KELING DA

KALIMUCHO-A : AUTONOMIC KNOWLEDGE-BASED CONTEXT-DRIVEN ADAPTATION PLATFORM

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

Spécialité : Informatique

Titre de la thèse :

KALIMUCHO-A : AUTONOMIC KNOWLEDGE-BASED CONTEXT-DRIVEN ADAPTATION PLATFORM

KELING DA

Thèse dirigée par Marc DALMAU

et codirigée par Philippe ROOSE

Thèse soutenue publiquement le 16 octobre 2014 devant le jury composé de :

M.	Philippe	LALANDA	Rapporteur
Mme	Chantal	TACONET	Rapporteur
M.	Franck	BARBIER	Examinateur
M.	Khalil	DRIRA	Examinateur
Mme	Manuele	KIRSCH PINHEIRO	Examinateur
M.	Thierry	NODENOT	Examinateur
M.	Romain	ROUVOY	Examinateur
M.	Marc	DALMAU	Directeur de thèse

天行健, 君子当自强不息; 地势坤, 君子以厚德载物.

– 易经

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 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.

La gestion du contexte est un élément clé lorsque l'on souhaite réaliser des applications pervasives sensibles au contexte. Les informations contextuelles issues d'un grand nombre de sources distribuées différentes sont, généralement, des informations brutes qui, sans interprétation, peuvent être dénuées de sens. En se basant sur des ontologies, il est possible de construire des modèles sémantiques qui seront alimentés par ces informations brutes et ainsi non seulement d'augmenter leur niveau de représentation sémantique mais surtout de pouvoir les utiliser pour prendre des décisions automatiques d'adaptation d'applications basées sur le contexte au runtime.

La démocratisation des périphériques conduit à ce qu'un usager dispose actuellement de plusieurs périphériques incluant postes fixes, téléphones, tablettes, box, etc. pour son usage personnel. Il est souhaitable que cet ensemble de ressources lui soit accessible en tout point et à tout moment. De même des ressources publiques (stockage, services, etc.) peuvent lui être offertes. En revanche, la protection de la vie privée et les risques d'intrusion ne peuvent être négligés. Notre proposition est de définir, pour chaque utilisateur, un domaine d'adaptation qui contient l'ensemble des ressources auxquelles il peut accéder sans limite. Ces ressources sont celles qu'il a accepté de rendre disponibles sur ses machines pour lui-même et celles que les autres utilisateurs ont accepté de partager. Ainsi la notion de contexte est liée à celle d'utilisateur et inclut la totalité des ressources auxquelles il a accès. C'est la totalité de ces ressources qui sera exploitée pour faire en sorte de lui offrir les services adaptés à ses choix, ses dispositifs, sa localisation, etc.

Nous proposons un middleware de gestion de contexte Kali2Much afin de fournir des services dédiés à la gestion du contexte distribué sur le domaine. Ce middleware est accompagné du module Kali-Reason permettant la construction de chaînes de raisonnement en BPMN afin d'offrir des fonctionnalités de raisonnent sur les informations de contexte dans le but d'identifier des situations nécessitant éventuellement une reconfiguration soit de l'application soit de la plateforme elle-même. C'est ainsi qu'est introduit l'aspect autonomique lié à la prise de décision. Les situations ainsi détectées permettent d'identifier le moment où déclencher les adaptations ainsi que les services d'adaptation qu'il sera nécessaire de déclencher. La conséquence étant d'assurer la continuité de service et d'ainsi s'adapter en permanence au contexte du moment. Le travail de reconfiguration d'applications est confié au service Kali-Adapt dont le rôle est de mettre en œuvre les adaptations par déploiement/redéploiement de services de l'application et/ou de la plateforme.

Un prototype fonctionnel basé sur la plateforme Kalimucho-A vient valider ces propositions.

ABSTRACT

The ubiquitous applications development is not a trivial task. Beyond the dynamic aspect of such applications, the evolution of computer science toward the proliferation of mobile devices does not make things easier. A solution to simplify the development and operation of such applications is to use software platforms dedicated to deployment and adaptation of applications and managing heterogeneous devices. Such platforms allow designers to focus on business issues and facilitate reuse.

Context management is a key element for making context-aware pervasive applications. Contextual information comes from many different distributed sources. It is generally raw information with no interpretation. It may be meaningless. Based on ontologies, it is possible to construct semantic models that would be powered by the raw information. This does not only increase the level of semantic representation but it can also be used to make automatic decisions for adapting context-based applications at runtime.

Devices' democratization allows a user to have multiple devices including personal computer, mobile phones, tablets, box, etc. for his personal use. It is desirable that the set of resources will be available to him from everywhere and at any time. Similarly, public resources (storage, services, etc.) would also be accessible to him. However, protection of privacy and intrusion risks cannot be ignored. Our proposal is to define, for each user, an adaptation domain that contains all his resources. Users can access their resources without limits. Users can agree on sharing resources with other users. Thus the notion of context is related to the user and includes all the resources he can access. All these resources will be exploited to offer him services adapted to his preferences, his features, his location, etc.

We propose a context management middleware Kali2Much to provide services dedicated to the management of distributed context on the domain. This middleware is accompanied by Kali-Reason module for building reasoning chains in BPMN. The reasoning chains provide context information reasoning functionality. They reason about context information in order to identify situations that might require a reconfiguration of the application or of the platform itself. Thus the autonomic aspect related to decision making is introduced. Situations detected allow to identify when there is a need to trigger adaptation. The consequence is to ensure continuity of service and thus constantly adapt to the current context. The reconfiguration applications work is dedicated to Kali-Adapt service whose role is to implement the adaptations deployment/redeployment of application services and/or platform.

A working prototype based on Kalimucho-A platform validates the proposals.

REMERCIEMENTS

Je tiens à remercier :

Madame *Chantal TACONET*, Maître de Conférences (HDR) à Télécom SudParis, et Monsieur *Philippe LALANDA*, Professeur à l'Université Joseph Fourier, qui m'ont fait l'honneur de lire, analyser mon manuscrit et de rapporter sur mon travail. Leurs remarques m'ont permis d'en améliorer certains aspects. Je les en remercie sincèrement.

Je remercie Madame *Manuele KIRSH PENEIRO*, Maître de Conférences à l'Université de Paris I, pour ses conseils et ses encouragements lorsque j'étais encore devant des pages bien blanches.

Merci également aux autres membres du jury qui ont accepté de juger ce travail : Monsieur *Khalil DRIRA*, Directeur de Recherche au LAAS – Toulouse, Monsieur *Romain ROUVOY*, Maître de Conférences à l'Université de Lille I, Monsieur *Thierry NODENOT* et Monsieur *Franck BARBIER*, Professeurs à l'Université de Pau et des Pays de l'Adour.

Mes plus sincères remerciements à Monsieur *Marc DALMAU* et Monsieur *Philippe ROOSE*, Maîtres de Conférences (HDR) à l'Université de Pau et des Pays de l'Adour, mes directeurs de thèse, pour leur encadrement et leur soutien. La confiance que vous m'accordée m'a permis de mener à bon bien cette thèse. Vous m'avez donné suffisamment de liberté pour choisir la direction de cette thèse. Je vous remercie également pour votre soutiens et votre patience pendant les périodes difficiles.

Je remercie également tous ceux qui m'ont soutenu et encouragé tout au long de ces quatre années de thèse.

Mon collègue d'équipe *Joseba*, merci pour ton soutien et ton aide. Et mes autres collègues d'équipe : *Ghada, Eliana, Régina, Khouloud, Salomon et Dia*.

Mes amis *Julienne, Dimmitry, Eric, Gauthier, Céline* que je n'ai pas mentionné ci-dessus mais dont la compagnie et le soutien n'ont pas été des moindres.

Je remercie mes parents et ma famille, qui ont toujours cru en moi et qui m'ont toujours soutenu. Merci maman et papa, sans vous je ne serais jamais parti dans cette aventure ! Malgré les 10000 km qui nous ont séparés pendant les 11 dernières années vous avez su être toujours près de moi pour m'encourager dans les moments difficiles et pour fêter mes réussites.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

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 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, etc. 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 pour une durée suffisante.

Pour que les applications puissent être adaptées à l'environnement pendant qu'elles sont en cours d'utilisation et sans devoir les arrêter elles doivent adopter une architecture permettant cette dynamique. C'est à dire une architecture modifiable durant l'exécution. L'approche par composants et connecteurs adoptée par la plateforme Kalimucho se prête tout particulièrement à ce mode de fonctionnement. Réaliser des applications reconfigurables en fonction du contexte suppose de mettre en place les étapes suivantes : Acquisition du contexte, modélisation du contexte, interprétation du contexte, identification des besoins d'adaptation, prise de décisions d'adaptation et mise en oeuvre de ces décisions. Ce sont ces opérations qui forment le coeur des travaux présentés dans ce mémoire.

Dans la littérature, il existe des plateformes visant un objectif similaire comme, par exemple, MUSIC [51], CAMPUS [143], COSMOS [44], CAPPUCINO [108], etc. Toutefois aucune ne se préoccupe de la continuité de service de la plateforme elle-même. Or, dans un environnement mouvant, les services de la plateforme peuvent eux-mêmes être perdus ou ne plus pouvoir fonctionner correctement, par exemple, en raison de la disparition de certaines ressources. Nous avons abordé ce problème en faisant en sorte que les services de l'application et ceux de la plateforme soient traités de la même façon et puissent faire l'objet de reconfigurations contextuelles. Par ailleurs, si la capture et l'analyse du contexte sont indispensables au fonctionnement de la plateforme, elles doivent également être accessibles à l'application elle-même car

seule la logique métier de l'application peut traiter certains types de contexte. Nous proposons dans ce but un modèle de contexte extensible en cours d'exécution et partagé entre la plateforme et les applications. Ce modèle, associé à des raisonnements, permet de transformer les informations contextuelles de bas niveau en informations de plus haut niveau plus facilement interprétables.

