Kalimucho : Plateforme de supervision d'applications sensibles au contexte

Keling Da, Marc Dalmau, Philippe Roose
LIUPPA/IUT de Bayonne
2, Allée du Parc Montaury
64600 Anglet, France
{Prenom.Nom}@iutbayonne.univ-pau.fr

Résumé. Le développement d'applications ubiquitaires est particulièrement complexe. Au-delà de l'aspect dynamique de telles applications, l'évolution de l'informatique vers la multiplication des terminaux d'accès mobiles ne facilite pas les choses. Une solution pour simplifier le développement et l'exploitation de telles applications est d'utiliser des plateformes logicielles dédiées au déploiement et à l'adaptation des applications et gérant l'hétérogénéité des périphériques. Elles permettent aux concepteurs de se focaliser sur les aspects métiers et facilitent la réutilisation. C'est dans cette optique qu'a été conçue et développée la plateforme Kalimucho. Elle permet l'exécution et la supervision d'applications à base de composants logiciels et offre aux applications un accès à leur contexte d'exécution.

1. Introduction

Les récentes avancées technologiques de ces dernières années ont mis l'accent sur la démocratisation des réseaux sans-fil et sur la miniaturisation des appareils de communication. Actuellement nous pouvons trouver sur le marché une multitude d'appareils de plus en plus légers, compacts, mobiles et dotés de capteurs et de divers moyens de communication sans-fil tels que les téléphones portables, les *smartphones*, les tablettes, les ordinateurs portables ou encore les capteurs.

De plus nous devons faire face à une demande grandissante pour des services de plus en plus riches et personnalisés. Le défi est de pouvoir proposer des applications qui s'adaptent tant aux souhaits des utilisateurs qu'à l'environnement physique. Ce type d'appareils mobiles a la capacité de pouvoir rendre compte de son environnement matériel et logiciel mais également, avec l'arrivée de périphériques tels que les capteurs sans fils ou les capteurs intégrés aux téléphones portables, de pouvoir mesurer des grandeurs physiques comme la température, la pression, la vitesse de déplacement ... L'intégration de tels appareils dans les applications peut permettre de proposer aux utilisateurs des services mieux adaptés à leur situation courante. Cependant, ces appareils possèdent des caractéristiques (autonomie énergétique, mobilité, ressources limitées) qui nécessitent l'adaptation des applications ainsi que des services rendus par celles-ci pour assurer un fonctionnement correct pendant une durée suffisante.

La solution que nous proposons est une plateforme logicielle hébergée sur chaque dispositif physique (PC, Smartphone, tablette). Elle surveille l'utilisation des ressources matérielles (batterie, mémoire, CPU) et le contexte d'exécution (réseau, besoins des utilisateurs, règles d'usage de l'application, etc.). En effet, lorsqu'un périphérique ne possède plus suffisamment de batterie, ce sont tous les services qu'il offre qui cessent de fonctionner impliquant une rupture de service. La réduction de la consommation d'énergie d'un périphérique peut se faire en répartissant la charge de l'application sur les périphériques voisins dans le cas où les traitements distribués ne demandent que peu de connexions réseaux (la répartition a l'effet inverse sinon). Il faut donc pouvoir proposer des solutions telles que la délocalisation de services en cours d'exécution. Notre choix se porte donc vers des applications modulaires à base de composants logiciels distribués. Cette modularité permet de proposer des solutions ad hoc reconfigurables à chaud et garantissant la continuité des applications et leur pérennité dans le temps. L'application est, dans un premier temps, conçue comme un ensemble de fonctionnalités interconnectées. Chaque fonctionnalité est elle-même constituée d'un ensemble de composants logiciels reliés par des connecteurs. Ces fonctionnalités peuvent être réalisées de différentes façons à partir d'assemblages de composants différents. La plateforme dispose donc de plusieurs décompositions fonctionnelles correspondant aux diverses configurations de l'architecture. Il est à noter qu'à chacun de ces assemblages correspond une qualité de service (Laplace, 2007).

Nous avons choisi d'utiliser une plate-forme d'exécution qui connaît l'application en cours et son contexte - réflexivité - (Sousa, 2002). Cette plateforme doit être distribuée sur l'ensemble des dispositifs impliqués. La plateforme Kalimucho peut assurer la continuité de service tout en prenant en compte la pérennité globale de l'application. Une telle solution permet au concepteur de définir un ensemble de fonctionnalités dont certaines sont substituables et à l'utilisateur de disposer de ces fonctionnalités selon ses besoins. Il est donc nécessaire de capturer tous les changements de contexte qu'ils soient liés aux besoins des utilisateurs aux ressources ou à la mobilité puis d'interpréter ces changements afin de réagir de la meilleure façon. Les travaux présentés ici concernent la plateforme en tant que support d'exécution d'applications capable de mettre en oeuvre des reconfigurations en cours d'exécution et de capturer et d'interpréter le contexte d'exécution et environnemental.

Cet article est organisé comme suit : la première partie présente un état de l'art des plateformes logicielles et des types d'adaptation possibles. Nous présenterons plus spécialement les plateformes spécialisées dans l'adaptation structurelle des applications. Dans un second temps, nous présenterons l'architecture générale des applications pour Kalimucho et, en particulier, les modèles de composants et de connecteurs utilisés. Enfin nous décrirons l'architecture de la plateforme Kalimucho permettant l'adaptation structurelle des applications et la prise en compte du contexte.

2. Travaux connexes

La particularité des systèmes sensibles au contexte est qu'ils réagissent aux changements du contexte afin de fournir à l'utilisateur des services adaptés à la situation. On distingue cinq types d'adaptations. La première est l'adaptation de contenu. En fonction des situations, les données sont modifiées pour ne présenter à l'utilisateur que celles qui sont pertinentes à sa

situation (Dom, 2012). La deuxième est l'adaptation de présentation et touche le domaine des IHM. En fonction du statut hiérarchique de l'utilisateur, l'interface de l'application présentera ou non une information et proposera ou non une fonctionnalité (Gabillon, 2001). Le troisième type est l'adaptation de comportement qui se traduit par l'adaptation des fonctionnalités fournies par un composant ou un service (Peyman, 2008). Le quatrième type est l'adaptation structurelle. Elle vise à modifier la composition de l'application et/ou les connexions entre les différents composants dans le but d'obtenir une application dont le comportement reste inchangé. C'est l'adaptation la plus utilisée actuellement dans le domaine des applications distribuées à base de composants. Enfin, le dernier est l'adaptation de déploiement. Il vise à proposer des déploiements qui prennent en compte les propriétés des périphériques supportant l'application. C'est une adaptation de plus en plus utilisée pour faire face aux problèmes engendrés par les limitations matérielles des périphériques mobiles et contraints massivement utilisés de nos jours (Gui, 2011).

Les trois premiers types : adaptation de contenu, adaptation de présentation et adaptation de fonctionnalité sont essentiellement tournés vers l'utilisateur. Le contenu et les fonctionnalités sont adaptés en fonction de ses préférences et la présentation est adaptée en fonction de son statut par exemple. Les adaptations de structure et de déploiement conviennent particulièrement aux contraintes matérielles et réseaux. Les fonctionnalités restent inchangées malgré les changements de contexte. C'est dans cette vision que nous œuvrons.

Le paradigme composant (Szyperski, 1998) (Meyer 2003) (Heineman, 2001) paraît être une solution intéressante pour assurer cette propriété. Une application est alors représentée par un assemblage de composants qu'il est possible de modifier par des opérations élémentaires telles que l'ajout ou la suppression de composants ainsi que de connexions entre ces composants. Ces opérations élémentaires agissent sur sa structure.

Les paragraphes suivants décrivent les travaux traitant respectivement de l'adaptation structurelle et de l'adaptation de déploiement dans le domaine des applications sensibles au contexte et des systèmes pervasifs.

2.1 Adaptation structurelle

WComp (Maurin, 2009) est une approche basée composants légers pour concevoir des services web composites. Elle fournit un cadre de travail permettant de construire des applications sous forme de graphes de services web basés sur le concept de Container. D'autre part, elle fournit un intergiciel basé sur le concept d'Aspects d'Assemblage permettant d'adapter les services web. Le cadre de travail de WComp repose sur le paradigme SLCA, *Service Lightweight Component Architecture* (Maurin, 2009) (Hourdin, 2008). Ce paradigme regroupe les principes du paradigme service web basé événement et du paradigme composant. Le premier apporte l'interopérabilité face à l'hétérogénéité des dispositifs qui composent un système ubiquitaire. L'utilisation de services tels qu'UPnP (Jeronimo, 2005) et DPWS (Shlimmer, 2006) facilitent la communication des services. La réutilisation des services permet l'extensibilité et la communication basée sur les événements garantit une forte réactivité du système. Enfin, un autre avantage du paradigme services web est qu'il permet la mobilité des applications. Le paradigme composant, quant à lui, apporte la dynamique et la flexibilité notamment au niveau de la structure qu'il est possible d'adapter. Le projet MUSIC (Rouvoy, 2009) fait partie des travaux de référence dans l'expérimentation

des processus d'adaptation des applications mobiles. MUSIC est un intergiciel permettant la reconfiguration d'applications mobiles et sensibles au contexte. Le processus d'adaptation défini dans MUSIC repose sur les principes de l'adaptation par planification (*planning based adaptation*). L'adaptation par planification réfère à la capacité de reconfiguration d'une application en réponse aux changements de contexte en exploitant les connaissances de sa composition et des métadonnées de QdS associées à chacun des services la constituant (Floch, 2006).

CADeComp (Ayed, 2008) est un intergiciel pour le déploiement sensible au contexte des applications basées composants. Cet intergiciel étend les services de déploiement existants en y intégrant les capacités d'adaptation nécessaires au domaine des applications mobiles et des périphériques contraints. Il propose un déploiement automatique à la volée et sensible au contexte : une application est installée au moment de son accès et désinstallée juste après la fin de son utilisation. Les applications sont considérées comme une collection de composants distribués sur le réseau et reliés entre eux via des ports. Le déploiement est défini selon cinq paramètres : l'architecture de l'application, le placement des instances des composants, le choix de leur implémentation, les propriétés des composants et leurs dépendances. CADeComp repose sur un modèle de données permettant de décrire le contexte qui agit sur le déploiement et de définir des contrats de déploiement qui associent à chaque situation de contexte toutes les variations possibles des paramètres de déploiement. Le contexte modélise essentiellement les caractéristiques des instances des composants. Ces informations sont collectées lors de la spécification et du développement par le producteur du composant. Elles permettent de spécifier des contraintes sur le placement des composants ainsi que sur les connexions, obligatoires ou optionnelles.

