pannes de serveur. Les aspects qui doivent être rajoutés à l'application sont les suivants :

Création de copie : Il est important de décider quelles données et quels composants du serveur doivent être dupliqués. Si l'objectif était la répartition de charge il aurait été souhaitable par exemple de

partitionner les données entre les différents serveurs en fournissant les mêmes traitements. Vu que nous considérons le cas des pannes de serveurs et que le but est de pouvoir rediriger un client d'un serveur défaillant vers un autre serveur de manière transparente, la duplication concernera la totalité des données et des traitements.

Gestion du groupe de copies : La gestion de copies implique leur placement sur des machines ainsi que des mécanismes de mise à jour de la vue de groupe (détection de panne et effacement, création de serveur et rajout). Le groupe peut être défini statiquement ou évoluer dynamiquement[2].

Gestion des connexions client-serveur : Pour éviter qu'un client soit connecté à un serveur défaillant il faut, d'une part, établir la connexion initiale vers un serveur opérationnel et, d'autre part rediriger "à la volée" cette connexion lors d'une panne.

Gestion de la cohérence : Pour permettre la transparence des pannes pour les clients, il faut gérer la cohérence entre les copies de serveur. Ceci implique que toute requête (de modification d'état) d'un client doit être exécutée sur toutes les copies de manière atomique.

3.3. Adaptation pour le support de la déconnexion des mobiles

Nous considérons ici le cas de création de copie en local pour les dispositifs mobiles. Dans le cas de l'application, il s'agit toujours de dupliquer le serveur. Regardons quels sont les traitements à rajouter dans ce cas :

Création de copie : Comme dans le cas précédent il est nécessaire de fournir un traitement de duplication. Cependant dans ce cas il faut prendre en compte les capacités limitées du dispositif. La limitation en mémoire interdit par exemple la duplication de toutes les données de la base gérée par le serveur. Le dispositif peut également imposer l'utilisation d'un autre service de persistance que la base de données relationnelle. Ceci influence la gestion des propriétés non fonctionnelles. Selon le type de déconnexion (anticipée ou pas), la duplication peut être paresseuse (déclenchement explicite avant déconnexion) ou active (anticipation de déconnexion).

Gestion du groupe de copies : Dans le cas de plusieurs serveurs et pour permettre au mobile de se connecter toujours au serveur le plus proche pendant ses déplacements, une gestion de groupe est nécessaire.

Gestion des connexions client-serveur : Un mécanisme de redirection des connexions doit prendre en charge les mises à jour de références entre client et serveur. En mode déconnecté le serveur est local, alors qu'en mode connecté il est à distance. Si un groupe de serveurs existe, la reconnexion peut éventuellement faire usage d'informations de localisations pour choisir le serveur le plus proche.

Gestion de la cohérence : Permettre à l'utilisateur mobile l'utilisation de l'application et la création de réservations pendant la déconnexion implique que la cohérence gérée ne peut être que faible. Il faut prévoir un traitement de détection et de réconciliation de conflits. Appliquer une solution classique où un conflit est défini par deux écritures sur le même objet et où la réconciliation consiste à affecter l'état de l'un à l'autre est peut être une approche trop restrictive. Elle risque surtout d'être décevante pour l'usager mobile dont le travail serait perdu. Pour éviter cette situation, il est possible de fournir un traitement plus sophistiqué de détection et de réconciliation des conflits spécifique à la sémantique de l'application. Un exemple simple est d'avertir l'utilisateur du fait qu'une opération peut être conflictuelle et de lui proposer de fournir des alternatives. En cas de connexion intermittente, le processus de réconciliation peut être actif et utiliser toute possibilité de reconnexion pour envoyer des modifications. Il peut également être passif et ne faire la réconciliation qu'en une fois (cas de déconnexion volontaire).

Gestion de journal (*log*) : La journalisation est nécessaire pour tenir compte des opérations en mode déconnecté et être en mesure de les rejouer lors de la reconciliation.