Notre objectif est de définir, pour chaque utilisateur, un domaine d'adaptation qui contient l'ensemble des ressources auxquelles il peut accéder sans limite. Ces ressources sont celles qu'il a accepté de rendre disponibles sur ses machines pour lui-même et celles que les autres utilisateurs ont accepté de partager. Ainsi la notion de contexte est liée à celle d'utilisateur et inclut la totalité des ressources auxquelles il a accès. C'est la totalité de ces ressources qui sera exploitée pour faire en sorte de lui offrir les services adaptés à ses choix, ses dispositifs, sa localisation, etc.

Nous proposons une plateforme logicielle (Kalimucho-Adaptation (Kalimucho-A) Platform) 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.) afin de mettre en place une politique d'adaptation basée sur la modification de l'architecture (son déploiement) de l'application.

Dans cette plateforme, nous proposons un middleware de gestion de contexte Kali2Much afin de fournir des services dédiés à la gestion du contexte distribué sur le domaine. Ce middleware est accompagné du module Kali-Reason permettant la construction de chaînes de raisonnement en BPMN afin d'offrir des fonctionnalités de raisonnent sur les informations de contexte dans le but d'identifier des situations nécessitant éventuellement une reconfiguration soit de l'application soit de la plateforme elle-même. C'est ainsi qu'est introduit l'aspect autonomique lié à la prise de décision. Les situations ainsi détectées permettent d'identifier le moment où déclencher les adaptations ainsi que les services d'adaptation qu'il sera nécessaire de déclencher. La conséquence étant d'assurer la continuité de service et d'ainsi s'adapter en permanence au contexte du moment. Le travail de reconfiguration d'applications est confié au service Kali-Adapt dont le rôle est de mettre en œuvre les adaptations par déploiement/redéploiement de services de l'application et/ou de la plateforme.

De telles reconfigurations ne peuvent être réalisées sans avoir une vue globale de l'application, seul moyen de disposer d'une cartographie complète des périphériques accessibles et des ressources (batterie, CPU, composants disponibles, capteurs, etc.). Néanmoins, s'il est nécessaire d'avoir cette vue globale, cela ne signifie pas que le choix d'une solution centralisée soit judicieux, en effet, cela ferait de ce nœud central un point critique.

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 prend la forme d'un ensemble de composants logiciels reliés par des connecteurs. Ces fonctionnalités peuvent être réalisées de différentes façons à partir de différent assemblages de composants. 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 [125].

Pour pouvoir s'adapter dynamiquement, l'application doit avoir une connaissance d'elle-même (réflexivité) afin de pouvoir remplacer un service défectueux ou inadapté au contexte par un autre [80]. Dans ce but nous avons choisi d'utiliser une plateforme d'exécution qui connaît l'application en cours d'exécution et son contexte. Cette plateforme doit être distribuée sur l'ensemble des dispositifs impliqués.

La plateforme Kalimucho-A que nous proposons peut alors prendre des décisions de reconfigurations dynamiques en toute connaissance de cause. Elle peut ainsi 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.

1.1 APERÇU DE LA PROPOSITION

Cette partie présente un aperçu de notre proposition. Nous nous intéressons d'abord au contexte du travail : la plateforme Kalimucho qui est la base de notre plateforme d'adaptation. Ensuite, nous abordons l'une des notions fondamentales du fonctionnement de Kalimucho-A : celle de domaine d'adaptation. Pour finir, nous présentons l'architecture et les flux de données de Kalimucho-A.

1.1.1 *Contexte du travail*

La plateforme Kalimucho-A est basé sur la plateforme Kalimucho. La plateforme Kalimucho est un intèrgiciel capable de mettre en oeuvre des reconfigurations en cours d'exécution. Kalimucho implémente un modèle de composants appelé Osagaia, possédant un cycle de vie supervisable par la plateforme et permettant la migration à chaud. 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.

Kalimucho, 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. Kalimucho-A utilise la plateforme Kalimucho pour collecter le contexte lié aux états des composants et des connecteurs, pour gérer les communications par réseau ainsi que pour déployer, supprimer et migrer des composants au runtime.

1.1.1.1 *OSAGAIA : conteneur de composants métier*

Le conteneur Osagaia possède trois types d'unités fonctionnelles : l'Unité d'entrée (UE), l'Unité de sortie (US) et l'Unité de contrôle (UC). Les unités d'entrée et de sortie peuvent être connectées à un ou plusieurs connecteurs simultanément. Elles permettent au composant métier (CM) qui contient la logique métier de l'application 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.

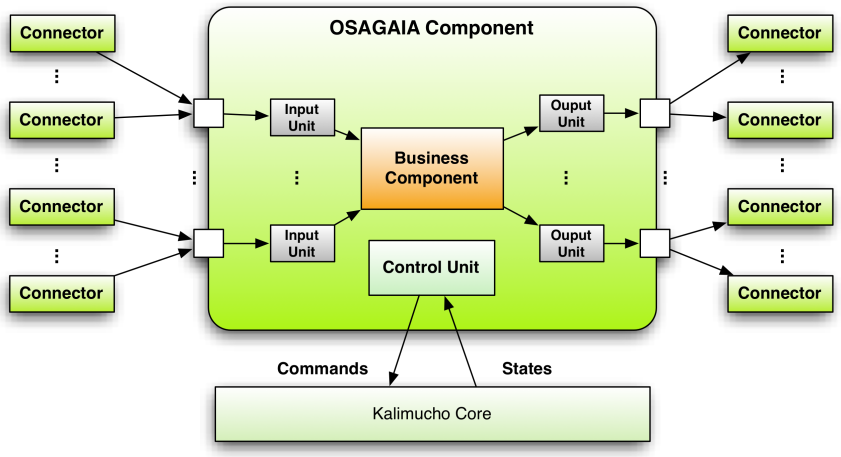

FIGURE 1 – Modèle de conteneur de composant métier OSAGAIA

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.

1.1.1.2 *KORRONTEA : conteneur de connecteur*

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.

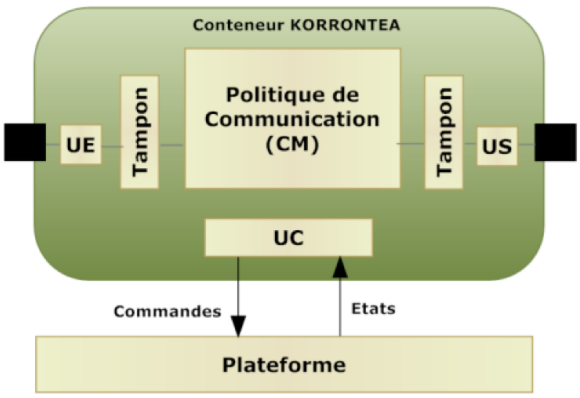

FIGURE 2 – Modèle de Connecteur Korrontea

Le connecteur Korrontea informe la plateforme de la circulation des données dans l'application. Il lève des alarmes quand des données s'accumulent dans ses tampon mais également quand la circulation des données devient fluide après une accumulation. Ceci permet à la plateforme de surveiller la circulation de données dans un hôte ou entre deux hôtes sur le réseau.

Les connecteurs fonctionnent en mode synchrone : pour chaque donnée envoyée un acquittement est renvoyé. Aucune nouvelle donnée ne peut être envoyée avant que l'acquittement n'ait été reçu. Ce mécanisme est nécessaire parce que la plateforme Kalimucho doit contrôler et mesurer la circulation des données. Ce qui signifie qu'aucune donnée ne peut être accumulée hors de son middleware (par exemple dans les buffers des sockets).

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, bluetooth, 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 ou plusieurs hôtes pouvant servir de passerelle entre ces deux machines.

1.1.2 *Notion de domaine*

L'adaptation des applications repose sur le partage de ressources. A priori, la plateforme peut déployer des composants sur toute machine supportant Kalimucho. Toutefois, dans le cadre d'applications liées à des utilisateurs, il est clair que nous devons tenir compte de la dimension privée des dispositifs. Un utilisateur n'est, a priori, pas disposé à partager ses ressources avec la totalité des utilisateurs de la plateforme. C'est pourquoi nous devons faire en sorte de concilier les besoins de l'adaptation aux exigences des utilisateurs. Dans ce but nous avons choisi de laisser chaque utilisateur libre de désigner les ressources qu'il accepte de partager et celles qu'il veut garder pour son usage exclusif.

La démocratisation des périphériques conduit à ce qu'un usager dispose actuellement de plusieurs périphériques incluant postes fixes, téléphones, tablettes, box, etc. Pour son usage personnel il est souhaitable que cet ensemble de ressources lui soit accessible en tout point et à tout moment. De même des ressources publiques (stockage, services, etc.) peuvent lui être offertes. En revanche, la protection de la vie privée et les risques d'intrusion ne peuvent être négligés.

Notre proposition est de définir, pour chaque utilisateur, un domaine d'adaptation qui contient l'ensemble des ressources auxquelles il peut accéder sans limite. Ces ressources sont celles qu'il a accepté de rendre disponibles sur ses machines pour lui-même et celles que les autres utilisateurs ont accepté de partager.