La plateforme à service OSGi (OSGi, 2013) implémente un modèle de composant (appelé *Bundle*). Ces derniers possèdent un cycle de vie leur permettant d'être arrêté, démarrés, mis à jour et désinstallés à chaud. Le service *registry* permet d'enregistrer des bundles en tant que services et ainsi d'en détecter l'apparition ou la suppression. Tout comme OSGi, Kalimucho implémente un modèle de composants appelé Osagaia, possédant un cycle de vie comparable mais y ajoute la notion de migration. OSGi est basé sur la découverte de services alors que Kalimucho met en œuvre des schémas de déploiement prédéfinis au moment de la conception. La plateforme Kalimucho établit les connexions entre composants via des connecteurs de première classe appelés Korrontea implémentant une partie métier. Ces derniers permettent le traitement à la volée des données transmises. Le tableau suivant présente une comparaison de Kalimucho avec les plateformes OsGi et CaDeComp en termes de modèle de composants, de communication entre les composants et de mécanismes de déploiement et de reconfiguration contextuels.

Critère	OsGi	CaDeComp	Kalimucho
Modèle de composant	Bundle	CCM	Osagaia
Liens entre composants	Services + paquetages importés/exportés	ports	Connecteur Korrontea
Déploiement contextuel	NON	OUI	OUI
Prise en compte du contexte en cours d'exécution	NON	NON	OUI
Reconfiguration	Ajout Suppression	Ajout Suppression	Ajout Suppression Migration

Les approches que nous avons exposées dans ce paragraphe montrent différentes façons de considérer l'adaptation structurelle des applications : canevas logiciel, intergiciel, plateforme d'exécution. Nous retiendrons de ces différentes approches l'importance de la flexibilité permettant d'agir sur la structure de l'application.

Notre approche consiste à proposer une plateforme qui est la seule application devant être installée sur chaque périphérique. Ceci est particulièrement intéressant pour les périphériques mobiles contraints comme les *smartphones*. Lorsqu'elle est installée, il suffit de lui fournir une décomposition fonctionnelle d'application pour qu'elle puisse la déployer en allant chercher les composants sur des dépôts disponibles et identifiés n'importe où sur le réseau. Ces composants sont ensuite dynamiquement déployés sur les différents périphériques hébergeant une plateforme Kalimucho pour être ensuite supprimés une fois l'application terminée, ne laissant ainsi aucun résident sur les dits périphériques. C'est ce qui s'appelle communément *short-lived installation*.

3. Exemple d'application répartie dynamique

Le déploiement et redéploiement dynamique d'applications nécessitent qu'elles aient une architecture supervisable. Dans notre modèle, une application est composée d'un ou de plusieurs services et chaque service peut être réalisé par un ou plusieurs assemblages de composants reliés par des connecteurs. L'état d'une application est alors constitué de l'ensemble des états des composants, des périphériques, des connecteurs et de l'environnement. La plate-forme doit recueillir toutes ces données afin de les traiter et de mener les actions de reconfiguration qui conviennent. Adapter l'application consiste alors à ajouter, supprimer ou modifier des services. Nous identifions trois moyens d'action permettant de modifier la structure d'une application :

La migration de service : consiste à modifier la distribution d'un service sans en modifier les composants. Les composants sont déplacés, avec leurs états, d'un périphérique vers un autre sans remplacement ni ajout ni suppression.

Le déploiement ou redéploiement d'un service : consiste à proposer un nouvel assemblage de composants pour réaliser ce service ou un service équivalent.

La modification d'un service consiste à remplacer un ou plusieurs de ses composants par d'autres.

Afin d'illustrer nos propos, nous allons présenter dans un exemple simple les principales fonctionnalités que peut permettre la plateforme Kalimucho. Afin de ne pas surcharger la présentation, l'exemple choisi n'utilise pas la migration de composants mais ce mécanisme sera décrit en 4.2.2.

Dans le cadre du projet ANR MOANO[1], nous avons développé une application de prise de notes à destination des jardiniers d'un parc botanique. Dans le parc Mosaic de plusieurs hectares, les jardiniers ont besoin de réaliser un suivi des plantes et de leurs évolutions (*prises de photos, prises de notes, localisation, etc.*).

Sur cette base, nous proposons un exemple simplifié d'application hébergée sur des *smartphones* permettant de prendre des notes géolocalisées, écrites ou orales. Chaque plante étant accompagnée d'un panneau identificateur par QRCode, le scan de ces QRCodes facilite la désignation des plantes dans les notes. Pour des raisons pratiques (position du preneur de notes par rapport au panneau), il est possible de déléguer ce scan à un autre jardinier. Il est également possible de permettre à un autre jardinier de compléter une prise de notes.

Lorsqu'un jardinier arrive dans le parc avec Kalimucho en cours d'exécution sur son *smartphone*, un composant d'IHM lui est déployé, offrant 3 boutons (Edition/Sélection d'une note, enregistrement d'une note vocale, activation de la géolocalisation).

S'il sélectionne le premier bouton, un composant de choix de note écrite ou orale est déployé lui permettant de sélectionner une note déjà présente tandis qu'un composant d'accès à la base de données des notes est déployé sur le PC central du parc. Ces deux composants sont reliés entre eux par des connecteurs. S'il choisit une note écrite, un composant d'édition est déployé. Lorsqu'il souhaite introduire un scan de QRCode dans sa note, la liste des périphériques des autres jardiniers présents dans le parc lui est proposée. Il peut alors choisir de scanner lui-même la note ou de déléguer cette tâche à l'un de ses collègues. Dans ce dernier cas, un composant d'alerte (vibreur+message) est déployé sur le terminal de ce collègue afin de l'avertir de la demande. Si le collègue accepte, ce composant d'alerte est remplacé par un composant de scan de QRCode qui est relié par connecteur au composant de prise de notes du premier jardinier. Dès que le scan a été fait et le résultat inséré dans la note, le composant de scan et le connecteur sont automatiquement supprimés.

Si le jardinier a mis en marche la géolocalisation, lorsqu'il sauvegarde sa note sur le serveur du parc, elle sera automatiquement géolocalisée. Lors d'une consultation ultérieure de cette note, cette géolocalisation ainsi que la date seront accessibles.

[1] Ces travaux sont partiellement soutenus par le projet ANR MOANO : http://moano.liuppa.univ-pau.fr

4. Architecture générale de la plateforme

4.1 Implémentation des applications

Kalimucho, telle que nous l'avons définie, est une plate-forme de supervision qui, par sa distribution sur tous les périphériques de l'application, a connaissance des composants et connecteurs déployés et peut récupérer les informations de contexte que ceux-ci lui transmettent. C'est en fonction de ces informations que les décisions de reconfiguration pourront être prises.

Pour cela elle doit surveiller le fonctionnement des composants et la circulation des flux de données entre ces composants. Afin de recueillir des informations sur ces deux entités, nous avons choisi d'utiliser des conteneurs qui, de plus, apportent une solution à la gestion de l'hétérogénéité matérielle et logicielle des périphériques ainsi qu'à la mobilité des périphériques.

4.2 Modèle de composants OSAGAIA

Le modèle Osagaia (Bouix, 2005) propose de séparer la logique métier contenue dans un composant métier de la supervision gérée par un conteneur. Le Composant Métier (CM) peut recevoir plusieurs flux de données en entrée et produire plusieurs flux de sortie et chaque flux de sortie peut être dupliqué. Le conteneur encapsule un et un seul composant métier et implémente les propriétés non-fonctionnelles telles que la gestion du cycle de vie, la récupération des informations de qualité de service et la gestion des communications.

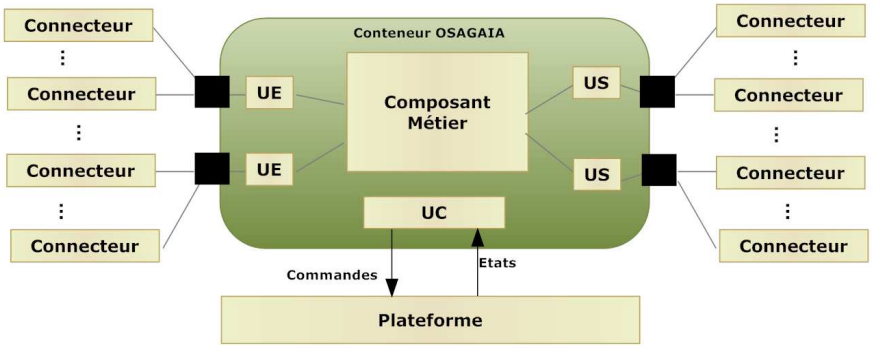

FIG. 1 - *Modèle de Composant OSAGAIA*

Le conteneur possède trois types unités : l'Unité d'entrée (UE), l'Unité de sortie (US) et l'Unité de contrôle (UC). Chaque unité d'entrée ou de sortie peut être connectée à un ou plusieurs connecteurs. Elles permettent au CM de lire et écrire des données provenant ou à destination d'autres composants métier. Le composant métier peut ainsi lire des données via les UE, effectuer son traitement et écrire les résultats dans les US. L'Unité de contrôle (UC) permet à la plate-forme de superviser le conteneur.

4.2.1 Les composants métier

Le concepteur de l'application écrit les composants métiers (CM) qui seront encapsulés par la plate-forme dans un conteneur Osagaia qui en contrôlera le cycle de vie. Le principe retenu est celui que l'on trouve pour les applets, les *midlets* ou les activités sur Android : Le cycle de vie correspond à l'appel de méthodes que le développeur doit surcharger.

4.2.2 Cycle de vie d'un CM

Le cycle de vie d'un composant répond aux trois types d'actions que la plateforme doit pouvoir effectuer : la création, la suppression et la migration.

Création d'un composant : La création d'un composant consiste en l'instanciation d'un objet de la classe indiquée, à son encapsulation dans un conteneur puis à la connexion de ses flux d'entrée et de sortie.

Suppression d'un composant : La suppression d'un composant consiste en l'arrêt du composant métier encapsulé puis la suppression du conteneur. Ses flux d'E/S restent en attente d'un nouveau composant ou d'être, à leur tour, supprimés.

Migration d'un composant : Lorsqu'un composant s'exécutant sur un hôte A doit être migré vers un hôte B, la plate-forme en A arrête le composant (comme lors d'une suppression), puis l'envoie à la plate-forme en B par sérialisation de ses propriétés. Après quoi elle détourne l'ensemble des connecteurs reliés à ce composant vers sa nouvelle destination (B). Le composant reçu en B est encapsulé dans un conteneur puis la plate-forme attend que ses flux d'entrée et de sortie soient à nouveau connectés.