4. Éxpérience et évaluation

Dans cette section nous décrivons notre expérience d'implémentation de l'application de réservation en utilisant la plate-forme JavaPod. Le langage utilisé est Java[13]. Nous commençons par une brève présentation de la plate-forme JavaPod suivie d'une description de l'implémentation de la version de base de l'application. Nous rappelons ensuite nos objectifs de réalisation des adaptations dans le cadre de cette application. Nous montrons comment ces objectifs sont atteints dans les deux scénarios d'adaptation et

détaillons les implémentations basées sur le mécanisme d'extension des JavaPod. Les conclusions sur les avantages d'utilisation d'une telle plate-forme pour la réalisation d'adaptations ainsi que la faisabilité de ces adaptations dans le cadre précis de l'aspect de duplication sont présentées à la fin de cette section.

4.1. La plate-forme JavaPod

La plate-forme JavaPod est un intergiciel pour les applications réparties à base de composants. Dans le modèle à composants choisi il n'est pas prévu de composants composites. L'objectif de JavaPod est de définir des mécanismes permettant d'associer aux composants, de façon transparente, les propriétés non fonctionnelles requises par chaque application. Ces propriétés doivent pouvoir être choisies parmi une liste de propriétés ouverte et extensible. Comme son nom l'indique, JavaPod utilise le langage de programmation Java.

4.1.1. Éléments d'architecture

L'architecture de la plate-forme repose sur les trois concepts principaux de serveur, conteneur et objet de liaison.

Le serveur, notion inspirée par l'architecture des EJB, est un support d'exécution pour les conteneurs, qui sont eux même une structure d'accueil pour composants. Un serveur fournit tous les services systèmes dont peuvent avoir besoin les conteneurs : protocoles de communication, service de persistance, gestion de ressources, etc.

Le conteneur, notion également inspirée par les EJB, est la partie système correspondant à un composant. Le conteneur d'un composant "encapsule" celui-ci au sens où toutes les interactions du composant avec l'extérieur doivent passer par le conteneur. Grâce à cette interposition, le conteneur peut gérer les propriétés non fonctionnelles du composant : modèle de persistance, de synchronisation, de duplication, etc.

L'objet de liaison (connecteur), inspiré par le modèle ODP[7], est un objet spécialisé dans les communications entre les composants. Il a la particularité d'être constitué de plusieurs morceaux pouvant être répartis sur des sites différents. L'objet de liaison n'a pas d'existence réelle dans la plate-forme. Il est représenté par des talons et des squelettes pouvant implémenter les deux bouts de différents types de communication (RPC, messages asynchrones, diffusion, etc.)

4.1.2. Adaptabilité de la plate-forme

L'originalité de la plate-forme n'est pas dans le choix des entités intervenant. Elle réside dans la qualité d'adaptation et d'extension de ces entités. En effet, tous les quatre types de composants de la plate-forme (serveurs, conteneurs, talons et squelettes) JavaPod peuvent être étendus à l'aide d'entités, appelés extensions. Sachant que les composants de base fournis par JavaPod sont génériques et ne contiennent aucun traitement en ce qui concerne les services système, c'est en définissant l'assemblage d'extensions que sont définies, de façon modulaire, les propriétés non fonctionnelles de la plate-forme.

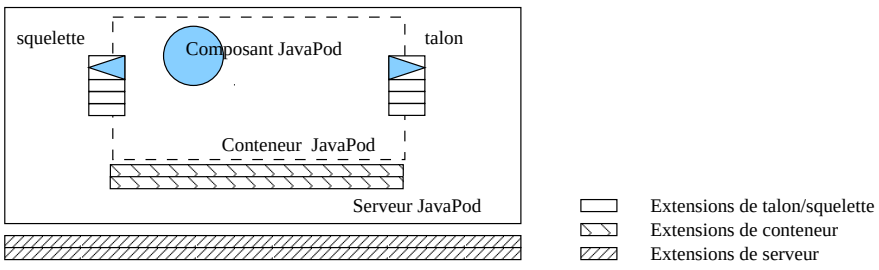

FIG. 1 – *Architecture JavaPod*

La composition des extensions pour une entité donnée est linéaire c'est à dire toute extension surcharge et/ou augmente les traitements fournis par l'extension la précédant. La composition peut être définie de manière statique (configuration initiale) avant le lancement du système et peut être modifieé dynamiquement. Plus de précisions sur le mécanisme et la réalisation de la composition peuvent être trouvées dans [1].