La plateforme Kalimucho intègre un mécanisme de découverte du voisinage. Quand elle démarre, elle envoie un message en broadcast pour signaler sa présence sur le réseau local et un message en point à point à toutes les machines dont elle connaît, a priori, l'identité. Ce mécanisme a été mis en place pour permettre les fonctions de routage de la plateforme qui fonctionne en mode pair à pair et doit donc pouvoir établir des connexions entre toutes les machines présentes sur le réseau. Nous utilisons ce mécanisme pour

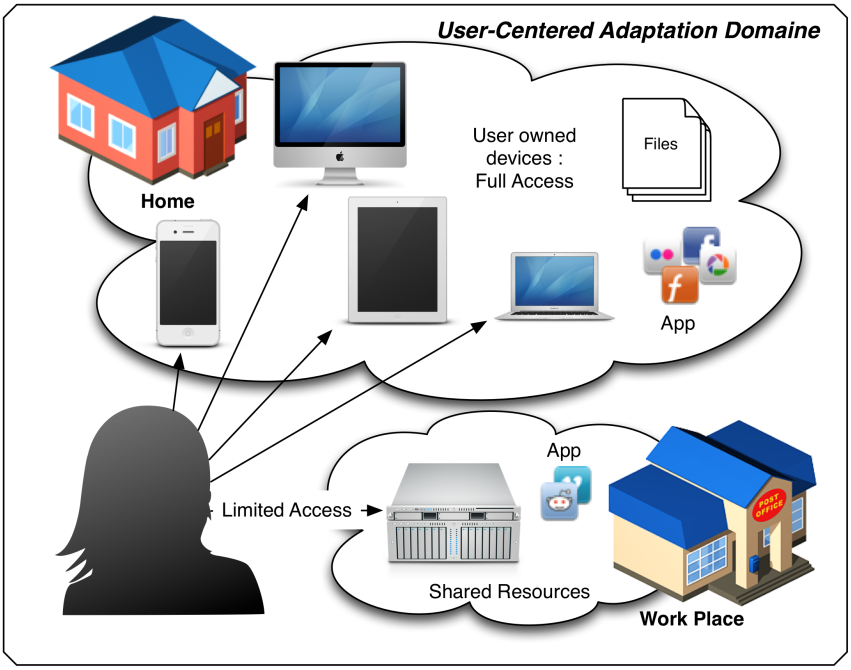

FIGURE 3 – Domaine d'adaptation

diffuser les informations sur les ressources que la machine met sur le domaine privé de son propriétaire et celles qu'elle partage.

Le fonctionnement que nous souhaitons mettre en place substitue à la notion habituelle de machine support d'application celle de domaine support d'application. Notre objectif est de faire en sorte qu'un utilisateur voit ses applications fonctionner sur l'ensemble des ses dispositifs et dispose de l'ensemble des ressources de ces dispositif plus celles partagées par les autres utilisateurs. Partant de ce principe, la plateforme peut librement disposer de la totalité du domaine d'adaptation de chacun des utilisateurs pour reconfigurer ses applications.

Cette notion de domaine d'adaptation sera présente dans toute notre approche de l'adaptation. Ainsi la prise de décision conduisant aux adaptations des applications se fera au cœur de chacun des domaines d'adaptation donc pour chacun des utilisateurs. Ceci signifie que le contexte pris en compte pour la décision est celui du domaine d'adaptation, que les ressources utilisables pour mettre en œuvre les reconfigurations sont celles du domaine d'adaptation et que la décision est prise par l'une des machines du domaine d'adaptation que nous appellerons le Domain Manager (DM) mais sera exécutée par toutes.

Il est clair que cette approche est centrée sur les utilisateurs et sur la qualité et la continuité de service. Notre but est de viser la satisfaction des utilisateurs. Nous sommes bien conscients que nous ne pouvons pas prétendre atteindre ainsi la meilleure utilisation des ressources pour une application répartie mais seulement la satisfaction des utilisateurs en fonction des ressources constituant leur domaine d'adaptation.

1.1.3 *Workflow de la plateforme : Kalimucho-A*

Kalimucho-A est basée sur la plateforme Kalimucho qu'elle étend. Elle l'utilise comme support de gestion de composants logiciels dynamiquement supervisables (déploiement, suppression, migration à chaud) et comme middleware de communication.

L'architecture de la plateforme Kalimucho-A contient plusieurs parties indépendantes (voir figure 4) qui exploitent des connaissances sémantiques (stockées dans des ontologies).

Kali2Much est notre middleware de contexte. Il contient quatre entités : "KaliSensor", "Context Collector", "Context Transformer" et "Context Searching Service".

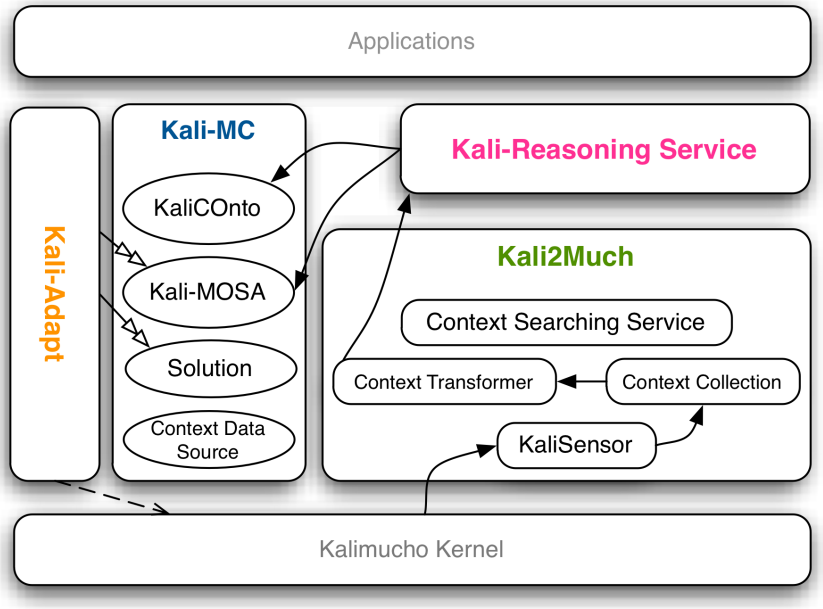

FIGURE 4 – Architecture de la plateforme : Kalimucho-A

Les données contextuelles brutes sont collectées par les KaliSensors. Selon les besoins des consommateurs de ces données, la plateforme peut configurer les Context Collectors pour leur permettre d'accéder à ces informations par consultation ou par notification. Des Context Transformers peuvent être mis en place pour modifier les données avant qu'elles soient délivrées aux consommateurs. Ces dispositifs de collecte d'informations contextuelles peuvent être déployés sur chaque hôte du domaine d'adaptation.

Le service "Context Searching Service" permet aux consommateurs de rechercher, dans le domaine, des sources d'informations contextuelles de bas niveau à partir de descriptions sémantiques. Ce service est déployé en tant qu'instance unique pour chaque domaine d'adaptation.

La plateforme Kalimucho-A utilise donc Kali2Much pour collecter les informations de contextes de bas niveau qu'elle transmet au Kali-Reasoning Service pour les interpréter et produire des informations de contexte de plus haut niveau sémantique (contexte sémantique). Ces informations de contexte de haut niveau produisent par la suite des situations d'adaptation.

Toutes ces informations sont stockées dans des modèles sémantiques basés sur des ontologies. Nous proposons pour cela un modèle de contexte (KaliCOnto), un modèle de situation (KaliMOSA), une ontologie pour la recherche de contexte et une ontologie de solution pour la recherche de solutions d'adaptation.

Dès qu'une situation d'adaptation est identifiée, le service Kali-Adapt reçoit une notification qui déclenche la recherche d'une solution applicable dans le contexte actuel. Pour finir la reconfiguration est réalisée par l'envoi de commandes aux différentes plateformes Kalimucho hébergées sur les hôtes concernés, provoquant ainsi les modifications de l'architecture de l'application ou de la plateforme elle-même.

Le schéma suivant (figure.5 - **à voir en couleur**) présente les différentes étapes nécessaires permettant à Kalimucho-A de fournir la fonctionnalité 'autonomique' permettant les reconfigurations nécessaires pour assurer la continuité de services.

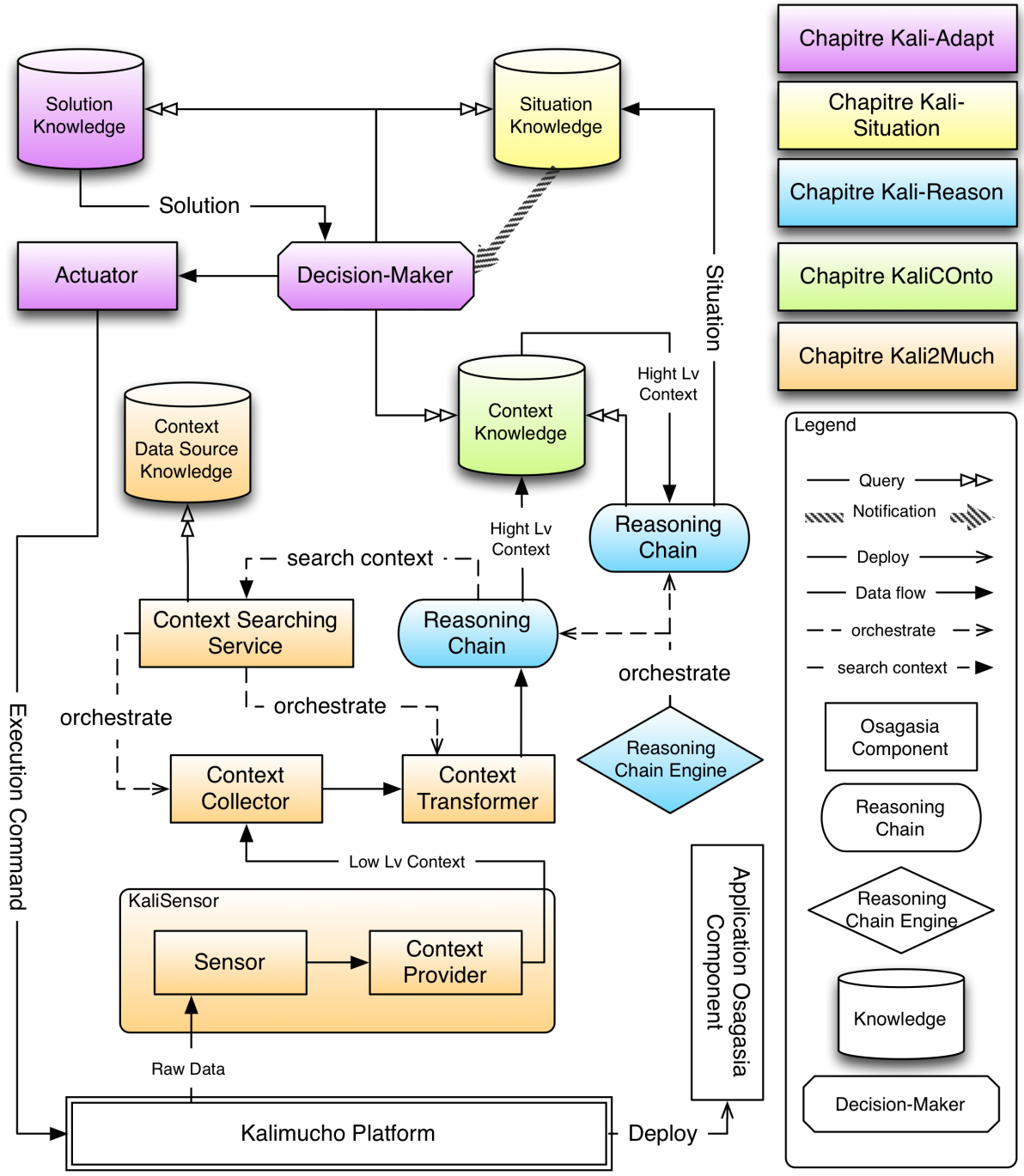

FIGURE 5 – Workflow de la plateforme : Kalimucho-A

1.2 ORGANIZATION DE LA THÈSE

Ce mémoire est organisé en trois parties :

— Partie I : Etat de l'art constitué de sept chapitres ;

— Partie II : Contribution constitué de cinq chapitres ;

— Partie III : Conclusion et Perspectives.