Le cycle de vie d'un CM est représenté par la figure 2 :

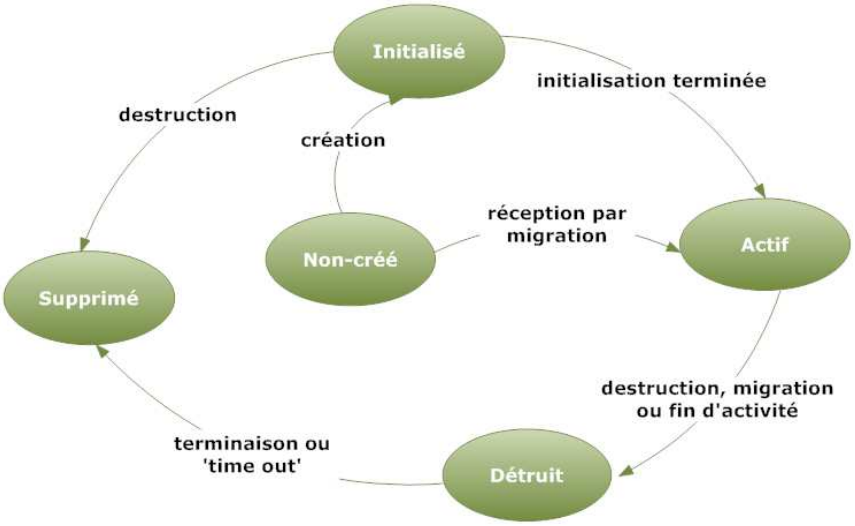

FIG. 2 - *Cycle de vie d'un CM*

4.2.3 Les flux d'entrée et de sortie

Le composant métier accède à ses flux d'entrée et de sortie par le biais de services offerts par son conteneur. Les données lues ou écrites sont des objets sérialisables définis par le concepteur. Comme les flux peuvent transporter des données continues ou discontinues, deux types de moyens d'accès sont proposés, le premier est adapté aux flux continus et le second aux flux non continus :

Accès direct : soit le CM lit dans le flux de son choix par une opération bloquante jusqu'à ce qu'une donnée soit disponible. Soit il veut récupérer la première donnée disponible dans l'un de ses flux d'entrée par une opération qui le bloque jusqu'à ce qu'une donnée soit disponible sur l'un des flux d'entrée.

Accès par événement : le CM associe un événement (désigné par un numéro) à un flux. Un événement (classe « BCEvent ») lui sera automatiquement transmis dès qu'une donnée sera présente sur ce flux. Cet événement contient la donnée reçue. L'utilisation des événements est laissée au libre arbitre du concepteur de CM qui peut en placer sur certaines entrées, sur toutes ou sur aucune. Un événement associé à une entrée peut être supprimé à tout moment par le CM qui retournera alors au mode d'accès direct.

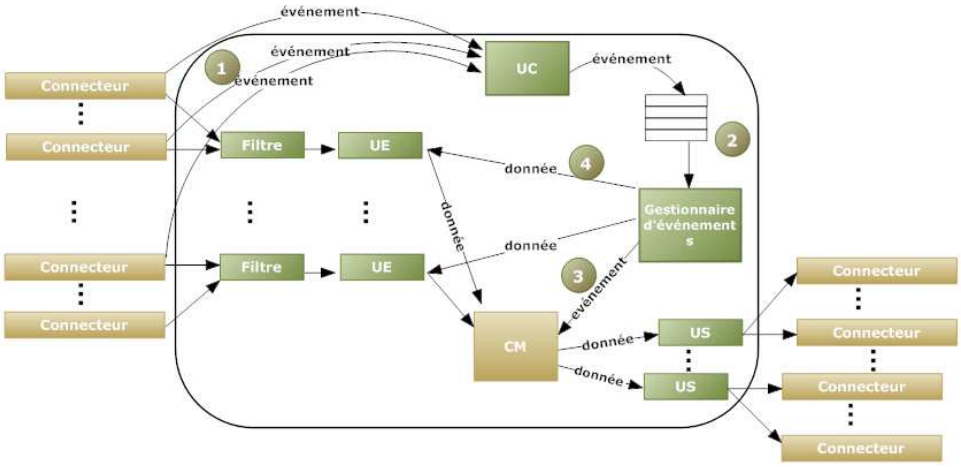

FIG. 3 - *Structure interne d'un conteneur de composant métier*

La figure 3 montre la structure interne d'un conteneur de composant métier.
Lorsqu'un connecteur d'entrée dispose d'une nouvelle donnée, il en informe l'UC du conteneur ① qui transmet cette information au gestionnaire d'événements. Celui-ci gère une file d'attente des événements à transmettre ② et les envoie au CM ③ après avoir récupéré la donnée correspondante dans l'UE ④. Lorsque le CM doit être arrêté, la plate-forme arrête ce gestionnaire par les mêmes mécanismes que le CM lui-même (exceptions). L'écriture dans un flux de sortie se fait par une opération d'accès direct qui ne peut être bloquante que si aucun flux n'est connecté à cette sortie.

Le démarrage d'un CM par la plateforme n'est pas assujetti à l'existence des connecteurs qui lui sont reliés. Son exécution peut se poursuivre tant qu'il ne tente pas d'accéder à un flux d'entrée ou de sortie.

Le fonctionnement des flux d'entrée est géré par les UE du conteneur. Celles-ci peuvent être arrêtées ou en marche et connectées on non connectées. Lorsqu'une UE est arrêtée, la prochaine tentative de lecture par le CM provoquera une exception qui arrêtera le CM. Ainsi, quand un CM doit être arrêté ou migré, la plate-forme place toutes ses unités d'entrée en mode arrêté. Lorsqu'une UE est en marche elle peut être ou non reliée à un connecteur. A chaque tentative de lecture, l'UE vérifie qu'elle soit toujours reliée à un connecteur, si c'est le cas elle tente de récupérer une donnée. Dans le cas contraire elle bloque le CM jusqu'à ce qu'elle soit à nouveau connectée ou que la plate-forme l'arrête. Lors d'une tentative de lecture dans le connecteur en entrée, le CM est bloqué jusqu'à ce qu'une donnée soit présente. Pendant que le CM est bloqué, l'UE s'assure que le connecteur reste présent. Dans le cas où il viendrait à disparaître ou serait déconnecté de cette entrée, l'UE suspend la lecture en cours et attend qu'une nouvelle connexion soit réalisée. Dès que c'est le cas, elle reprend la lecture suspendue. Bien entendu, au cours de toutes ces étapes, la plate-forme peut à tout moment arrêter le CM. Un ensemble de sémaphores est utilisé de façon à bloquer le CM et à permettre l'exécution des autres CMs pendant que l'un d'entre eux est en attente soit d'un connecteur soit d'une donnée.

Les flux de sortie peuvent être dupliqués autant de fois qu'on le souhaite pour être transmis à plusieurs CMs. Cette duplication est totalement transparente au CM, de sorte que l'ajout ou le retrait d'un connecteur n'a pas d'incidence sur son fonctionnement. Seul le cas où il n'y a plus de flux en sortie provoquera le blocage du CM dès qu'il souhaitera produire une donnée sur cette sortie, il sera automatiquement relancé dès la connexion d'un nouveau connecteur et la donnée en attente y sera écrite. Tout comme une UE, une US peut être arrêtée ou en marche et connectée ou non. Lorsqu'elle est arrêtée, la prochaine tentative d'écriture par le CM provoquera une exception qui l'arrêtera à son tour.

Ce mode de fonctionnement permet de dynamiquement supprimer/déconnecter/reconnecter des flux d'entrée/sortie sans perturber le fonctionnement des CM qui sont seulement suspendus lorsqu'ils tentent d'y accéder et relancés lorsqu'ils sont à nouveau disponibles.

Cette possibilité de connexion/déconnexion dynamique est particulièrement adaptée aux environnements mobiles où de telles situations peuvent se produire fréquemment.

4.3 Modèle de Connecteur : KORRONTEA

Pour relier les E/S des composants, le mécanisme de connecteur est généralement admis dans le domaine des architectures logicielles à base de composants.
Korrontea fournit un conteneur pour les connecteurs. La fonctionnalité principale d'un connecteur est de relier deux composants et de faire circuler l'information entre eux. De la même manière qu'un composant Osagaia, un connecteur est une entité de première classe. Il ne se limite pas à la mise en œuvre d'un ou de plusieurs modes de communication (*Client/Serveur, Pipe & Filter, etc.*). Nous souhaitons également qu'il puisse agir sur l'information elle-même de façon à faire de l'adaptation de données à la volée. Il est donc nécessaire de le doter, lui aussi, d'un composant métier.

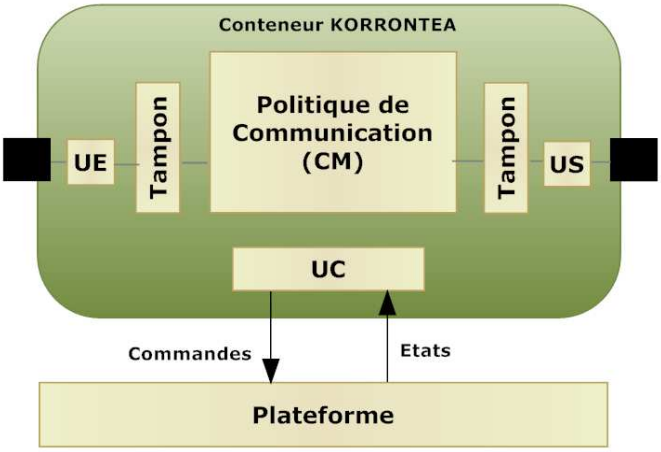

FIG. 4 - *Modèle de Connecteur Korrontea*

Le traitement implémente une politique de communication (*assurant la partie métier du connecteur : synchronisation, communication temps/réel, etc.*). Il est encapsulé dans un conteneur doté d'une UE, d'une US et d'une UC de façon comparable au conteneur d'un composant métier. Cependant, contrairement au conteneur de composant métier, le conteneur de connecteur n'accepte qu'une seule entrée et qu'une seule sortie. Enfin l'UC permet à la plate-forme de superviser le connecteur. L'UE et l'US ont la particularité d'être reliées à des tampons permettant d'éviter les pertes de données lors des reconfigurations. Les données sont stockées dans ces tampons jusqu'à ce qu'elles puissent être transférées. De plus, ils permettent de détecter des situations pouvant engendrer des reconfigurations. La saturation d'un tampon de sortie indique que le composant suivant ne traite pas les données assez rapidement ou que la liaison réseau est lente si le composant suivant est distant. Celle du tampon d'entrée indique un dysfonctionnement du composant contenu dans le connecteur. De même, le fait que ces tampons soient vides permet de détecter la fluidité de la circulation des données et provoquer une restructuration visant à augmenter la qualité du service.

4.3.1 Les types de connecteurs

Le connecteur Korrontea informe les capteurs logiques de la plateforme de la circulation des données dans l'application. Il lève des alarmes quand des données s'accumulent dans ses tampons mais également quand la circulation des données devient fluide après une accumulation. Ceci permet de surveiller la circulation de données dans un hôte ou entre deux hôtes sur le réseau. Les niveaux de ces alarmes sont paramétrables.