4.2. Version de base

Pour utiliser l'application dans le cadre de la plate-forme JavaPod, nous l'avons réarchitecturée en utilisant le modèle à composants proposé. Pour transformer le client et le serveur en composants JavaPod, nous utilisons une classe prédéfinie par la plate-forme : la classe `Component`. En étendant cette classe, le client et le serveur n'ont besoin que de spécifier leur code fonctionnel. Dans le cas du client, il s'agit de traiter les actions venant de l'interface graphique et de transmettre les requêtes au serveur. Dans le cas du serveur il s'agit de l'implémentation de son interface fonctionnelle que nous avons appeleé `ServerAgenda_itf`.

```
import javapod.*;

public class ClientAgenda extends Component
 implements ClientAgenda_itf, Runnable {
 ServerAgenda_itf server;

 public ClientAgenda(ServerAgenda_itf s) {...}


 public boolean createEvent(Reservation
                            reservation) {
   boolean result = true;
   try {
     result = server.createEvent(reservation);
   } catch (Exception e) {
     e.printStackTrace();}
   return result;}
... }
```

```
import javapod.*;

public class ServerAgenda extends Component
implements ServerAgenda_itf {
private ResourceManager resourceManager;;
privete ReservationManager reservationManager;



 public boolean createEvent(Reservation
                            reservation) {
   boolean result = true;
   try {
reservationManager.addReservation(reservation);
   } catch (OccupiedPeriodException e) {
      e.printStackTrace(); result = false;}
     return result;}
 ... }
```

FIG. 2 – *Schémas du code fonctionnel du client et du serveur dans la plate-forme JavaPod*

Comme on le voit dans l'extrait des codes fonctionnels des composants, le protocole de communication entre les deux n'est pas explicite. La communication est rajoutée ultérieurement à l'aide des extensions JavaPod qui viennent se placer au niveau des talons et des squelettes des composants. La communication que nous avons mis en place pour cette version de base est une communication à distance "à la RMI". Les extensions `Skeletonx` et `Stubx` sont rajoutées respectivement au niveau du squelette du serveur lors de l'exportation de sa référence, et au niveau du talon du client lors de l'importation de la référence du serveur. L'exportation et l'importation de références se font respectivement par les méthodes `bind` et `lookup`. La figure qui suit montre schématiquement la structure de ces méthodes. L'éventuel ajout d'une extension supplémentaire est montré dans les deux lignes en commentaire.

```
public final void bind (Object o, String itf,
     String name) throws Exception {

  Container cont = ...
  ExtensionSet set = new ExtensionSet();
  set.add(new Skeletonx());
  //set.add(new SkeletonExtension2()):
  Reference ref=cont.exportReference(o,itf,set);
  ...}
```

```
public final void Object(String itf,
     String name) throws Exception {
  Reference ref = ...
  Container cont = ...
  ExtensionSet set = new ExtensionSet();
  set.add(new Stubx());
  //set.add(new StubExtension2());
  return cont.importReference(ref,itf,set);
  ...}
```

FIG. 3 – *Configuration du protocole de communication à l'aide des extensions Skeletonx et Stubx*

En ce qui concerne le serveur, qui logiquement est un composant composite, nous avons préféré le représenter par un seul composant JavaPod. Ceci nous permet de le définir comme l'unité da manipulation (réutilisation, duplication, etc.) et nous évite la manipulation explicite de graphe de composants interconnectés. L'architecture de l'application est montrée de manière schématique à la figure 4.

4.3. Approche pour la réalisation des adaptations

Rappelons nos objectifs en ce qui concerne la nature des adaptations. Nous voulons une réutilisation maximale des composants et recherchons des adaptations qui peuvent être faites plutôt en rajoutant des traitements additionnels autour des composants qu'en modifiant le code de ceux-ci.