L'objectif de la Partie I est de proposer un état de l'art sur les plateformes d'adaptation dans un cadre applicatif lié à l'informatique ubiquitaire. Nous commençons par une étude des systèmes d'adaptation pour dégager les besoins de ces plateformes. Ensuite nous présenterons les technologies existantes pour les systèmes et les plateformes d'adaptation en terminant par les modèles d'application et l'ingénierie logicielle. Nous conclurons cette partie par une classification des approches étudiées.

La Partie II a pour permet de présenter les contributions principales de la thèse. Elle est organisée en cinq chapitres : "Ontologie de contexte de Kalimucho (KALICONTO)", "Middleware de gestion du contexte (KALI2MUCH)", "Raisonnement sur le contexte (KALIREASON)", "KALI-SITUATION : Identifier les besoins d'adaptation", et "Service d'adaptation (KALI-ADAPT)" qui correspondent aux cinq étapes de l'adaptation : Modélisation du contexte, Acquisition du contexte, Interprétation du contexte, Identification des besoins d'adaptation et Prise de décision et mise en œuvre de l'adaptation.

KaliCOnto est notre modèle de contexte basé sur une ontologie. L'objectif principal est de construire un modèle de contexte pour supporter les fonctionnalités de notre plateforme d'adaptation sensible au contexte. Il est utilisé par les applications et par la plateforme.

Kali2Much est notre middleware de gestion du contexte. Son rôle essentiel est de permettre la récupération des informations contextuelles de tous types.

KALI-REASON est notre architecture d'interprétation du contexte qui est réalisé par un moteur d'orchestration (Reasoning Chaine Engine) et des chaînes de raisonnement basées sur le langage BPMN 2.0.

KaliMOSA est notre modèle de situation d'adaptation basé sur une ontologie. Il permet la détection par le système des moments où doivent se faire des adaptations.

Enfin KALI-ADAPT est notre service de la prise de décision d'adaptation. Il fonctionne autour de l'ontologie de solutions d'adaptation liées à des actions mises en œuvre par des actionneurs.

Nous conclurons la thèse dans la Partie III en rappelant les points principaux de notre contribution et en proposant les perspectives envisagées pour de futurs travaux.

Première partie

ÉTAT DE L'ART

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

M. Weiser

2

INTRODUCTION

Le terme "informatique ubiquitaire" a été proposé initialement par Mark Weiser en 1991. Dans les environnements ubiquitaires, les ordinateurs peuvent mesurer leur environnement physique et partager des ressources. C'est une approche nouvelle d'afin d'augmentater leurs capacités et dont l'un des défis majeurs est de masquer la complexité de ce partage afin de rendre la technologie invisible aux utilisateurs finaux [144].

L'informatique ubiquitaire doit donc être omniprésente, facile à utiliser et adaptable. Les applications futures se heurtent à un environnement matériel et logiciel hétérogène et dynamique face auquel elles doivent être capables de s'adapter et de réagir dynamiquement. Elles doivent également être capables de s'adapter dynamiquement au contexte utilisateur et environnemental [109].

Le nombre de périphériques personnels mobiles et connectés (smartphones, tablettes, etc.), de périphériques connectés dans la maison (Smart-TV, Set-top-Box, etc.) et de petits périphériques communicants (capteurs) est en accroissement constant depuis plusieurs années. Ces périphériques disposent actuellement de ressources de calcul importantes et de capacités de communication (3G/4G, WiFi, etc.). Nous nous situons maintenant au début de l'ère "une personne, plusieurs ordinateurs" souvent désignée par le terme de 3ème vague. Toutefois, au niveau du logiciel, ces "smart-devices" restent encore individuels et leurs ressources ne bénéficient pas d'une gestion efficace de haut niveau. Pour que l'informatique ubiquitaire prenne toute sa place, les futurs logiciels doivent pouvoir s'adapter en temps réel tant à l'environnement qu'aux besoins des utilisateurs.

Les systèmes d'adaptation constituent une solution pour adapter le logiciel à l'environnement et ainsi satisfaire aux contraintes de l'informatique ubiquitaire. L'adaptation, et plus particulièrement l'adaptation en temps réel, soulève des problèmes scientifiques complexes ainsi que de nouveaux défis pour l'exécution et le développement du logiciel. Par exemple, comment collecter les informations de contexte dans un environnement distribué et très dynamique ? Comment adapter l'application pour obtenir de bonnes performances dans un environnement mouvant ? Comment les applications peuvent-elles exploiter des infrastructures hétérogènes tout en tirant profit des possibilités offertes par ces environnements ? Un système d'adaptation pour l'informatique ubiquitaire doit être capable de fournir des solutions appropriées à ces questions. Il doit assurer quatre rôles essentiels : Le rôle de Gestionnaire de Contexte, celui de Middleware, celui de Planificateur et celui de la prise de Décision.

Chacun de ces rôles soulève des défis différents qui seront développés dans la section suivante. Le système d'adaptation agit à deux niveaux : celui de l'adaptation de l'application et celui de l'adaptation du système lui-même. Le premier niveau n'agit qu'au niveau de l'application elle-même. Par exemple, les

systèmes basés sur des webservices peuvent agir sur l'application, mais si un périphérique mobile perd la connexion réseau, il perd aussi ses possibilités d'adaptation. L'adaptation au niveau du système concerne le système lui-même qui est alors capable de s'adapter comme il le fait pour l'application. On parle de systèmes adaptables ou de plateformes autonomes comme la plateforme MUSIC [110]. Il est préférable que le système d'adaptation soit exécuté sur chaque périphérique et non pas sur un serveur. En effet ceci permet de fournir des services d'adaptation plus flexibles et plus puissants dans la mesure où ils font eux mêmes partie de l'environnement d'exécution.

L'objectif de cette partie est de réaliser un état de l'art sur les plateformes d'adaptation dans un cadre applicatif lié à l'informatique ubiquitaire. Elle est organisée en six parties. Nous commençons par un état de l'art des systèmes d'adaptation pour introduire les besoins futurs des plates-formes. Ensuite nous présenterons les technologies existantes pour les systèmes et les plates-formes d'adaptation en terminant par les modèles d'application et l'ingénierie logicielle. Nous conclurons cette partie par une classification des approches existantes.

3

SYSTÈMES D'ADAPTATION

Nous présentons ici les principaux concepts des systèmes d'adaptation en environnement ubiquitaire. Nous commencerons par la présentation de l'informatique pervasive, puis des plateformes d'adaptation et de leur rôle dans l'informatique ubiquitaire.

3.1 QU'EST-CE QUE L'ADAPTATION ?

Le terme d'adaptation est associé à l'idée "d'un changement de structure ou de fonctionnement d'un organisme qui le rend mieux adapté à son environnement" (cf. réf. Définition Oxford Dict.). Selon cette définition, l'adaptation logicielle agit de deux façons : en changeant la structure du logiciel ou en modifiant la fonction. Ces changements ont pour but de rendre application mieux adaptée aux évolutions de son contexte. Le contexte inclut celui de l'environnement d'exécution et celui de l'usage de l'application et du matériel. L'environnement d'exécution concerne toute information observable par le système, comme les interactions de l'utilisateur, les capteurs et périphériques externes, les infrastructures réseau, etc. [98]. L'usage représente tout ce qui concerne la relation qu'entretient l'utilisateur avec l'application c'est-à-dire ses préférences, ses handicaps, ses intentions, etc.

3.2 QUAND ET POURQUOI ADAPTER ?

Il y a trois raisons pour adapter une application [33]. Chacune correspond à un type d'adaptation. Il s'agit de l'adaptation réactive, de l'adaptation évolutive et de adaptation par intégration. L'adaptation réactive change le comportement de l'application en accord avec les changements de contexte d'exécution ou des préférences de l'utilisateur. L'adaptation évolutive étend les fonctionnalités d'une application pour corriger ses erreurs ou pour augmenter sa performance. Un changement de la qualité de service ou des souhaits de l'utilisateur peut déclencher une adaptation évolutive. L'adaptation par intégration traite les problèmes des services ou des composants incompatibles (interfaces matérielles/logicielles différentes, protocoles différents, etc.). Ce type d'adaptation est souvent utilisé lorsqu'une décision d'adaptation vient d'être prise. Par exemple, lorsqu'une adaptation a été déclenchée par un changement de contexte, la mise en œuvre de cette adaptation peut conduire à ce qu'un composant ne puisse plus communiquer avec un autre parce qu'ils font appels à des protocoles différents. Dans ce cas, le système peut recourir à une adaptation par intégration.