Un connecteur peut être utilisé pour relier deux composants sur la même machine (on parle alors de connecteur interne) ou deux composants placés sur des machines différentes (on parle alors de connecteur distribué). En raison de l'hétérogénéité des réseaux (*wifi, 3G, etc.*) il est possible que deux machines devant être reliées par un connecteur ne puissent pas communiquer directement. Dans ce cas la plateforme se charge de trouver un hôte pouvant servir de passerelle entre les deux types de réseaux (cf 5.2.4). La plateforme Kalimucho propose donc trois types de connecteurs :

Connecteur Interne : un connecteur est interne à un hôte lorsqu'il relie deux CMs sur la même machine.

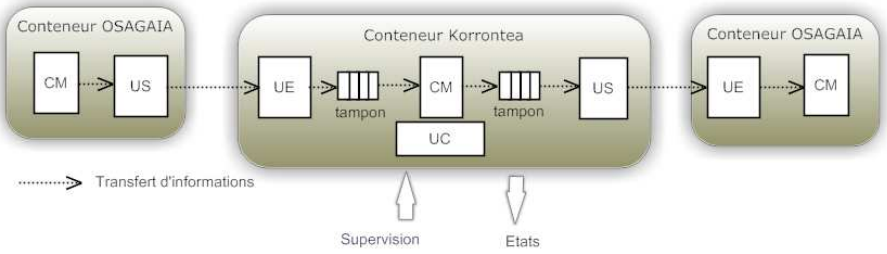

FIG. 5 - *Connecteur Interne*

Connecteur distribué : un connecteur peut être distribué sur deux hôtes (*deux smartphones, deux PCs, un smartphone et un PC, etc.*). Il établit alors une communication par réseau.

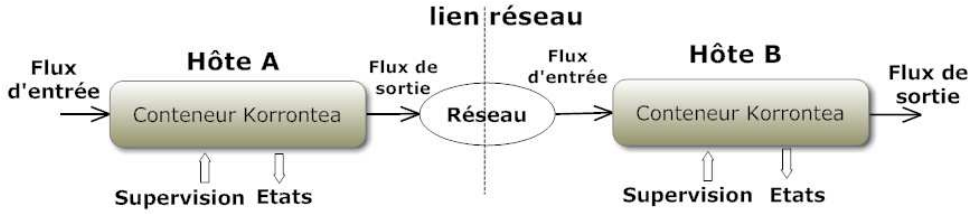

FIG. *6 - Connecteur distribué sur le réseau*

Quand la plateforme sur l'hôte A reçoit une commande de création d'un connecteur de A vers B, elle cherche une route vers B (cf. 5.2.4). Si la route est directe, A envoie une commande à la plateforme sur B pour qu'elle crée un conteneur Korontea. Pendant ce temps elle crée elle même un conteneur Korrontea. Le connecteur final est un élément en deux parties avec un thread client d'un coté et un thread serveur de l'autre.

Connecteur relais : un connecteur peut également être utilisé comme relais sur le réseau. Ce mécanisme n'a pas pour objectif de créer des routes complexes (multi hop) mais des ponts entre deux réseaux de types différents c'est pourquoi Kalimucho n'utilise qu'un seul hôte relais par connecteur. Ainsi, pour relier deux CM sur deux hôtes utilisant des réseaux différents, Kalimucho crée un connecteur relais sur un hôte ayant simultanément accès à ces deux réseaux.

La figure 7 présente un connecteur relais qui relie deux CM Osagaia à travers une connexion 3G et une connexion wifi :

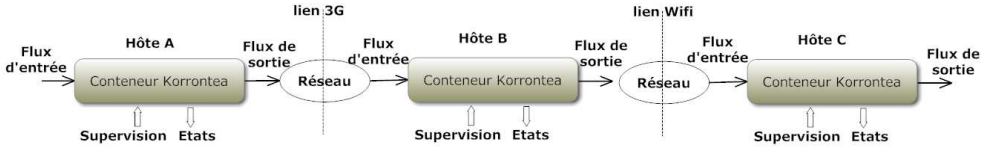

FIG. 7 - *Connecteur relais*

4.3.2 Les objets échangés

Les connecteurs de Kalimucho assurent les communications entre CMs par sérialisation des objets échangés. Kalimucho offre une classe *Sample* de laquelle doivent hériter toutes les classes d'objets échangés au travers des connecteurs. Aux données sont ajoutées des informations indiquant leur date de création et le flux sur lequel ils ont été transportés. Le code des classes des CMs et des objets échangés n'est pas résidant sur chacun des hôtes, il est téléchargé à la demande par Kalimucho. Toutefois, pour pouvoir transporter des objets par sérialisation il faut disposer des classes de ces objets ce qui n'est pas le cas sur la machine qui accueille un connecteur relais. La solution retenue pour résoudre ce problème est d'encapsuler les classes des objets transmis dans une classe de Kalimucho (*EncapsulatedSample*) de sorte que les connecteurs ne manipulent que des objets de cette classe.

5. Kalimucho

Les paragraphes qui suivent décrivent le fonctionnement de la plateforme.

5.1 Le middleware

Le middleware prend en charge les communications entre composants en fournissant le support physique des flux d'information. Il est constitué par les connecteurs que la plateforme met en place entre les CMs. La création d'un connecteur peut aboutir, selon la localisation des composants qu'il relie, à la création d'un connecteur interne, d'un connecteur distribué ou d'un connecteur avec relai. Lorsqu'un composant est migré de A vers B, il est arrêté sur A puis sérialisé vers B et enfin ses connecteurs d'entrée et de sortie sont redirigés de façon à ce qu'ils aboutissent maintenant sur B et non sur A. La redirection d'un connecteur peut aboutir à différentes situations selon où étaient jusqu'à lors connectées son entrée et sa sortie. Le tableau suivant présente les différents cas possibles :

Sur A, le connecteur venait de ou allait vers	Sur B, le connecteur vient de ou va vers
A (connecteur interne)	A (connecteur distribué ou par relais)
B (connecteur distribué ou par relais)	B (connecteur interne)
C (connecteur distribué ou par relais)	C (connecteur distribué ou par relais)

La plateforme s'appuie sur ce middleware pour créer, supprimer, migrer, connecter et déconnecter des composants pendant que l'application est en cours de fonctionnement.

5.2 Kernel

Le noyau de la plateforme est présenté sur la figure 8.

Il est constitué de services dont certains peuvent être ou pas installés selon la configuration de la plateforme (plugins). Kalimucho propose les services suivants *:*

- ***Enreg. De services :*** service permettant à la plateforme et à l'application d'accéder aux services offerts par Kalimucho
- ***Supervision :*** service de supervision de l'application exécutant les commandes de création/suppression/migration/connexion /déconnexion de composants
- ***Gest. de code chargé :*** *service* permettant de charger le code des classes correspondant aux CMs et aux objets transférés lorsqu'elles doivent être utilisées. En effet, la partie applicative n'est pas résidente sur les périphériques mobiles : les classes sont chargées lors de la création de la première instance d'un composant et supprimées lors de la suppression de sa dernière instance.
- ***Communication par réseau*** : service permettant aux plateformes de communiquer entre elles et servant de support au middleware (connecteurs)
- ***Plugins :*** services pour l'application permettant aux composants d'accéder à des ressources gérées par la plateforme. Il s'agit soit de ressources classiques (textes, images, …) soit de ressources spécifiques du matériel (GPS, SMS, etc.) soit d'accès à des services web (Google Maps).
- ***Capture de Contexte :*** *service* permettant aux applications et à la plateforme de capturer le contexte à l'aide de capteurs logiques ou physiques.

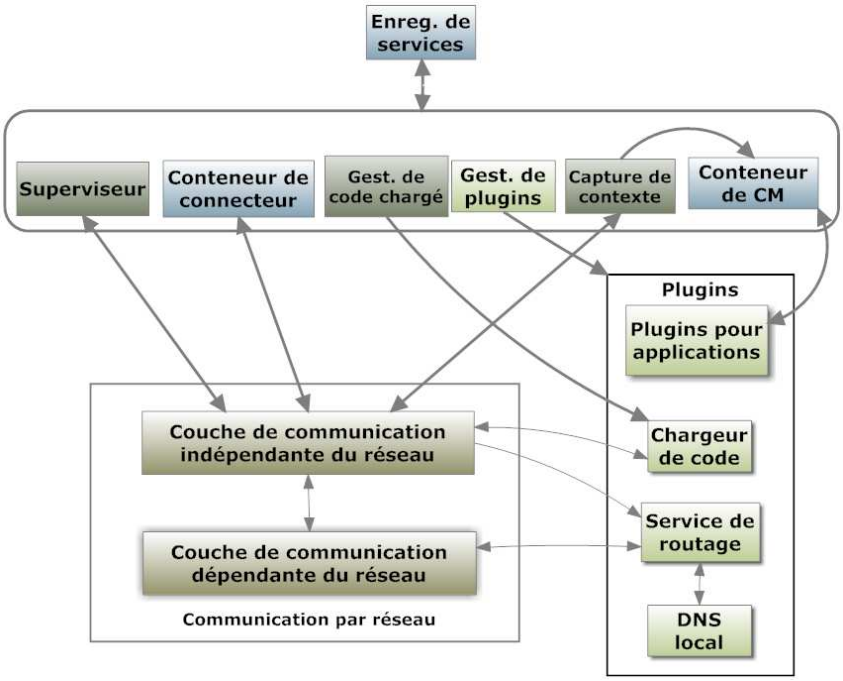

FIG. 8 - *Architecture du noyau de la plateforme*

Nous allons maintenant décrire chacun de ces services en insistant plus particulièrement sur la capture du contexte.

5.2.1 L'enregistreur de services

Il fonctionne comme un *rmiregistry* mais en mode local seulement, c'est-à-dire faisant uniquement référence aux services hébergés sur la plateforme locale au périphérique. Les services de la plateforme y sont enregistrés. Par exemple, lors de la création d'un connecteur distribué, il faut envoyer des commandes à d'autres plateformes, pour ce faire on interrogera l'enregistreur de services pour obtenir le service chargé des communications par réseau. De la même façon le middleware enregistre chaque conteneur Korrontea comme un service qui sera utilisé par la plateforme pour le contrôler et dynamiquement découvert par les conteneurs Osagaia pour établir leur connexion aux connecteurs d'entrée ou de sortie. Enfin chaque conteneur Osagaia enregistre son UC comme un service permettant à la plateforme de contrôler le cycle de vie du CM.

Les CM peuvent accéder aux services de Kalimucho par le biais de l'enregistreur de services, toutefois seuls les services autorisés leur sont accessibles (services pour les applications). Les autres services ne sont accessibles que par la plateforme elle-même.