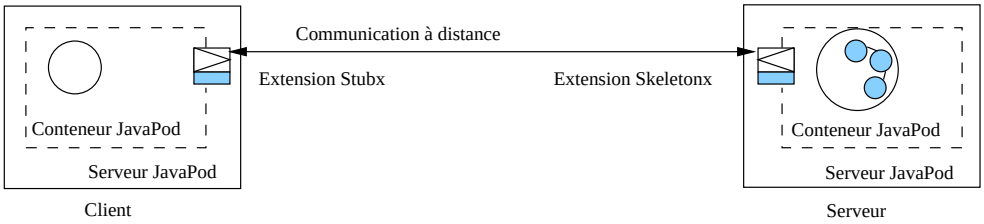

FIG. 4 – *Architecture de la version de base dans la plate-forme JavaPod*

Dans le cadre précis de la duplication auquel on se place, nous prenons l'approche suivante.

Tous les aspects purement non fonctionnels inclus dans la gestion de la duplication et de la cohérence sont gérés sous formes d'extensions rajoutés au niveau des talons/squelettes et si nécessaire au niveau des conteneurs. Il s'agit par exemple du type de communication ou la gestion du groupe de copies.

En ce qui concerne les traitements de gestion de l'état des composants lors de la duplication ou la mise en cohérence, nous avons besoin de modifier leur code. En effet, pour manipuler l'état il y a deux possibilités. La première est une manipulation de l'extérieur impliquant une transgression des règles d'encapsulation. La deuxième est la modification du code du composant pour inclure des traitements additionnels pour la gestion d'état. Le fait de rajouter ce code dans le composant permet de fournir des traitements de duplication et de mise en cohérence qui sont spécifiques à l'application et de ce fait plus efficaces. Dans le cas de composants composites, ces traitements peuvent nécessiter la création de nouveaux sous-composants (changement de l'architecture interne du composite) voire la modification de certains sous-composants. Nous choisissons donc cette deuxième approche qui est utilisée à travers un mécanisme d'appel ascendant (*upcall*).

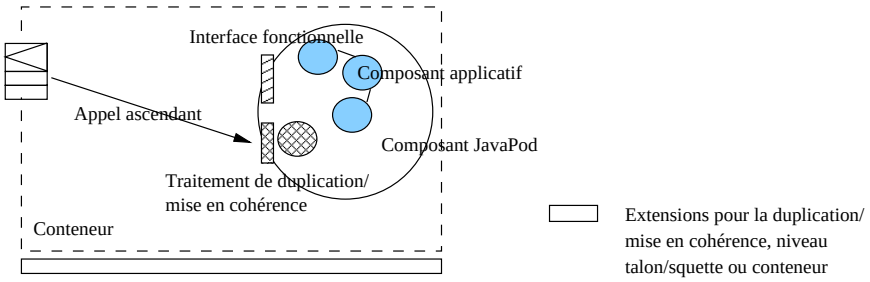

FIG. 5 – *Modèle de composant pour la duplication/mise en cohérence*

4.4. Adaptation pour la tolérance aux pannes

Les pannes que nous considérons dans ce scénario sont les pannes de serveur. Nous supposons que ce sont des pannes de type arrêt sur défaillance (*fail-stop*).

Pour aider à fournir les traitements de duplication et de mise en cohérence, nous avons modifié le code pour ajouter des méthodes permettant la manipulation de l'état (utilisation du patron des méthodes "get" et "set").

Les méthodes "get" et "set" sont utilisées par une extension `ServerStateMgtExtension`. Son rôle est d'implémenter la procédure de capture et de restauration d'état dans ce scénario. Elle fournit respectivement les méthodes `Capture` et `Restore`. La méthode Restore est utilisée pour la mise en cohérence et en effet substitue l'état courant d'un serveur avec l'état capturé d'un serveur fonctionnel. Elle est appelée au moment de la création dynamique ou lors du retour au fonctionnement normal d'un serveur. L'état capturé est passé sur le réseau en utilisant un format spécifique.

Pour garantir une synchronisation sur le serveur (capturé ou restauré) correcte, à l'extension `ServerStateMgtExtension` vient se rajouter une extension `ServerLockExtension`. Son rôle est de verrouiller et de ce fait d'éviter toute mise à jour de l'état d'un serveur pendant le processus de capture ou de restauration.