De plus, il existe trois façons de décider quand faire une adaptation : décision dirigée par le contexte, décision dirigée par la QoS et décision dirigée par l'utilisateur. Elles fonctionnent souvent de pair mais avec des priorités différentes selon les situations (en fonction des intentions de l'application et des motivations de l'adaptation). Les méthodes dirigées par le contexte prennent une décision quand elles reçoivent un événement de changement de contexte et après avoir analysé l'ensemble des informations contextuelles. L'adaptation réactive est liée aux méthodes dirigées par le contexte. Les méthodes dirigées par la QoS réagissent aux événements de qualité de service et sont essentiellement tournées vers la qualité globale du système. Dans l'adaptation évolutive, c'est la QoS qui est prioritaire. Les méthodes dirigées par l'utilisateur ne s'intéressent qu'aux modifications de configurations du système ou de préférences par l'utilisateur et ont, généralement, la plus haute priorité.

3.3 BOUCLE D'ADAPTATION

La boucle générale d'un système adaptatif inclut l'observation de l'environnement, le choix des adaptations et leur mise en œuvre. Nous l'appellerons boucle CADA qui signifie "Collection, Analysis, Decision and Action". La fig.6 décrit une telle boucle d'adaptation dans laquelle le système peut être autonome ou impliquer l'humain.

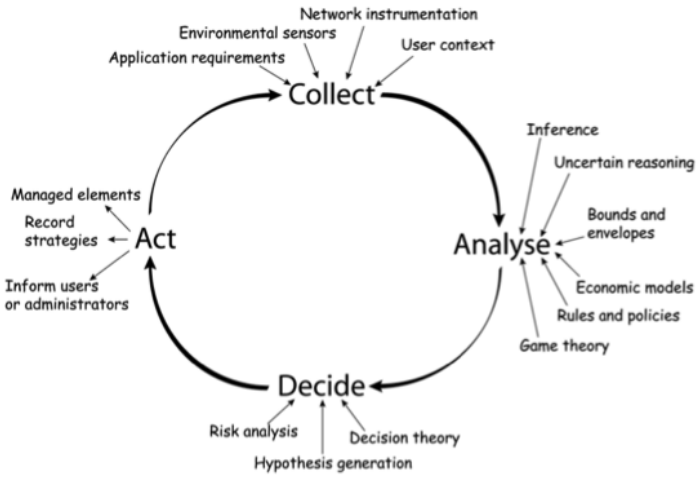

FIGURE 6 – Boucle d'adaptation CADA [47]

Tout d'abord, le collecteur de contexte récupère les informations de l'environnement d'exécution à partir de l'OS et le contexte utilisateur (cf. fig.7). Ces informations sont appelées méta-informations de contexte. Ensuite, le collecteur exploite ces méta-informations pour construire un modèle de contexte de haut niveau. Il s'agit d'un modèle abstrait qui représente l'information de contexte. L'analyseur évalue ces informations de contexte et produit un plan d'adaptation. Il peut produire un plan ou une liste de plans. Le décisionneur analyse le risque associé à chaque plan et prend la décision finale. Pour terminer, l'acteur exécute l'adaptation décrite par le plan choisi. Le cœur de l'adaptation est constitué de l'analyseur et du décisionneur. Le collecteur et l'acteur en sont les collaborateurs.

Dans les deux sections suivantes, nous présentons différents types d'adaptations sous les points de vue du niveau d'adaptation et de la réalisation de l'adaptation.

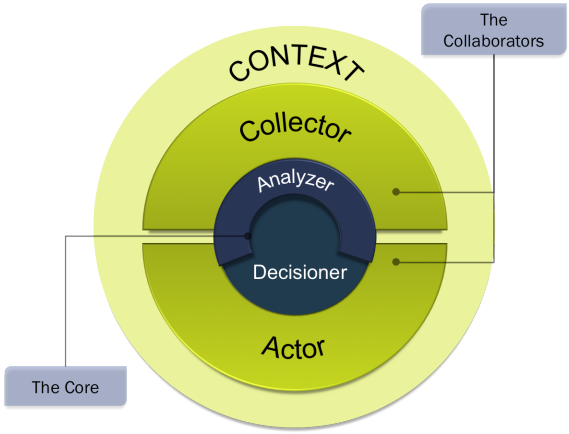

FIGURE 7 – Architecture générique de CADA

3.4 ADAPTATION AUTONOMIQUE VS SUPERVISÉE

Du point de vue du niveau de l'adaptation (c'est-à-dire des architectures logicielle adaptables), il y a deux types d'adaptations.

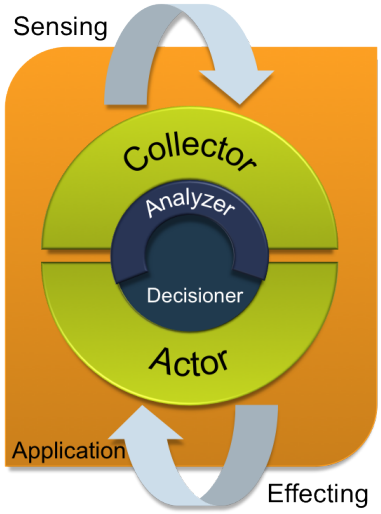

FIGURE 8 – Adaptation autonomique

Adaptation autonomique : c'est l'application qui gère l'adaptation. Le cœur et les collaborateurs sont inclus dans l'application. Ce type d'applications autonomes s'adapte par lui-même sans support extérieur (cf. fig. 8). Il ne nécessite ni plateforme d'adaptation ni middleware d'adaptation (généralement, le middleware assure une part du service d'adaptation comme celle constituée du collecteur de contexte et de l'acteur). Chaque application doit proposer sa propre solution d'adaptation. Par conséquent le développeur d'une application auto-adaptative ne peut pas facilement réutiliser le code lié à l'adaptation. C'est pour cette raison que, dans ce chapitre, nous avons choisi de ne pas prendre en considération ce type d'adaptation.

Adaptation supervisée : C'est la plateforme qui gère les adaptations (cf. fig.9) et l'adaptation est transparente aux applications. La plateforme peut inclure le cœur et les collaborateurs ou seulement le cœur (un décisionneur et un analyseur intelligent). Dans le cas où seul le cœur est présent, il utilise des

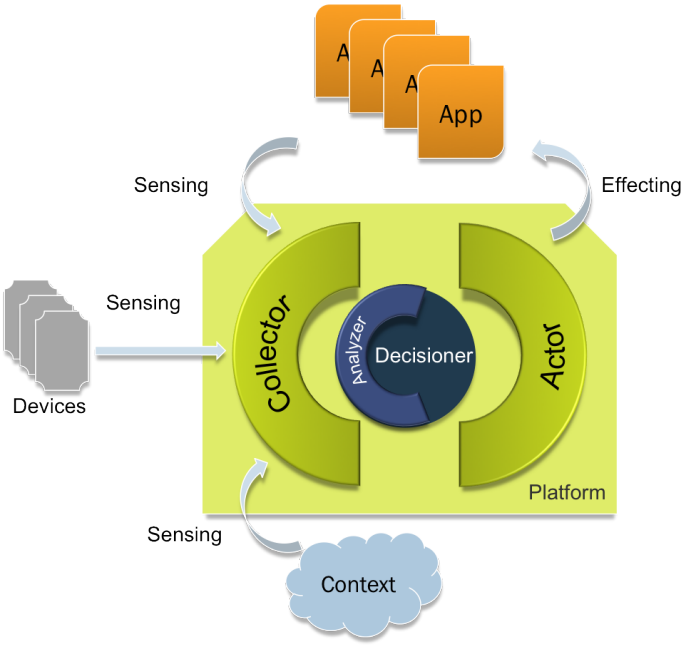

FIGURE 9 – Adaptation supervisée

collaborateurs offerts par des fournisseurs tiers sous forme de services ou de combinaisons de services. Par exemple, la plateforme peut contenir un collecteur de contexte local, faisant appel à des collecteurs de contexte distants pour récupérer les informations de contexte des utilisateurs. Ces collaborateurs prennent en charge les mécanismes de récupération du contexte et de communication. Par exemple, un acteur de middleware d'adaptation se charge de la transparence des opérations d'adaptation distribuées et des opérations inter-plateformes. Il n'a aucune intelligence dans le sens où il ne fait qu'appliquer le plan d'adaptation choisi. L'adaptation supervisée présente l'avantage que les développeurs de l'application n'ont pas à se préoccuper de l'adaptation et peuvent se concentrer sur la logique métier de l'application. Le défi que rencontrent de telles plateformes est comment prendre la bonne décision pour toute application et en toute situation.

3.5 ADAPTATION STRUCTURELLE ET COMPORTEMENTALE

La plateforme d'adaptation distingue deux catégories du point de vue de la réalisation de l'adaptation : changement de structure et changement de comportement [6].

Adaptation structurelle : la structure même du programme change (par programme nous entendons l'application, la plateforme et le middleware). Pour les systèmes à base de composants, ce changement peut porter sur la composition par ajout ou suppression de composants. Par ailleurs, la migration de composants est également possible : en changeant l'hôte d'accueil d'un composant et en le liant dynamiquement aux autres composants. L'opération de migration nécessite de conserver la cohérence des états avant et après la migration. Pour les systèmes à base de services, les changements se font au niveau de la configuration des services. Certaines plateformes ou middlewares comme SamProc [122] supportent la migration de services. L'adaptation structurelle distingue deux propriétés : la configurabilité et la reconfigurabilité. La configurabilité est la possibilité de modifier la structure d'un système pour le mettre dans

différentes configurations [67]. Certains chercheurs désignent la configurabilité à l'exécution par le terme de reconfigurabilité [24].

Adaptation comportementale : Elle est également appelée adaptation fonctionnelle. Il s'agit de changer le comportement du programme. Cela signifie d'ajouter ou de supprimer certaines fonctions ou d'en changer la qualité de service. De plus, est possible d'utiliser des services ou des composants équivalents en remplacement de certains services ou composants utilisés. Dans sa thèse, An P.K [77] parle d'adaptation fonctionnelle quand la fonctionnalité du système est modifiée par l'application, dans les autres cas, les adaptations sont appelées non fonctionnelles.