5.2.2 Le service de supervision

Ce service reçoit et exécute les demandes de déploiement ou de reconfiguration provenant des autres plateformes, des CMs ou de la prise de décision locale de Kalimucho. Ces demandes sont exprimées dans le langage suivant :

Commandes relatives aux composants: il s'agit des commandes de création, suppression, migration, connexion, déconnexion et duplication des flux de sortie des composants. Les composants y sont désignés par leur nom (*nomc*)

- **CreateComponent** nomc classe [entrée1 entrée2 …] [sortie1 sortie2 …]
  - Les listes d'entrée et/ou de sortie peuvent être vides ([null])
  - Une entrée ou une sortie peut être marquée "not_used" pour être utilisée plus tard
- **RemoveComponent** nomc
- **SendComponent** nomc vers
- **DisconnectInputComponent** nomc numéro_d_entrée
- **DisconnectOutputComponent** nomc numéro_de_sortie
- **ReconnectInputComponent** nomc numéro_d_entrée nouvelle_entrée
- **DuplicateOutPutComponent** nomc numéro_de_sortie nouvelle_sortie

Commandes relatives aux connecteurs : il s'agit des commandes de création, suppression et redirection des connecteurs. Les connecteurs y sont désignés par leur nom (*nomk)*

- **CreateConnectorv** nomk depuis vers // depuis et vers peuvent désigner l'hôte local ("internal")
- **RemoveConnector** nom
- **RedirectInputOfConnector** nomk vers
- **RedirectOutputOfConnector** nomk vers

Ce langage définit également des commandes relatives aux états de la plateforme, des conteneurs et de l'hôte qui ne seront pas décrites ici.

5.2.3 Le service de chargement dynamique de code

Le byte code java des composants métiers est chargé dynamiquement lors de chaque création ou migration d'un composant. Ce code est extrait d'un dépôt de composants placé sur les hôtes non contraints (*PCs*). Ce dépôt est structuré en sous dépôts correspondant aux différents types d'hôtes (*PC, périphérique Android*). Chaque composant y est représenté par un fichier *jar* contenant :

- Le byte code de chaque classe nécessaire au fonctionnement de ce composant et des classes des objets échangés;
- Les ressources utilisées par ce composant (images, fichiers, etc.);
- Un *manifest* indiquant le nom de la classe du CM. Ce *manifest* permettra à la plateforme de retrouver le fichier répondant à une demande de chargement de classe.

Un hôte a besoin de récupérer un élément du dépôt lors de la création d'un nouveau CM ainsi que lors de la récupération d'un CM migré. En effet la migration d'un composant se traduit par la sérialisation de ses propriétés mais l'hôte qui reçoit ce composant doit également pouvoir disposer de sa classe (*byte code*).

Quelle que soit la raison pour laquelle un hôte doive récupérer un élément dans le dépôt il envoie un message à tous ses voisins en stipulant le nom de la classe recherchée et son propre type (*Android, PC*). Cette diffusion de message peut se faire, selon ce que permet le type de réseau, en *broadcast*, en multicast ou en point à point. Un hôte recevant une telle demande lui envoie le fichier jar contenant cette classe. Ceci est possible soit parce qu'il gère un dépôt de composants soit parce qu'il s'agit d'un périphérique, du même type que le demandeur, qui détient actuellement ce fichier parce qu'il accueille un composant de cette classe. Le *manifest* contenu dans les fichiers *jar* permet de savoir s'ils contiennent la classe demandée.
Afin de ne pas occuper inutilement de la place en mémoire, le fichier *jar* téléchargé sur un périphérique contraint est supprimé dès que la dernière instance de CM l'utilisant est supprimée. Java permet le chargement dynamique de code par le biais de spécialisations de la classe du chargeur de la machine virtuelle. Le service de chargement dynamique de code de Kalimucho est construit autour d'un chargeur de classes spécifique (*KalimuchoClassLoader*) qui hérite de la classe *ClassLoader*. Sur Android il hérite de la classe *DexClassLoader* car le byte code et la façon de le charger y sont différents. Toute création d'objet par la plate-forme fait appel à ce chargeur de classes.
A chaque CM créé est associé le chargeur de classes qui a servi à le créer de telle sorte que les créations d'objets issues de ce CM puissent se faire au travers de ce chargeur de classes. Le lien entre le CM et son chargeur de classes est mémorisé dans le conteneur de CM. De plus, cette association permet l'accès par un CM aux ressources incluses dans le fichier *jar* qui le contient au travers des méthodes *getResourceAsStream* et *getResourceAsByteArray* de la classe *BCModel* dont il hérite (voir 4.2.1).

5.2.4 Services de communication par réseau

Ces services sont organisés en : Service de communication entre plateformes, Service de recherche de routes et Service de DNS local.

Service de communication entre plateformes : Lorsqu'un message est envoyé par un service de la plateforme, il est placé dans l'une des files d'attente selon sa priorité. De là l'émetteur le transmet au client d'envoi correspondant au réseau approprié en fonction de son destinataire. Si ce client ne peut pas atteindre le destinataire, l'émetteur fait appel au service de routage (s'il est présent) pour qu'il détermine un hôte pouvant servir de relais. Le message est alors transmis à cet hôte qui le fera passer à son destinataire. Lorsqu'un message est reçu, si le destinataire est bien l'hôte local, il est placé dans la boîte à lettres associée au service auquel il est destiné. Dans le cas contraire, il est immédiatement renvoyé de façon à assurer la fonction de relais permettant les passerelles entre types de réseaux différents.

Service de recherche de routes : Parce que Kalimucho peut fonctionner sur plusieurs types de réseaux les hôtes d'une application ne peuvent pas toujours établir de communication directe. Or il est important de pouvoir établir à tout moment une connexion entre chaque plateforme Kalimucho instanciée.

Le service de routage est utilisé par l'émetteur de la plateforme pour trouver, si nécessaire, un relais pour les communications entre plates-formes. Il est également utilisé par le service de supervision pour trouver l'hôte devant accueillir un connecteur relais lorsque deux composants doivent être reliés alors qu'ils ne disposent pas de possibilité de lien direct. Ainsi quand la plateforme sur l'hôte A reçoit une commande pour créer un connecteur avec l'hôte B, A interroge le service de routage pour trouver une route vers B. Cette route peut être directe ou indirecte, dans ce dernier cas un hôte pouvant servir de relai est identifié et un connecteur avec relais est créé.

Service de DNS local : Chaque plateforme gère un DNS local dans lequel sont enregistrées toutes les plateformes avec lesquelles elle est entrée en communication. Pour chaque plateforme connue on y conserve les informations suivantes :

- Identificateur unique de la plateforme
- Liste des adresses connues de cette plateforme sur chacun des réseaux auxquels elle accède
- Décalage d'horloge avec cette plateforme et erreur maximale de la mesure de ce décalage

Lors de chaque réception de message, le DNS enregistre la plateforme émettrice et son adresse. Lors d'échanges de messages de recherche de routes, à la réception de la réponse le DNS calcule le décalage d'horloges (ces messages contiennent l'heure d'émission extraite de l'horloge locale de la plateforme émettrice). Le temps d'échange de ces messages donne en outre une borne maximale d'erreur de la mesure de ce décalage. Ces valeurs de décalage utilisées par le middleware pour dater les objets échangés en heure locale. En effet, les données envoyées sont automatiquement datées à leur création et cette date est ajustée par les connecteurs à la réception. A intervalles réguliers les plateformes échangent les contenus de leurs DNS. Les informations ainsi reçues sont utilisées pour mettre à jour les DNS locaux.

En raison de la mobilité, les enregistrements du DNS ont une durée de vie limitée et sont supprimés à son terme. Cette durée de vie retrouve sa valeur maximale lors de chaque communication réussie et est réajustée à partir de celles des DNS reçus (conservation de la valeur maximale). Ainsi une plateforme qui disparaît ou perd toute connexion sera enlevée de tous les DNS. De même une plateforme qui ne communique avec aucune autre (ni

messages de plateformes ni connecteurs) sera supprimée, elle réapparaîtra dès qu'elle communiquera à nouveau.

5.2.5 Services pour l'application

Lorsque la plate-forme démarre elle lance les plugins définis dans un fichier de configuration. Il s'agit de services non indispensables selon le contexte (par exemple le service de routage n'est pas utile lorsque tous les hôtes sont sur le même réseau) ou de services propres à un type d'hôte (par exemple permettant l'accès à des fonctionnalités particulières comme l'appareil photo sur un Android). Dans cette dernière catégorie on trouve les services suivants :

Services d'accès à l'interface (seulement sur Android) : sur un PC les composants peuvent créer leurs propres interfaces (*JFrame*) et afficher des messages sur la console par *System.out*. Il n'en va pas de même sur les mobiles. Pour palier à ce problème, Kalimucho met en place une interface à onglets à laquelle les composants peuvent ajouter/enlever des onglets contenant leurs propres IHM et un service de gestion de ces IHM.

Services d'accès au matériel et au Web (seulement sur Android) : ces services spécifiques ne seront pas détaillés ici, il s'agit : *du service d'accès à l'appareil photo, du service d'envoi et réception de SMS et du service d'accès à Google Maps.*

5.2.6 Service de capture du contexte

Il n'existe pas de définition unique du contexte. Il dépend de l'application, en l'occurrence, il doit prendre en compte la mobilité, l'environnement, les contraintes matérielles des périphériques, etc. Puisqu'il n'y a pas de définition unique, nous nous référons à celles de (Schilit, 1994) et (Ledoux, 2005) parce qu'elles lient le contexte à la qualité de service.

Il existe trois catégories de contexte : le contexte utilisateur, le contexte d'usage et celui d'exécution. Le contexte utilisateur fait référence aux préférences de l'utilisateur. Le contexte d'usage fait référence aux contraintes de l'application, ce sont les spécifications fonctionnelles définissant ce que l'application doit et ne doit pas faire. Enfin, le contexte d'exécution fait référence au matériel (CPU, mémoire, énergie, réseau) et à l'environnement (localisation, température ...). C'est à ce dernier que nous nous intéresserons ici.

KaliSensor : Pour capturer le contexte d'exécution nous avons défini un capteur spécifique « KaliSensor » capable de supporter un fonctionnement en environnement multiprocessus. Ceci signifie que son API doit accepter la multi-notification, la multifréquence et les multi-conditions. Or la quasi-totalité des capteurs physiques ne supportent pas ce type de fonctionnement. Il faudra donc, pour avoir une interface unifiée et permettre l'utilisation des capteurs dans un environnement pervasif, que notre plateforme prenne en charge ce problème.

Dans ce but nous définissons une couche d'unification permettant de cacher l'hétérogénéité des capteurs et d'assurer la distribution des données appelée « ContextProvider ». Cacher l'hétérogénéité suppose d'unifier tant l'interface que les données.

Les capteurs « Sensors » accessibles par le service de capture du contexte sont des capteurs logiques (mesure de la charge CPU, de la batterie, de la mémoire, du débit sur le réseau) et des capteurs physiques (disponibles sur les Smartphones).