La gestion du groupe de serveurs est fournie par la `GroupMgtExtension`. Au niveau du conteneur des composants elle contient la liste des serveurs opérationnels. Cette liste est mise à jour par les actions de `ClientGroupMgtExtension` (coté client) ou de `ServerGroupMgtExtension` (coté serveur). `ClientGroupMgtExtension` est responsable de la redirection transparente (pour un client) d'un serveur défaillant vers un serveur opérationnel. Un éventuel échec de connexion à un serveur lors d'une requête est détecté au niveau de cette extension. Comme résultat celui-ci est enlevé de la liste des serveurs disponibles. Un autre serveur est choisi, la connexion de côté client est adaptée et la requête est réémise. Lors de la connexion à un serveur substitut, le client et le serveur échangent leur vues sur le groupe.

La vue initiale de groupe de copies est donnée explicitement lors du lancement d'une entité par l'administrateur.

Pour l'exécution atomique de toute requête sur tous les serveurs fonctionnels (et l'obtention d'une cohérence forte) nous avons réalisé l'extension `TransactionMgtExtension`. Une requête vers le serveur se déroule de la manière suivante. Pour le client, elle conserve sa forme initiale. Pour le serveur, la requête est transformée par l'extension en une propagation de l'appel vers les autres serveurs. Pour cela l'extension utilise les fonctions offertes par la `GroupMgtExtension`. Vu que nous ignorons les partitionnements du réseau, un échec de propagation se traduit par une mise à jour du groupe. Pour éviter la propagation des requêtes qui ne modifient pas l'état, une dernière extension (`RWExtension`) au niveau du serveur donne des informations sur la nature des méthodes (lecture/écriture).

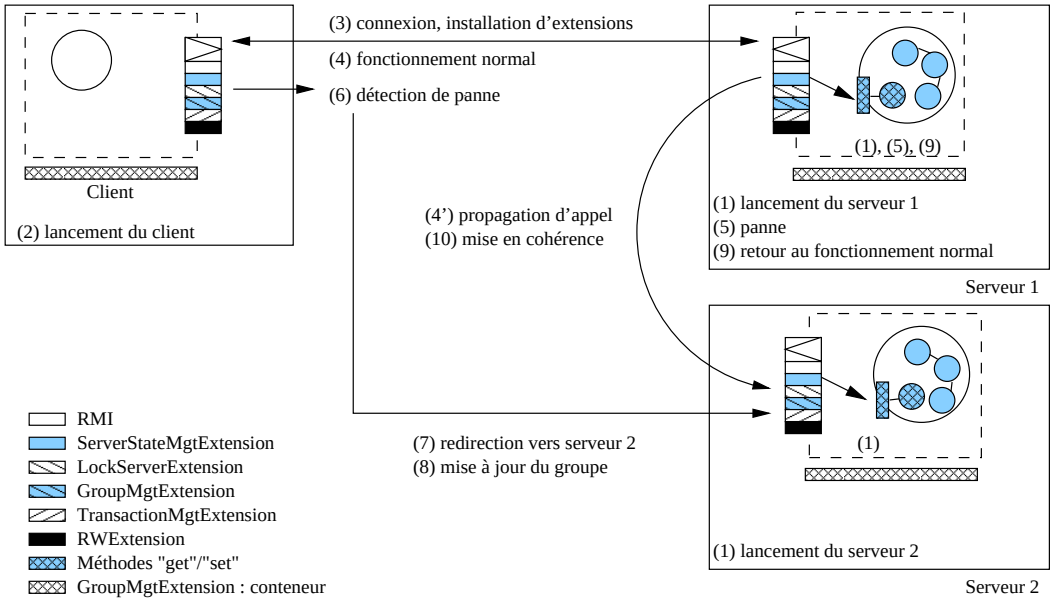

FIG. 6 – *Architecture adaptée de l'application pour la tolérance aux pannes*

4.5. Adaptation pour les usagers mobiles

Dans ce scénario nous introduisons la duplication et la mise en cohérence pour assurer la disponibilité de l'application de réservation en mode déconnecté. Les déconnexions que nous traitons sont explicites et initiées par les usagers mobiles. Une possibilité de gestion des déconnexions non anticipées va être abordée à la fin de cette section.