Comme signalé par [98], l'adaptation comportementale consiste à modifier des paramètres de composants ou de connecteurs ou à modifier l'architecture par ajout/suppression/remplacement de composants ou de connecteurs. Tous les mécanismes d'adaptation (aussi bien le changement de structure que de comportement) sont battis sur quatre opérations de base : **ajouter**, **supprimer**, **connecter** et **déconnecter**. Par exemple, pour remplacer un composant par un autre, nous devrons exécuter les opérations suivantes : déconnecter-supprimer-ajouter-connecter. Ainsi, un système qui peut accomplir ces quatre opérations de base dynamiquement est un système qui permet l'adaptation. A ces opérations de base nous ajouterons deux opérations indispensables à la migration de composants qui sont la sauvegarde et la restitution de l'état.

Comme nous l'avons dit précédemment, les systèmes d'adaptation constituent une solution à l'informatique ubiquitaire. Par la suite, nous présenterons ce que sont les systèmes d'adaptation et leur rôle dans l'informatique ubiquitaire.

3.6 QU'EST-CE QU'UN SYSTÈME D'ADAPTATION ?

Un système d'adaptation est un système qui adapte l'application à son environnement. Un tel système est constitué d'un middleware et d'une plateforme d'adaptation. Le middleware d'adaptation fournit les mécanismes permettant les tâches d'adaptation. Il constitue également une couche qui gère l'hétérogénéité ainsi que la collecte et la diffusion des informations de contexte. La plateforme d'adaptation, quant à elle, fournit les services d'analyse intelligente, d'heuristiques de planification et de prise de décision. D'un point de vue de l'infrastructure, il s'agit de deux systèmes différents.

3.6.1 *Systèmes d'adaptation locaux et distribués*

Un système d'adaptation local fonctionne sur une seule machine. Les applications utilisant ce système peuvent être distribuées mais tous les composants du système d'adaptation sont déployés sur une machine unique. Il ressemble et se comporte comme un serveur centralisé d'adaptation qui assure le service de collection de contexte, celui de planification de l'adaptation, etc. Le désavantage majeur d'un tel système est qu'il ne peut pas s'adapter lui-même aux changements d'environnement distants. Un serveur d'adaptation centralisé ne peut avoir qu'un nombre limité de clients en raison de goulots d'étranglements comme la bande passante du réseau, la puissance de calcul et les capacités de stockage. De tels services d'adaptation sont fréquemment utilisés mais sont de gros consommateurs de ressources. De plus, ils ont besoin de

grandes capacités de calculs pour assurer un fonctionnement performant en temps réel de sorte qu'un simple serveur ne suffit pas à garantir une bonne performance et ne constitue pas une bonne solution pour l'informatique ubiquitaire.

Un système distribué supporte l'adaptation au niveau du système lui-même et peut déployer ses composants au travers d'un réseau. La plateforme MUSIC supporte l'adaptation au niveau du système lui-même. Elle supporte aussi des adaptations et sur les appareils interconnectés par un réseau. Sur MUSIC, ce réseau s'appele le domaine d'adaptation [110]. Music doit fixer des rôles comme le superviseur du domaine (Master) et les hôtes du domaine (Slaves) avant son déploiement. MUSIC définit la notion de cœur de système qui a les fonctionnalités minimales pour contrôler le déploiement et le cycle de vie des composants. La plateforme MUSIC déploie le cœur sur chaque périphérique du domaine d'adaptation. De plus, le système peut s'adapter lui-même comme l'application le fait.

Les systèmes d'adaptation, qu'ils soient locaux ou distribués jouent le même rôle dans l'informatique ubiquitaire. C'est ce rôle que nous allons détailler maintenant.

3.6.2 *Rôle du système d'adaptation*

Dans l'informatique ubiquitaire, les systèmes d'adaptation assurent quatre rôles comme indiqué précédemment, ce sont : la gestion du contexte, la planification, la décision et la middleware (cf. fig.7). Tout comme dans les étapes de la boucle d'adaptation que nous avions décrite précédemment, on distingue deux types de rôles : les rôles cœur et les rôles collaborateurs. Les rôles collaborateurs ne sont pas impératifs et peuvent être remplacées par le middleware, tandis que, conformément à la définition d'un système d'adaptation, les rôles de planification et de décision sont indispensables.

Gestionnaire de contexte : c'est un rôle collaborateur. Le système doit collecter les informations de contexte et construire les modèles de contexte de haut niveau. Les informations de contexte peuvent provenir de différents périphériques ou de différents systèmes logiciels et même de réseaux sociaux. Par conséquent, le système peut utiliser un middleware de contexte pour assurer la collection et la construction des modèles. Néanmoins, il peut aussi déployer le gestionnaire de contexte en tant que composant ou service quelque part dans le domaine d'adaptation.

Planificateur : le système soit produire des plans d'adaptation dynamique à l'exécution. Une heuristique de planification est utilisée pour déterminer la meilleure configuration de l'application. Le raisonnement est basé sur les informations de contexte.

Décisionneur : à partir de la liste de plans d'adaptations produite par l'heuristique, le système analyse le risque de chacun des plans, prend la décision et demande à l'acteur de réaliser les changements.

Middleware : Il cache l'hétérogénéité du réseau. Il fournit des modèles avancés de coordination entre les entités distribuées. Finalement, il rend la répartition aussi transparente que possible [72]. Nous allons maintenant nous intéresser aux technologies utilisées dans les systèmes d'adaptation.

4

TECHNOLOGIES DES SYSTÈMES D'ADAPTATION

En accord avec la littérature, nous proposons une architecture générale de systèmes d'adaptation (cf. fig.10). Il s'agit d'une architecture à n couches. Du bas vers le haut, on trouve la couche du matériel, la couche de l'OS, la couche de l'adaptation et la couche des applications. La couche matérielle représente les périphériques mobiles, les ordinateurs, les équipements de réseaux, etc. Au-dessus de cette couche se situe l'OS comme Windows, Linux, iOS [8] et Android [56].

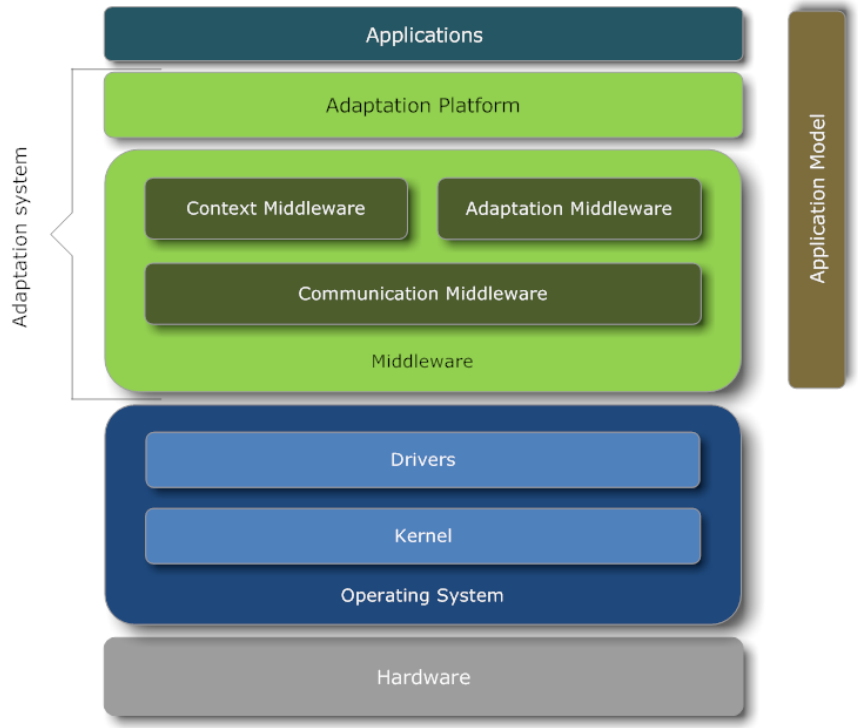

FIGURE 10 – Modèle de conception d'architecture de plateforme d'adaptation

La couche de la plateforme d'adaptation fournit l'analyse intelligente et la prise de décision tandis que le middleware est constitué du context middleware, de l'adaptation middleware et du communication middleware. Le middleware et la plateforme constituent la couche d'adaptation. Tout en haut, se situe la couche application. Les applications seront supervisées et modifiées par le système d'adaptation. Le modèle d'application (Application Model) concerne à la fois la couche application et la couche adaptation. Il peut être différent ou identique pour le système d'adaptation et pour l'application.

Dans cette partie nous nous concentrerons sur le système d'adaptation (Adaptation System) et sur le modèle d'application (Application Model). Nous décrirons leur rôle, leur composition, les technologies utilisées et les relations entre eux.