Un KaliSensor est obtenu comme un composant composite constitué d'un Sensor représentant un capteur physique ou logique et d'un ContextProvider selon le schéma suivant:

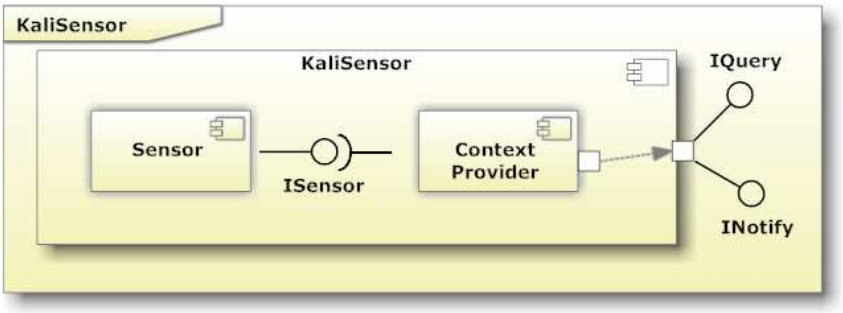

FIG. 9 - *KaliSensor*

Le **ContextProvider** est un composant logiciel lié au Sensor (capteur physique ou logique). Ceci signifie qu'à un modèle de Sensor peuvent être associées différentes implémentations de ContextProvider fournissant une interface uniforme pour son ou ses clients. Son rôle consiste à se connecter au Sensor, à en récupérer les informations et à proposer les modes d'observation en « Query » ou « Notify ».

Données de contexte de bas niveau : Le ContextProvider fournit deux interfaces « *IQuery* » et « *INotify* » correspondant aux deux modes d'observation. Ces deux interfaces utilisent, pour représenter les données capturées, le modèle de métadonnées présenté à la figure 10. Il correspond à une simple représentation des données brutes issues de capteurs logiques ou physiques.

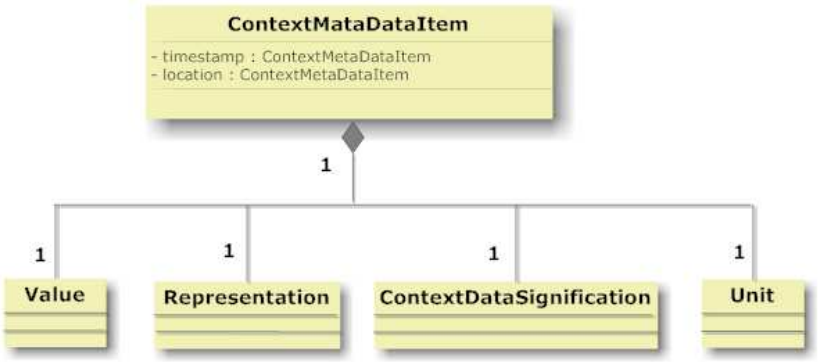

FIG. 10 - *Méta-modèle des données capturées*

« Value » est une valeur de type simple (Integer, Long, Float, String ou Boolean) mesurée par le capteur. Toutefois, ces valeurs simples n'ont aucun sens si elles ne sont pas associées à

une unité, par exemple, dire que la température est de 25 ne permet pas savoir si elle est exprimée en degrés Celsius ou Fahrenheit. C'est pourquoi à chaque « Value » est associée une unité « Unit ». « Representation » désigne le format de données, par exemple une latitude peut être représentée comme "43.4936° N" ou comme "Latitude = 43.4936°" ou encore en format « XML », « RDF », « OWL », etc. Enfin, à chaque « ContextMetaDataItem » est associé un « ContextDataSignification » donnant la sémantique de la donnée, par exemple la donnée est la latitude d'une position géographique.

Collecteur de Contexte : Les KaliSensors produisent des informations de contexte de bas niveau qui sont d'un niveau sémantique insuffisant pour pouvoir être utilisées telles quelles pour interpréter le contexte. Par ailleurs ils ne permettent ni un fonctionnement en mode multi-utilisateurs ni la gestion de modes de notification. C'est la raison pour laquelle nous avons défini un nouvel élément appelé « ContextCollector ». Pour supporter le multi-utilisateur, il faut que le ContextCollector supporte la multi-notification, la multifréquence, et les multi-conditions de notification par les clients. Nous proposons quatre types de collecteurs de contexte correspondant à quatre modes d'accès à l'information. En utilisant plusieurs de ces éléments simultanément, il est possible d'obtenir plusieurs notifications du même capteur.

Un ContextCollector ne peut être associé qu'à un seul KaliSensor mais plusieurs ContextCollectors peuvent être attachés au même KaliSensor. En effet, chaque ContextCollector gère une configuration d'acquisition, par conséquent si un consommateur a besoin de plusieurs modes d'acquisition il s'attachera un ContextCollector par mode d'acquisition. Bien entendu, un utilisateur peut s'attacher autant de ContextCollectors qu'il le souhaite.

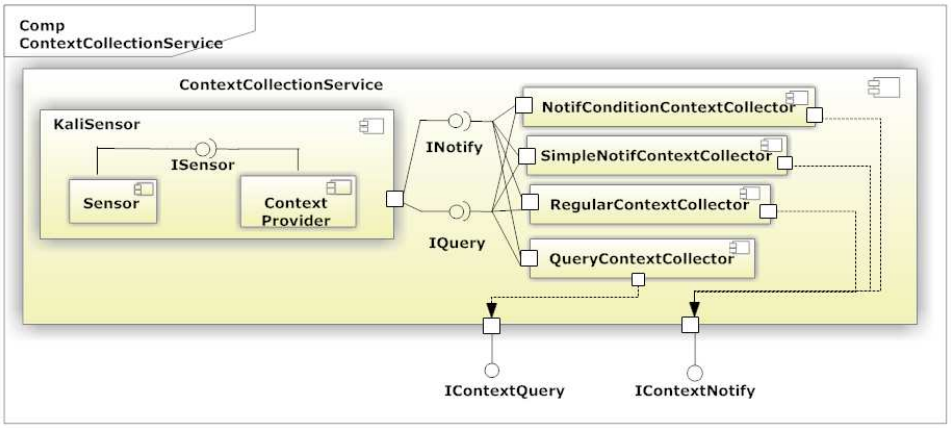

FIG. 11 - *Les quatre types de « ContextCollector »*

Le ContextCollector est un composant logiciel configurable soit en mode « Query » (lecture directe), soit en mode « SimpleNotify » (quand il y a un changement de valeur, le client sera notifié), soit en mode « RegularNotify » (le client sera notifié à une fréquence régulière), soit enfin en mode « ConditionalNotify » (le client sera notifié à partir de ses propres conditions de notification). Nous proposons un langage d'expression des conditions permettant de configurer les ContextCollector. La grammaire de ce langage est présentée ci-dessous :

NC	*= [Element Left]* ***Ops*** *[Element Right]*
	= Ops [Element Right]
	= [NC] ***Ops*** *[NC]*
	= ***Ops*** *[NC]*
Ops*: {AND, OR, NOT, >, <, ==, !=, >=, <=, =}*	
	Element *: « ContextMetaDataItem »*

Remarque : la définition des éléments de cette grammaire se fait à partir d'un *ContextMetaDataItem* dont seules les classes *Value* et *ContextDataSignification* sont utilisées (voir figure 12**FIG. 12**).

Il permet de définir des conditions complexes facilitant la notification précise d'événements de contexte. Par exemple : être notifié quand la latitude dépasse 45 degrés et que la longitude est inférieure à 11 degrés mais que l'altitude n'est pas de 20 mètres se représente de la manière suivante :

IF (« Latitude » >= « 45 ») AND (« Longitude » <= « 11 ») AND (« Altitude » != « 20 »)

Données de contexte de haut niveau : Le ContextCollector reçoit les données envoyées par un ContextProvider. Ces données sont conformes au méta-modèle présenté sur la figure 10 (*ContextMetaDataItem*). Il transforme ces informations de bas niveau en informations de haut niveau. Le méta-modèle associé complète l'information de contexte en ajoutant des informations relatives aux capteurs eux-mêmes, par exemple : l'identification et le type de capteur, le type de donnée mesurée, etc. (figure 12)

Ces informations peuvent être envoyées par réseau soit par des connecteurs pour la partie applicative soit par message d'une plateforme à une autre. Ainsi les composants métiers et les plateformes peuvent disposer d'informations contextuelles provenant d'un autre hôte.

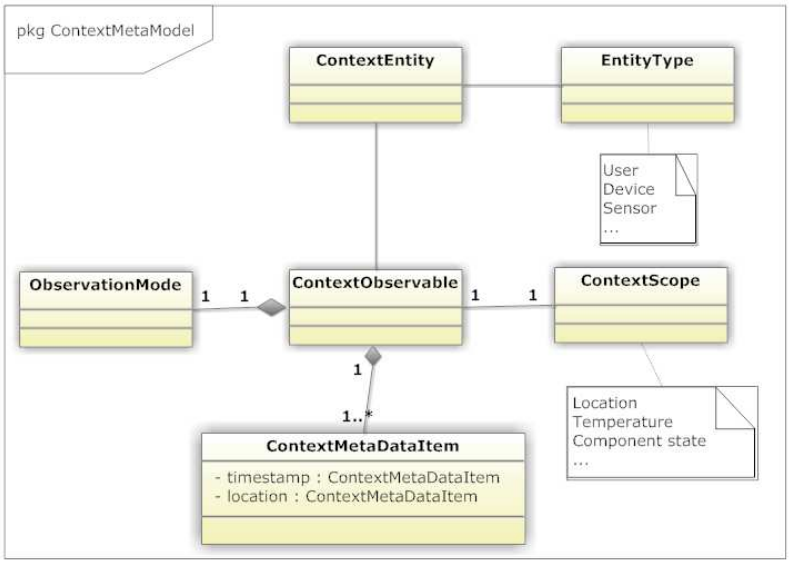

FIG. 12 - *Meta-modèle des données de contexte*

Le « ContextObservable » représente la vue logique d'un capteur. C'est un « ContextEntity », et il a un « EntityType » définissant l'origine de l'information. S'il s'agit d'un capteur physique (c'est à dire s'il est lié à un appareil), il a une « source » (l'appareil lié). Une « source » est un « ContextEntity » qui contient un « EntityType » de type « Device ».

Au « ContextObservable » est lié un « ContextScope ». Ce « ContextScope » représente le type de mesure, par exemple de type géolocalisation. Chaque « ContextScope » a ses propres champs de données, par exemple une position GPS peut ne contenir que trois champs de données : « Latitude », « Longitude » et « Altitude ». Toutefois au même capteur physique de GPS peut correspondre un KaliSensor différent dont le « ContextScope » contiendrait les trois champs précédents auxquels viendrait s'ajouter un champ « Précision ». Le « ContextScope » lié au « ContextObservable » permet d'indiquer les champs de données présents. Le « ContextObservable » peut supporter un ou plusieurs « ObservationMode » qui correspondent aux modes d'observation précédemment définis (Query ou Notify).