Au niveau des modifications du code des composants, en plus des méthodes “get” et “set”, nous effectuons une modification de l'interface graphique. Pour inclure la possibilité de déconnexion explicite dans le client, nous rajoutons les deux boutons “Déconnecter” et “Reconnecter”.

Pour implémenter le protocole de duplication et de mise en cohérence nous utilisons toujours les appels ascendants vers les méthodes “get” et “set” rajoutées au composant. Par conséquent nous faisons également usage de l'extension de verrouillage `LockServerExtension` décrite dans la section précédente.

Le protocole de duplication mis en place prend en compte les capacités limitées de stockage des dispositifs mobiles. En effet, en utilisant l'extension `PreferencesMgtExtension`, nous donnons la possibilité de duplication partielle des données gérées par le serveur. Cette extension est invoquée lors de l'action "Déconnecter" du client. En affichant une interface graphique, elle récupère les informations des périodes ainsi que des ressources auxquelles le client est intéressé. Ces informations sont utilisées de côté serveur de l'extension.

L'extension `DisconnexionMgtExtension` est responsable de la mise à jour de la connexion client-serveur lors des changements de mode de travail. Elle implémente deux méthodes : `Disconnect` et `Reconnect`. Le traitement de `Disconnect` côté client consiste à contacter l'extension correspondante côté serveur. L'état de ce dernier est récupéré selon les préférences dans `PreferencesMgtExtension` et en utilisant les méthodes "get". Un serveur local est lancé sur le dispositif mobile. A cause de la localité, il n'est plus nécessaire d'utiliser une communication à distance. Le serveur dupliqué utilise donc une extension `SkeletonLocalx` à la place de `Skeletonx`. La substitution de la référence distante par la référence locale (faite par `DisconnexionMgtExtension`) implique également le changement de la liste d'extensions au niveau du client et l'utilisation de `StubLocalx`. à la place de `Stubx`. Les actions dans `Reconnect` reconstruisent la connexion distante vers le serveur initial. Nous avons implémenté le cas le plus facile ou l'entité mobile se reconnecte au même serveur. Il est possible d'envisager lors de la reconnexion, d'interroger un serveur de noms qui rend la référence du serveur le plus proche.

Lors de la création du serveur local, deux extensions complémentaires lui sont rajoutées. La première, `LogMgtExtension`, est responsable de la trace des opérations effectuées en mode déconnecté. La deuxième, `ConflictMgtExtension`, donne la possibilité à l'utilisateur de fournir des alternatives pour la gestion d'éventuels conflits lors de la reconnexion. Ces informations sont sauvegardées dans le journal avec l'opération correspondante. Pour la mise en cohérence dans le méthode `Reconnect`, elles sont utilisées dans un schéma de réexécution.