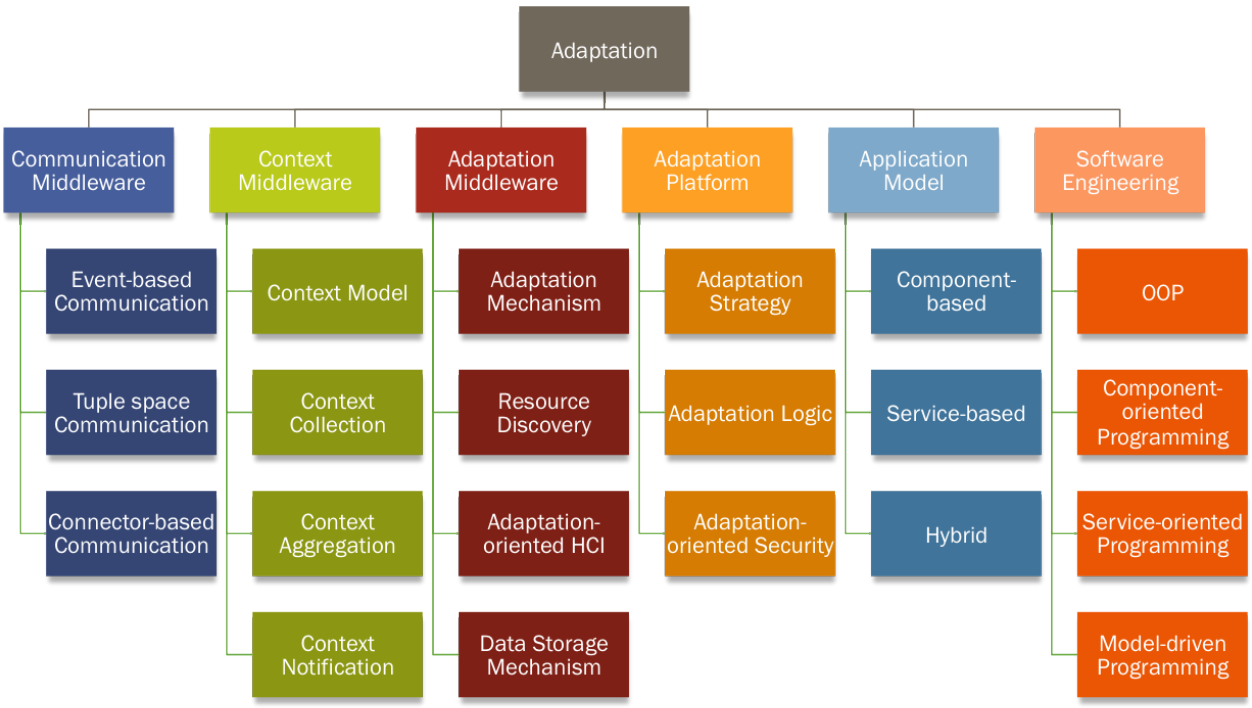

FIGURE 11 – Taxonomie des technologies des plateformes d'adaptation

En accord avec cette architecture générale, nous proposons la taxonomie des technologies des systèmes d'adaptation (cf. fig.11). Nous présenterons chacune de ces technologies, dans l'ordre de la figure, de la gauche vers la droite. Nous allons commencer par le middleware de communication.

4.1 COMMUNICATION MIDDLEWARE

Le middleware de communication est la base des systèmes distribués. L'application adaptative et les couches supérieures du middleware utilisent le communication middleware pour les communications intra et inter applications. Le concept de middleware étant maintenant connu, nous allons focaliser sur les trois mécanismes de communication les plus répandus (cf. fig.12) :

— communications basées sur les événements
— communications basées sur les tuple-spaces
— communications basées sur les connecteurs

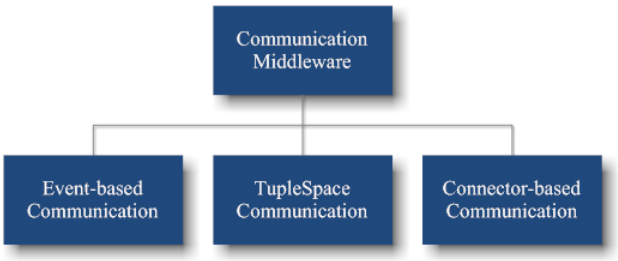

FIGURE 12 – Technologies des middlewares de communication

Les mécanismes basés sur les événements permettent la communication par événements. Les événements peuvent être filtrés, classés par type, et/ou être associés à un délai.

Le mécanisme basé sur les événements dissocie la communication de l'espace-temps et du flux de contrôle. Il accepte les communications multipoints, les réseaux intermittents et les réseaux ad'hoc Gaia [107], DREAM [81], Milan [104, 50] and Siena [32] sont des approches basées sur les événements.

Le mécanisme de tuple-spaces permet la communication par mémoire partagée. Il ressemble à un tableau blanc. Les producteurs placent des messages dans les tuple-spaces dans lequel les consommateurs les lisent. Les tâches peuvent générer des requêtes asynchrones. Le tuple-space fourni une structure de données commune publique pour les communications qui est appelé tuple. Il est réalisé le plus souvent par un couple clé/valeur. Les middlewares ubiquitaire de TOTA [85] et MESHMdl [69] utilisent le mécanisme tuple-space pour la communication.

Le mécanisme basé sur les connecteurs utilise la communication par connecteur. Sur un système à composants, le connecteur agit comme un conteneur de données. Les connecteurs fournissent aux composants une interface d'entrées/sorties unifiée. Ainsi, les communications à distance sont transparentes aux composants de l'application puisque le connecteur prend en charge ces opérations. Les connecteurs peuvent également offrir des fonctions avancées. Par exemple, un utilisateur (joueur1) utilise son téléphone pour jouer avec un autre utilisateur (joueur2) à travers une connexion bluetooth. Plus tard, parce que le joueur2 s'est déplacé, la connexion bluetooth est perdue. Le connecteur peut utiliser le service SRD (Service Resource Discovery) pour trouver une machine pouvant se connecter au joueur 2 par bluetooth et au joueur 1 par wifi. De sorte que ce connecteur utilisera cette machine comme relais pour connecter les deux joueurs. Evidemment, ces opérations de reconnexion sont transparentes aux utilisateurs. Kalimucho [43] utilise ce type de middleware de communication qui est appelé Korrontea [28]. C'est un middleware de communication distribué basé sur les connecteurs.

Nous allons nous intéresser maintenant au Context Middleware de la figure 11.

4.2 CONTEXT MIDDLEWARE

La prise en compte de l'environnement dans une application peut augmenter la difficulté de développement et la complexité de l'application. De plus, cela compromet la réutilisation de code. Le context middleware doit prendre en charge les opérations de collecte, d'agrégation et d'analyse du contexte et fournir un mécanisme de notification de contexte pour l'application et/ou la plateforme d'adaptation.

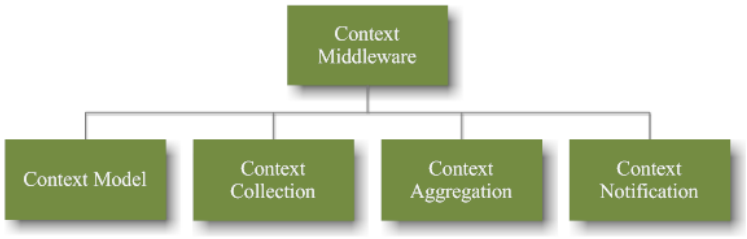

FIGURE 13 – Technologies des middlewares de contexte

Avant de présenter les technologies des middlewares de contexte, nous discutons d'abord l'abstraction de contexte. "Information from physical Sensors, called low-level context and acquired without any further interpretation, can be meaningless, trivial, vulnerable to small changes, or uncertain" [21]. Une façon de résoudre ce problème est de produire des informations de contexte de plus haut niveau par dérivation des informations de contexte directement capturées par les capteurs. Dans ce but l'abstraction d'informations de contexte s'étend sur 4 niveaux [21] (figure 14) :

- Les "Raw data" sont les données directement produites par les capteurs.
- Les "Low level context" sont les données composées des données produites par les capteurs (Raw data) et avec des informations des capteurs et de données spatio - temporelles.
- Les "High Level Context" sont des informations de contexte de haut niveau sémantique réutilisables dans différentes applications.
- Les "Situation" sont les informations contextuelles de plus haut niveau. Elles représentent une évolution du contexte déterminée par des relations entre les "High level context", susceptible de provoquer des reconfigurations. Par exemple, l'utilisateur se déplace vers un bâtiment ou utilisateur entre dans une zone d'exposition. Des situations liées par des relations peuvent, elles-mêmes, produire de nouvelles situations.

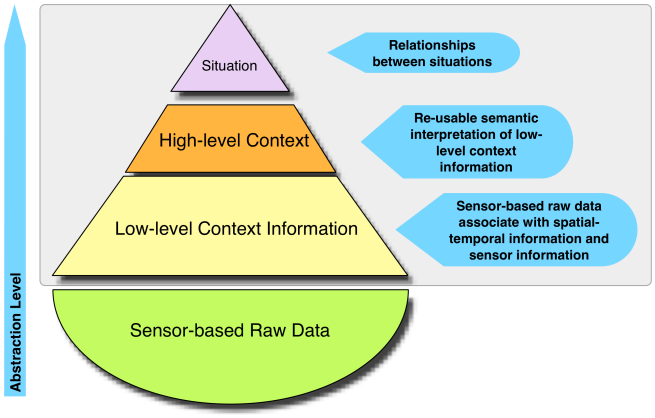

FIGURE 14 – Niveaux d'abstraction du contexte

Dans cette partie nous présenterons les technologies mises en œuvre dans le context middleware à savoir le **modèle**, la **collecte**, **l'agrégation** et la **notification** de contexte (cf. fig.13).

4.3 MODÈLE DE CONTEXTE

Le modèle de contexte est le format utilisé par le context middleware pour fournir à la couche supérieure les informations de contexte. Ce modèle doit être structuré, consistant, décomposable, assemblable et extensible. Dans [126], du point de vue de la structure de données, le modèle de contexte est divisé en six catégories : key-value models, markup-scheme models, graphicals models, object oriented models, logic based models and ontology bases models. Les "Key-value models" sont les structures de données les plus simples pour modéliser contexte. Ces modèles utilisent des paires de type de key-value pour représenter les informations contextuelles (par exemple distance-29Km). Les approches de "markup-scheme models" utilisent une structure de données hiérarchique constituée de balises avec des attributs et du contenu. Le contenu des balises est généralement défini récursivement par d'autres balises, par exemple, *Composite Capabilities / Preferences Profile* (CC/PP)[142]. Les "graphicals models" utilisent des graphiques pour représenter des structures de donnée du contexte, ainsi Bauer utilise *Unified Modeling Language* (UML) pour modéliser le contexte dans [16]. Les approches de type "object oriented models" modélisent des données contextuelles comme des objets pour bénéficier des avantages comme l'encapsulation et la réutilisabilité. Dans "logic based model", le contexte est donc défini comme des faits, des expressions et des règles. L'un des premiers modèles de contexte basé sur la logique fut publié comme *"Formalizing Context"* au début de année 1993 par McCarthy et son groupe à Stanford [86, 87]. Les approches "ontology based" utilisent des ontologies pour construire le modèle de contexte. Les ontologies sont un instrument prometteur de préciser les concepts et les interrelations.