Enfin, un « ContextObservable » a un ou plusieurs « ContextMetaDataItem » qui décrivent les données collectées par le KaliSensor. Chaque « ContextMetaDataItem » représente l'un des champs de données du « ContextScope ». Il contient la date de collecte (timestamp) et la localisation. En effet, sans connaissance de sa localisation, une donnée peut perdre tout son sens, par exemple, une mesure de température de 28° peut ne pas être exploitable si l'on ignore où elle a été mesurée. Il contient enfin la donnée capturée représentée par : « Value », « Unit », « Representation », et « ContextDataSignification » (cf. figure 10).

API d'utilisation du service de capture de contexte : Le service de capture de contexte propose une API permettant aux composants métier des applications de récupérer des informations de contexte de haut niveau. Cette API permet de choisir un capteur physique ou logique, de lui associer un ou plusieurs collecteurs de contexte configurés selon les besoins et d'en récupérer les informations selon un mécanisme identique à celui utilisé pour les données d'entrée (voir 4.2.3) c'est à dire soit par accès direct soit par événement.

La méthode *createKaliSensor* permet de créer un KaliSensor associé à un capteur physique ou logique prédéfini. Le paramètre désigne ce capteur, par exemple "GPSSensor" pour le GPS. Elle retourne un objet implémentant l'interface *IKaliSensor* correspondant au KaliSensor créé. C'est par cette interface que les informations de contexte pourront être récupérées et interprétées. Les autres méthodes correspondent à l'association à un *KaliSensor* d'un *ContextCollector* de l'un des types définis précédemment. Elles renvoient un objet implémentant l'interface *IContextCollector* correspondant au *ContextCollector* créé. C'est par cette interface que l'on pourra accéder au mode de fonctionnement de ce collecteur de contexte.

L'interface « IKaliSensor» : Cette interface permet l'accès au contenu des informations de contextes récupérées directement ou lors d'un événement. Par exemple, lorsqu'un client veut connaître la position géographique courante à partir d'un GPS. Dans un premier temps il récupèrera les champs de données du KaliSensor associé au GPS grâce à la méthode « getObservableContextScope ». Cette méthode retourne un objet « ContextScope » contenant un nom de « Scope » (Cordonnées GPS, Température, etc.) et une liste de « ContextDataSignification » qui donnent la signification de chaque champ. Dans cet exemple, le « Scope » est [“GPS Coordinate”], et la liste contient trois éléments :

[<Name= “Latitude”, Semantic Signification= “http://www.w3.org/2003/01/geo/wgs84_pos#lat”>,

<Name= “Longitude”, Semantic Signification= “http://www.w3.org/2003/01/geo/wgs84_pos#long”>,

<Name= “Altitude”, Semantic Signification= “http://www.w3.org/2003/01/geo/wgs84_pos# alt”>]

Ensuite, il utilise la méthode « getObservableDataModel » pour obtenir un modèle de données lui permettant la vérification du type d’unité et de la présentation. Après cette vérification, il appellera la méthode « getData » pour obtenir la position géographique courante. Le résultat se présentera comme ci-dessous si l'on se trouve à Bayonne en France :

[<timestamp=< timestamp=<null>,location=<null>, value=<”1353344759386”>, unit=<“millisecond”>, presentation=<“year/month/day/time presentation”>, signif=< Name= “timestamp”, Semantic Signification= http://www.w3.org/2006/time# Instant>>, location=<null>, value=<“43.4936”>, unit=<“°”>, presentation = <“default geo point presentation”>, signif=<Name= “Latitude”, Semantic Signification= http://www.w3.org/2003/01/geo/wgs84_pos#lat>>,

<timestamp=< timestamp=<null>,location=<null>, value=<”1353344759386”>, unit=<“millisecond”>, presentation=<“year/month/day/time presentation”>, signif=< Name= “timestamp”, Semantic Signification= http://www.w3.org/2006/time# Instant>>, location=<null>, value=<“1.4750”>, unit=<“°”>, presentation = <“default geo point presentation”>, signif=< Name= “Longitude”, Semantic Signification= http://www.w3.org/2003/01/geo/wgs84_pos#long>>,

<timestamp=< timestamp=<null>,location=<null>, value=<”1353344759386”>, unit=<“millisecond”>, presentation=<“year/month/day/time presentation”>, signif=< Name= “timestamp”, Semantic Signification= http://www.w3.org/2006/time# Instant>>, location=<null>, value=<“5”>, unit=<“metre”>, presentation = <“default geo point presentation”>, signif=< Name= “Altitude”, Semantic Signification= http://www.w3.org/2003/01/geo/wgs84_pos# alt>>]

L’interface « IContextCollector»

Cette interface ne sert qu'à connaître et gérer les modes de fonctionnement des collecteurs ajoutés à un KaliSensor. Nous allons prendre comme exemple le cas de deux composants métiers utilisant la géolocalisation. Cet exemple est décrit dans figure 13. Il y a deux consommateurs (CM) ayant chacun ses propres ContextCollector. Le capteur (« Sensor ») est un module GPS d’un téléphone fonctionnant sous « Android ».

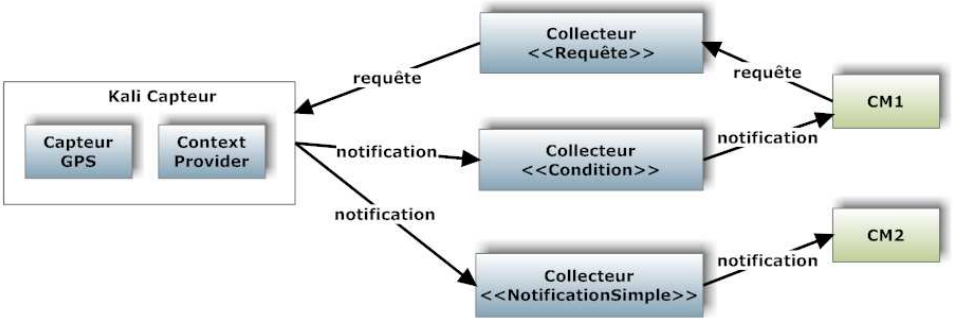

FIG. 13 - *Utilisation d'informations de contexte par des composants métier*

Le CM1 a pour objectif de faire du geofencing. Il a donc besoin d’une notification conditionnelle lorsque la position entre dans une zone donnée. Il souhaite également pouvoir fonctionner en mode « Query » pour suivre librement la position après avoir été notifié de son entrée dans la zone. *Le* CM*2* veut observer les déplacements du téléphone. Il demande donc une notification simple qui se produira à chaque modification de la position. *Le code ci-*

dessous correspond à la mise en place par le composant métier CM1 du capteur associé au GPS et des collecteurs de contexte.

```
ContextCapture cc = (ContextCapture) waitForKalimuchoService(Services.CONTEXTCAPTURE_SERVICE);
IKaliSensor iksbc1 = cc.createKaliSensor(KalimuchoSensors.GPS_SENSOR) ; // Crée un KaliSensor pour GPS

IContextCollector col1 = cc.addQueryContextCollector(iksbc1) ; // ajout d'un Context Collector de requête simple

BCEvent event = new BCEvent (ENTREE_ZONE) ; // constante entière ;

List<ContextMetaDataItem> lcmdi = iksbc1.getObservableDataModel() ; // Modèle de données associé au GPS

Iterator i = lcmdi.iterator();

String axis[] = new String [lcmdi.size()];

int index=0;

while ( i.hasNext() ) { // parcours du modèle de donnés

  // Récupération du nom des noms des champs (Longitude/Latitude/etc.).

  String axis[index] = i.next().getContextDataSignification().getName() ;

  index++;

}

// Expression condition (zone entre 40 et 45° de latitude, et 10 et 12 de longitude

String condition = new String (''IF ("+ axis[0]+">40) AND ("+axis[0] +"<45) AND ("+axis[1]+">10) AND
("+axis[1]+"<12)");

// Ajout du Contexte Collector de détection d'entrée en zone

IContextCollector col2 = cc.addNotifyConditionContextCollector(iksbc1, event, condition) ;
```

6. Mesures

Un certain nombre de mesures ont été faites sur cette plateforme, elles concernent sa taille, sa complexité, le temps d'exécution des commandes sur un Nexus One, les temps de transfert d'informations dans les connecteurs et le temps de déploiement d'une application.

6.1 Taille et complexité

Kalimucho est une plateforme qui s'apparente dans son fonctionnement à un système d'exploitation c'est pourquoi sa complexité relève davantage de la gestion des divers processus que de la complexité algorithmique. Le tableau suivant donne quelques indications sur le code, les threads et les sémaphores utilisés :

Mesures de Kalimucho	PC	Android
Taille[(1)]	512 Ko	608 Ko
Nombre de lignes de code	10557	13 709
Nombre de classes	178	262
Nombre de threads lancés au démarrage	20	21
Nombre de méthodes ou de blocs "*synchronized*"	279	244
Nombre d'opérations "*wait*" et "*notify*"	87	72

(1) la taille est celle du fichier *jar* pour les versions PC et celle indiquée par *parameters application* pour la version Android (la taille du fichier *apk* est de 225 Ko)

6.2 Temps d'exécution des commandes

L'objectif de Kalimucho étant de permettre de reconfigurer dynamiquement des applications en cours d'exécution en fonction des évolutions du contexte, il est important que les temps de reconfiguration ne soient pas rédhibitoires. La plupart des exécutions de commandes supposent des temps d'attente (réponse par réseau d'une autre plateforme, recherche de route...). Or une reconfiguration est, généralement constituée de plusieurs commandes (ajout/suppression de composants et/ou de connecteurs par exemple). C'est pourquoi ces temps d'attente sont mis à profit par la plateforme par une exécution des commandes en parallèle. Dès lors, s'il est possible de mesurer le temps d'exécution de chaque commande, ces valeurs ne sont guère significatives dans la mesure où le temps d'exécution d'une reconfiguration est inférieur à la somme des temps d'exécution de chacune des commandes prises indépendamment. Pour tenir compte de cette situation nous avons choisi de présenter des mesures correspondant à six opérations différentes de reconfiguration complètes :

- *Mesure 1* : déploiement d'un service sur une machine. Ce service est constitué de 3 composants dont les fichiers jar doivent être téléchargés et de 2 connecteurs internes.
- *Mesure 2* : extension d'un service par ajout d'une nouvelle instance d'un composant dont le fichier jar est disponible et d'un connecteur interne.
- *Mesure 3* : remplacement d'un connecteur distribué par un autre connecteur distribué
- *Mesure 4* : remplacement d'un composant par un autre composant. Ce dernier étant une instance d'un composant déjà installé son fichier jar n'a pas besoin d'être téléchargé.
- *Mesure 5* : Migration d'un composant et redirection des 3 connecteurs qui lui sont reliés. Le composant migré n'est pas connu et son fichier jar doit être téléchargé
- *Mesure 6* : Suppression d'un service constitué de 3 composants locaux et d'un composant déporté reliés par 2 connecteurs distribués.