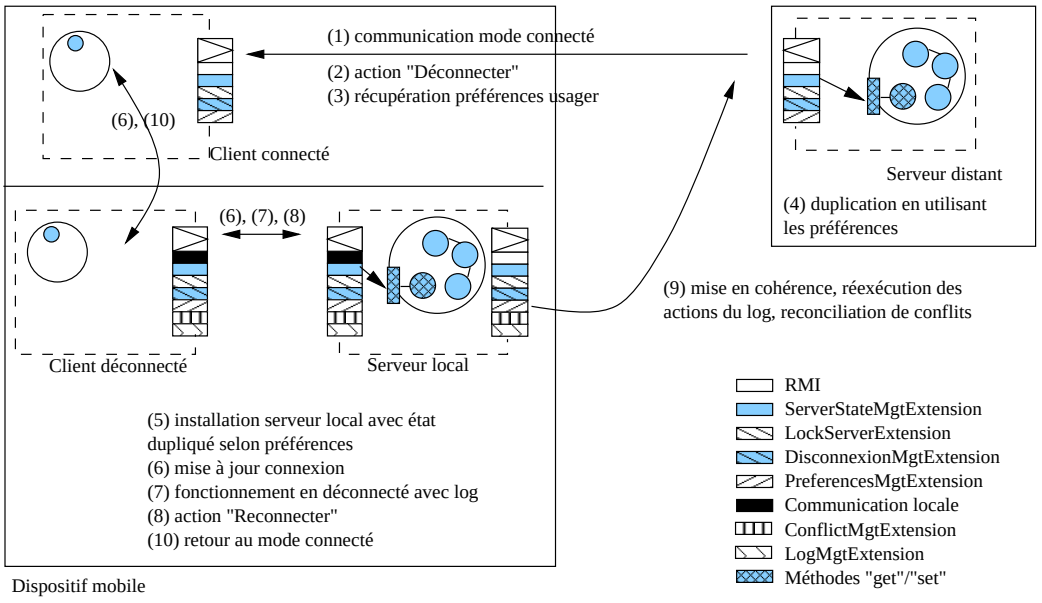

FIG. 7 – *Architecture adaptée de l'application pour les usagers mobiles*

Pour prendre en compte les déconnexions non anticipées, il est envisageable de construire une extension `NetMonitorExtension` qui suit l'état du réseau. En fonction des besoins de l'application, cette extension peut décider à quel moment le réseau devient disponible ou indisponible pour gérer les processus de duplication et de mise en cohérence.

4.6. Evaluation

Considérons dans quelle mesure les objectifs que nous nous posons d'adaptation d'une application existante à des contraintes réseau différentes sont atteints.

La première remarque est qu'aucune adaptation n'est possible si l'application ne suit pas de modèle permettant des adaptations. La plupart des adaptations réalisées dans notre cas ont été rendues possibles grâce aux capacités d'extensibilité et de configuration de la plate-forme JavaPod.

En ce qui concerne l'adaptation sans modification des composants, ces adaptations sont possibles dans une certaine mesure. Nous avons vu dans le cadre précis de la duplication et de la mise en cohérence que les traitements que nous appelons "purement" non fonctionnels peuvent être traités de manière séparée du code. Il s'agit là de traitements de journalisation, de traitements reliés à la communication, etc. En effet, ces traitements peuvent être réalisés par interception et encapsulation des interactions entre composants. Cela ne requiert pas la prise en compte de la structure interne des composants. Les adaptations concernant de tels aspects peuvent donc être faites sans modification du code fonctionnel de l'application. Ce n'est pas le cas des aspects de duplication et de cohérence qui sont fortement liés à l'état des composants et à la sémantique applicative. Il est difficile par conséquent de les qualifier de "non fonctionnels". Leur intégration dans une application existante nécessite des modifications dans le code des composants. Neanmoins, ces modifications ne concernent que l'insertion des traitements de capture/réstauration d'état. En s'appuyant sur ces primitives, la programmation des protocoles de duplication et de mise en cohérence peut être faite de façon modulaire.

Les modifications dans le code des composants pour l'intégration de la duplication et de la mise en cohérence, ainsi que le traitement des différentes contraintes réseau n'ont pas été faits dans notre implémentation selon un modèle précis. Pour le moment nous n'avons pas de mécanisme qui permettrait de spécifier le type de duplication/cohérence voulues, les contraintes environnementales, et de construire la pile d'extensions automatiquement. Cependant nous croyons que ces modifications peuvent être organisées selon un modèle prenant en compte la structure d'un composant ainsi que les conditions de l'environnement d'exécution. Pour considérer la structure des composants, l'approche des patrons[5], utilisée également dans les EJB, nous semble prometteuse. En imposant des règles sur la structure du code il est possible d'extraire des informations. Les patrons permettent également la génération automatique de code. En ce qui concerne l'environnement d'exécution, différents modèles peuvent être créés. Des efforts ont déjà été faits par exemple dans les architectures traitant la qualité de service des applications[16].

Le mécanisme d'extension de la plate-forme JavaPod n'a été utilisé que partiellement. Dans notre expérience, nous avons réalisé des extensions principalement au niveau des talons et des squelettes. Une telle utilisation s'est avéré relativement simple et nous avons apprécié la facilité de manipulation de la composition linéaire des propriétés. La possibilité d'extension à trois niveaux (serveur, conteneur, stub/squelette) nous a semblé difficile à appliquer. Il n'est pas clair comment les aspects de duplication et de mise en cohérence doivent être décomposés selon ces trois niveaux.

En ce qui concerne le modèle à composants JavaPod, nous aurions apprécié un modèle composite afin de mieux structurer le code de gestion de la duplication et de la mise en cohérence.

5. Conclusion et perspectives

Nous avons présenté dans cet article notre expérience d'adaptation d'applications réparties à base de composants. Nous nous sommes intéressés aux adaptations permettant d'ajouter aux applications des traitements garantissant leur disponibilité dans des environnements de contraintes réseau différentes. Nous avons implémenté deux adaptations d'une application existante : l'une garantissant la tolérance aux pannes, la deuxième traitant les cas de déconnexion d'usagers mobiles. La plupart des aspects considérés peuvent être rajoutés et gérés séparément du code fonctionnel. Cependant, la spécificité de l'aspect de duplication et de mise en cohérence ne permet pas la réalisation des adaptations sans vraiment toucher le code fonctionnel des composants. Néanmoins, il est intéressant de poursuivre la voie entreprise et d'essayer de répondre à des questions concernant la structuration et la minimisation des modifications. Un objectif à plus long terme serait la définition de modèle de duplication et de mise en cohérence pour la disponibilité. Des aspects à considérer seront l'influence de la structure des composants, ainsi que les dépendances de l'environnement d'exécution. Le besoin de description des composants et de l'environnement pour ce modèle pourront s'inspirer de la logique des patrons et des formalisations existantes des réseaux.

Bibliographie

1. E.Bruneton, M.Riveill. JavaPod : une plate-forme à composants adaptable et extensible, Rapport de recherche INRIA RA RR-3850. Inria Rhône-Alpes, janvier 2000. http://sirac.inrialpes.fr/Biblio/rapports.html.
2. A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, et B. Welch. The Bayou Architecture: Support for Data Sharing Among Mobile Users. Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, pages 2-7, December 1994.
3. G.Eddon, H.Eddon. Inside Distributed COM. Microsoft Press 1998.
4. T. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies, et P. Robin. Supporting adaptive multimedia applications through open bindings. Proceedings of International Conference on Configurable Distributed Systems (ICCDS '98), Annapolis, Maryland, USA, May 1998.
5. E.Gamma, R.Helm, R.Johnson et J.Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
6. R. Hayton, A. Herbert, et D. Donaldson. Flexinet: a flexible, component oriented middleware system. SIGOPS'98, Sintra, Portugal, September 1998. http://dedanann.dsg.cs.tcd.ie/ vjcahill/-sigops98/papers.html.
7. ITU-T Recommendation X.901 — ISO/IEC 10746. Reference Model - Open Distributed Processing.1995. http://www.dstc.edu.au/Research/Projects/ODP/standards.html.
8. A.Joseph, A.deLespinasse, J.Tauber, D.Gifford, et M.Kaashoek. Rover: A toolkit for mobile information access. Proceedings of the 15th Symposium on Operating Systems Principles (SOSP), December 1995.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, et J. Irwin. Aspect-Oriented Programming. ECOOP'97, Jyväskylä, Finland, June 1997. http://www.parc.xerox.com/spl/-groups/eca/pubs/complete.html.
10. C.Lopes, G. Kiczales, Recent Developments in AspectJ. ECOOP'98 AOP Workshop, Brussels, Belgium, July 1998. http://wwwtrese.cs.utwente.nl/aop-ecoop98/position.html.
11. Object Management Group. CORBA Component Model RFP. Request for Proposal. OMG Document orbos/96-06-12.
12. Object Management Group. The Common Object Request Broker:Architecture and Specification, Revision 2.4. OMG October 2000. ftp://ftp.omg.org/pub/docs/formal/00-10-02.pdf
13. Sun Microsystems. Java 2 SDK, Standard Edition. Sun Microsystems. http://java.sun.com/products/jdk/1.2/.
14. Sun Microsystems. Enterprise Java Beans (EJB (TM)) Specification Version: 2.0. Sun Microsystems. October 2000. http://java.sun.com/products/ejb/.
15. A.Tanenbaum. Distributed operating systems. Prentice-Hall, 1995.
16. R.Vanegas, J.Zinky, J.Loyall, D.Karr, R.Schantz, D.Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), 15-18 September 1998