La figure 14 montre les résultats obtenus sur un Nexus One. Les fichiers jar sont téléchargés depuis un dépôt distant :

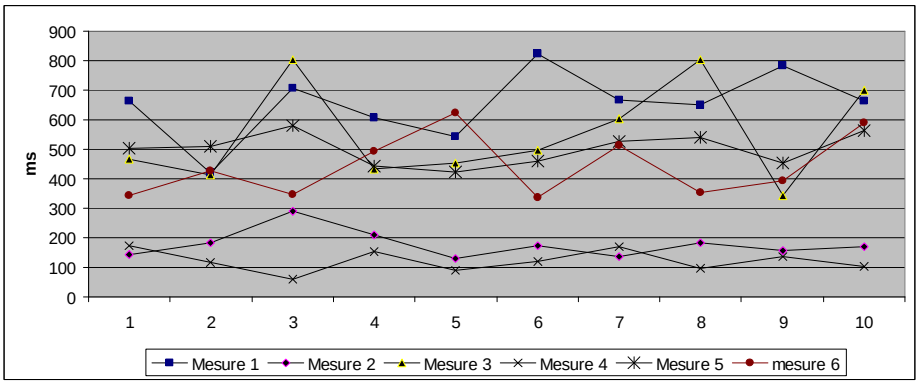

FIG. 14 - *Temps de reconfiguration sur Nexus One*

La figure 15 montre les résultats obtenus sur un PC. Le PC disposant d'un dépôt de composants le téléchargement des fichiers jar se résume à une lecture de ce fichier sur le disque local :

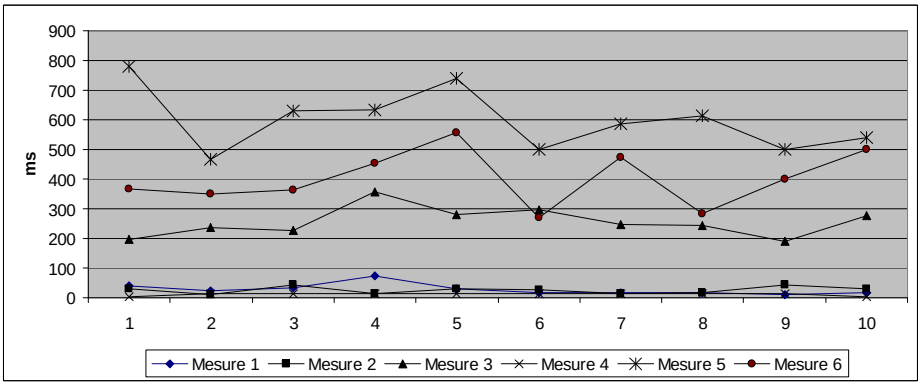

FIG. 15 - *Temps de reconfiguration sur PC*

Commentaires : Les variations visibles sur les courbes sont essentiellement liées à celles des temps d'établissement des connexions sur le réseau (ici un réseau en wifi). Le tableau suivant présente les temps moyens et les écarts types constatés :

Durées En ms	Sur *Nexus One*		Sur PC	
	Moyenne	Ecart type	Moyenne	Ecart type
Mesure 1	653	114	28	18
Mesure 2	178	46	26	12
Mesure 3	552	165	255	50
Mesure 4	121	37	12	4
Mesure 5	500	53	599	103
Mesure 6	442	107	402	93

Ainsi on peut voir, par exemple, que la reconfiguration 3 sur le *Nexus One* qui provoque l'échange de huit messages entre plateformes sur le réseau présente un écart type bien supérieur (165) à celui de la reconfiguration 2 qui n'en provoque que deux (46). Ces temps de reconfiguration restent cependant suffisamment faibles pour permettre à la plateforme d'adapter rapidement l'application à un changement de contexte. Le passage à l'échelle (nombre plus important de périphériques impliqués dans la reconfiguration) ne constitue pas un réel problème puisque chaque plateforme sur chacun des périphériques exécute ses commandes en parallèle des autres.

7. Conclusion

Nous avons présenté la plateforme Kalimucho, son architecture générale puis, de manière plus précise, l'ensemble de ses services. L'implémentation des composants logiciels et des connecteurs suit un *framework* encapsulant la logique métier et permettant à la plateforme de réaliser les opérations de supervision. Il s'agit de connaître l'état en cours des composants et connecteurs et d'agir sur leur cycle de vie lorsqu'une opération de supervision modifiant la structure de l'application doit avoir lieu. La plateforme Kalimucho permet d'exécuter des

applications (*ie. des composants formant tout ou partie d'une application*) sur différents dispositifs mobiles et d'agir à chaud sur l'architecture de l'application lorsque nécessaire. Notre objectif final est de proposer une plateforme autonomique en utilisant le service de capture du contexte dans la plateforme elle-même pour décider des reconfigurations (figure 15). Nous menons ces travaux actuellement dans le cadre de thèse de doctorat et avons basé cette prise de décision sur l'utilisation d'une ontologie du contexte permettant d'identifier des situations de reconfiguration qui seront mises en place à l'aide de chaînes de type BPM.

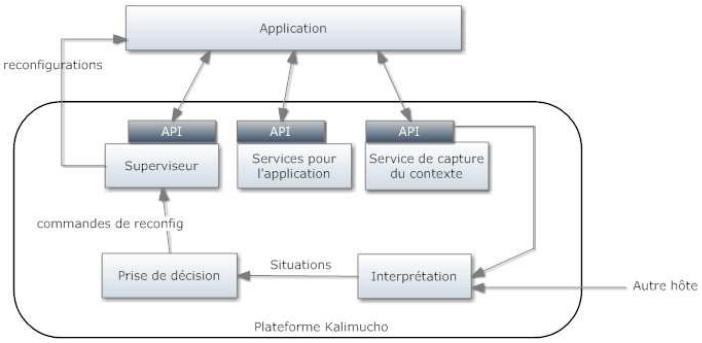

FIG. 16 - *Prise de décision*

Si l'ensemble des fonctionnalités décrites ici sont implémentées, il reste un module optionnel sur lequel nous travaillons actuellement : la *Prise de Decision*. L'objectif de ce module est d'implémenter différentes politiques d'adaptation permettant à la plateforme de décider par elle-même des reconfigurations. Il s'agit d'une étape nécessaire à l'évolution de la plateforme en vue de lui permettre d'exécuter des applications accompagnant l'utilisateur dans son quotidien, également appelées *long-life applications.*

Références

Ayed D., C. Taconet, G. Bernard, et Y. Berbers (2008). *Cadecomp : Context-aware deployment of component-based applications.* J. Network and Computer Applications, 31(3).

Bouix, E., M. Dalmau, P. Roose, et F. Luthon (2005). *A Multimedia Oriented Component Model*, AINA 2005, The IEEE 19th International Conference on Advanced Information Networking and Applications - Tamkang University, Taiwan, March 28 - March 30.

David, P., C., et T. Ledoux (2005). *Wildcat: a generic framework for context-aware applications*. In Sotirios Terzis and Didier Donsez, editors, MPAC, volume 115 of ACM International Conference Proceeding Series, pages 1–7. ACM.

Dorn, C., et R. N. Taylor (2012). *Co-adapting human collaborations and software architectures*, Proc. of the 2012 International Conference on Software Engineering, ICSE 2012, pp. 1277-1280

Floch J., SS. O. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, et E. Gjrven (2006). *Using architecture models for runtime adaptability*. IEEE Software, 23(2) :62-70.

Gabillon, Y. , G. Calvary, et H. Fiorino (2011). *Composition d'Interfaces Homme-Machine en contexte: approche par planification automatique* – Revue TSI. Vol. 30.

Gui, N., V. De Florio, H.Sun, C. Blondia (2011). *Toward architecture-based context-aware deployment and adaptation,* The Journal of Systems and Software 84 (2011) 185–197 – Elsevier.

Heineman, G. T., et W. T. Council (2001). *Component-based software engineering, putting the pieces together*. Addison-Wesley.

Hourdin V., J.Y. Tigli, S. Lavirotte, G. Rey, and M. Riveill (2008). *Slca, composite services for ubiquitous computing.* In Jason Yi-Bing Lin, Han-Chieh Chao, and Peter Han Joo Chong, editors, Mobility Conference, page 11. ACM.

Jeronimo, M., et J. Weast (2003). *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press.

Laplace, S., M. Dalmau, et P. Roose (2007). *Kalinahia : modèle de qualité de service pour les applications multimédia reconfigurables*, Numéro Spécial Revue ISI "Conception : patrons et spécifications formelles", Vol. 12 N°4/2007, ISBN: 978-2-7462-1968-7.

Maurin, V., N. Dalmasso, B. Copigneaux, S. Lavirotte, G. Rey, and J. Y. Tigli (2009). *Simply engine-wcomp : plate-forme de prototypage rapide pour l'informatique ambiante basée sur une approche orientée services pour dispositifs réels/virtuels*. In David Menga and Florence Sedes, editors, UbiMob, volume 394 of ACM International Conference Proceeding Series, pages 83-86. ACM.

Meyer B. (2003). *The grand challenge of trusted components*. In Software Engineering, International Conference on Software Engineering, page 660.

Oreizy, P., N. Medvidovic, et R.N. Taylor (2008). *Runtime Software Adaptation: Framework, Approaches, and Styles*, ICSE Companion '08 Companion of the 30th international conference on Software engineering, pp. 899-910.

OSGi, http://www.osgi.org/Main/HomePage

Rouvoy R., P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A.Mamelli, et U. Scholz (2009). *MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments*, Book on Software Engineering for Self-Adaptive Systems . LNCS 5525.

Schilit, B., N., et M. Theimer (1994). *Disseminating active map information to mobile hosts*. IEEE Network, 8(5): 22–32.

Shlimmer, J., et J. Thelin (2006). *Device profile for web services. Technical* report, http://schemas.xmlsoap.org/ws/2006/02/devprof/.

Sousa, J. P., et D. Garlan (2002). *Aura: an architectural framework for user mobility in ubiquitous computing environments.* In Jan Bosch, W. Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, WICSA, volume 224 of IFIP Conference Proceedings, pages 29-43. Kluwer.

Szyperski, C. (2001). *Component software - Beyond object-oriented programming*. Addison-Wesley, 1998.

Summary

The development of ubiquitous applications is particularly complex. Beyond the dynamic aspect of such applications, the evolution of the computer science to the proliferation of terminals with mobile access does not make things easier. A solution to simplify the development and exploitation of such applications is to use software platforms dedicated to the deployment and adaptation of applications and the management of devices heterogeneity. They allow designers to focus on the business aspects and facilitate the reuse. It is from this perspective that the platform Kalimucho was designed and developed. It allows the execution and the monitoring of component-based applications and provides to the applications an access to their execution context.