Dans [21], sont décrits trois types d'approches courantes de modélisation du contexte : object-rôle, spatial, ontology based ainsi que des approches hybrides.

Ces trois types de modèles de contexte sont les plus utilisés dans l'informatique ubiquitaire actuelle (cf. fig.15).

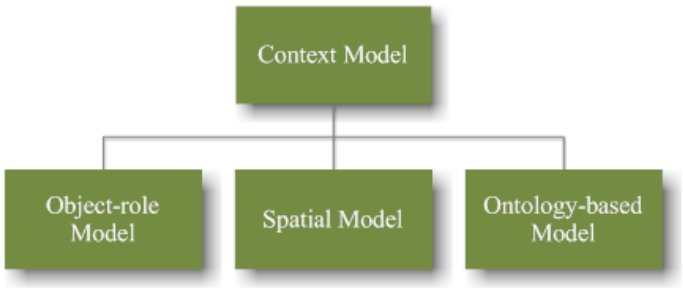

FIGURE 15 – Modèle de contexte

Object-role model : les approches de modélisation du contexte orientées objet bénéficient de l'encapsulation et de la réutilisation. Elles peuvent être appliquées à la dynamique du contexte dans un environnement ubiquitaire [126]. Elles rendent inutile la dynamique des approches de modélisation du contexte antérieur comme les couples key-value. Parmi ces approches, on peut citer le projet TEA [120], Active Object Model [38] et Context-Modeling-Language [64]. Elles peuvent être combinées à des modèles graphiques comme ORM (Object-Role Model) qui permettent l'analyse et la conception des contraintes de contexte et les représentations relationnelles pour des abstractions de haut niveau du contexte. Ces modèles

offrent un support plus complet pour la capture et l'évaluation d'informations historiques et imparfaites que la plupart des autres approches de modélisation de contexte [21].

Spatial Model : dans le monde réel, les gens utilisent leur téléphone mobile pour obtenir des réponses à des questions comme : où es-tu ? Où pouvons-nous nous rencontrer ? Où suis-je ? etc. Dans l'informatique ubiquitaire, la localisation et les ressources environnantes sont très importantes. C'est pourquoi la localisation et l'espace sont des facteurs essentiels du contexte. Schilit, Adams et Want définissent le contexte comme "Where you are ? Who you are with ? What resources are nearby" [119]. Le spatial model concerne les informations de localisation. Il existe deux systèmes de coordonnées pour représenter les informations de localisation : les coordonnées géométriques et les coordonnées symboliques. Les coordonnées géométriques représentent des points ou des zones dans un espace métrique comme le fait le GPS et le capteur GPS econstitue l'équipement privilégié dans ce but. Les coordonnées symboliques représentent la position comme un identifiant comme la position d'un capteur, la position d'un hotspot wifi, un numéro de chambre, un nom d'immeuble, etc. Une caméra, un téléphone ou quelque chose d'autre peut être un moyen de collecte. Augmented World Model est développé dans le projet Nexus [93] et utilise à la fois un système de coordonnées géométriques et symboliques. Le spatial model doit modéliser à la fois la position et les relations entre les objets. L'information de position doit inclure un point (point de l'objet) et un intervalle. Ceci permet de positionner d'autres objets dans l'intervalle qui sont appelés voisins proches [21]. Comme signalé auparavant, les spatial-models sont particulièrement appropriés à l'informatique ubiquitaire. La faiblesse de ce type de modèle est qu'il se concentre essentiellement sur la position. C'est pourquoi, les chercheurs ont associé ce modèle avec d'autres pour construire des modèles hybrides comme Spatial + Object-Based Model [93] Spatial + Ontology-Based Model [53].

Ontology Based Model : Les modèles basés sur les ontologies offrent de bonnes capacités et facilitent la réalisation de raisonnements. Depuis peu, ils sont massivement utilisés. Le concept d'ontologie vient de la philosophie où il désigne l'étude des propriétés générales de tout ce qui existe. Studer et al le définissent comme : "an ontology is a formal explicit specification of a chaired conceptualization" [127]. Les ontologies peuvent être représentées de différences façons comme des langages naturels ou des langages logiques. Dans le domaine du web sémantique, on trouve essentiellement deux langages de description pour les ontologies : RDF (Resource Description Framework) et OWL (Ontology Web Language). De nos jours, les middlewares de contextes basés sur les ontologies utilisent souvent OWL pour le modèle de contexte comme le font CoBrA [35] et SOCAM [59]. CoBra et SOCAM utilisent respectivement les ontologies SOUPA [36] et CONON [153]. Le middleware Gaia [107] utilise les langages d'ontologie DAML et OIL (prédécesseurs d'OWL) pour construire le modèle de contexte. Ce modèle est utilisé pour représenter les informations de contexte de haut niveau en vue du raisonnement sur le contexte. Le collecteur de contexte récupère les informations brutes des capteurs à partir desquelles des données de contexte complexes seront construites avec OWL par l'agrégateur de contexte puis transmises à l'analyseur de contexte.

Avant d'aborder les modèles hybrides, nous présentons un tableau (cf. tableau.1) qui compare les trois modèles présentés précédemment. La satisfaction partielle des exigences est indiquée par ' ' dans le tableau, la satisfaction complète est indiquée par '+' et la non satisfaction par '-'.

Hybrid Model : pourquoi aurions-nous besoin d'un modèle hybride ? La réponse habituelle à cette question est : car les solutions existantes ne sont pas satisfaisantes. Donc, le mélange de ces solutions

	Object-role	Spatial	Ontological
Heterogeneity	+	~	+
Mobility	~	+	–
Relationships	~	~	+
Timeliness	+	+	–
Imperfection	~	~	–
Reasoning	~	–	+
Usability	+	~	~
Efficiency	~	+	–

TABLE 1 – Comparaison de modélisation de contexte par des exigences de contexte [21]

imparfaites permet de trouver une meilleure représentation. Les modèles de contexte présentent les points forts et les points faibles montrés dans le tableau 1. Il existe plusieurs moyens de mélanger ces modèles comme : Henricksen et al [68], le framework CARE [1], le modèle Spatial+Object [93], Spatial+Ontology [53] et COSMOS [29] (Dans l'article [29], COSMOS a été couplé avec une ontologie, Bouzeghoub et al. ont montré la complémentarité des approches orientés processus (approche impérative) et orientée ontologie (raisonnement logique)).

4.4 COLLECTE DE CONTEXTE

Le but de la collecte de contexte est de récupérer les informations sur l'état d'utilisation des ressources et les performances de chaque périphérique et OS. De plus, pour la collecte de contextes centrés sur l'utilisateur, nous devons également récupérer la position géographique, les ressources environnantes et temporelles. La collecte de contexte fournit un niveau d'abstraction logiciel des capteurs pour la couche supérieure ou directement pour l'application comme le fait Context Toolkit [114]. Context Toolkit fait une abstraction des capteurs en tant que widgets dans le middleware. Ces widgets fournissent une interface unifiée. Les approches similaires sont : SOCAM Context Provider [60], CoBrA Context Acquisition Component [34] et EMM Context Representer [151].

En accord avec la définition actuelle de contexte, la collecte de contexte inclut également des informations sur la façon dont les utilisateurs interagissent avec leur environnement comme, par exemple, des informations sur les réseaux sociaux (quelles couleurs ils aiment, quels sont leurs hobbys, qu'est-ce qu'ils aiment manger, etc.). Dans [150] et [117] une technologie pour résoudre ces problèmes est représentée.

Le rôle de la collecte de contexte dans un système d'adaptation est de fournir des méta-informations de contexte à chaque partie du système.

4.4.1 *Agrégation de contexte*

L'objectif de l'agrégation de contexte est de filtrer, analyser et construire les informations de contexte de haut niveau. Après quoi, les informations de contexte de bas niveau (raw data) seront traduites dans le modèle de contexte en informations de contexte de haut niveau. Ces informations de haut niveau permettront de simplifier la tâche d'analyse de contexte et le raisonnement dans le middleware d'adaptation. Les middlewares de contexte basés sur les ontologies intègrent les données de bas niveau dans un modèle de contexte unifié décrit en OWL. Pour éliminer l'imprécision et les conflits des données de bas niveau, l'agrégateur de contexte devra interagir avec le raisonneur de contexte. Le raisonneur de contexte, dans l'architecture générale présentée précédemment, se situe dans la couche plateforme d'adaptation. Toutefois, dans un système sans plateforme d'adaptation, le raisonneur de contexte est habituellement inclus dans le middleware

4.4.2 *Notification de contexte*

Les moyens de notifier la couche supérieure des changements de contexte sont généralement soit actifs soit passifs. Dans la notification active, quand le contexte change, la couche supérieure est immédiatement notifiée, généralement par un mécanisme d'événements. Dans la notification passive, quand la couche supérieure doit détecter un changement de contexte, elle utilise une API. Dans ce but, elle interroge une base de données ou lit un fichier de log ou utilise un mécanisme de tableaux blanc comme tupple space qui s'appuie sur une mémoire partagée logicielle.

Les systèmes dirigés par le contexte utilisent en générale la notification par événements. Ces mécanismes de notification offrent une bonne extensibilité, un bon passage à l'échelle et un couplage faible. Par exemple, Context Toolkit et Gaia utilisent ce mécanisme.

Dans un système d'adaptation distribué utilisant la notification par événements, le middleware de contexte doit identifier quels événements doivent ou pas être notifiés mais également à qui ils doivent l'être. Evidemment, selon le type d'application, la sélection et la destination des notifications seront différents. Lorsque l'on utilise la notification par événements, le middleware de contexte doit fournir un mécanisme de pré-analyse pour tenir compte de ces différents types d'applications. Lorsque l'on utilise un mécanisme de tableau blanc, il doit être capable de savoir quels messages seront décrits après un certain temps et lesquels doivent être persistants.

Le tableau 2 présente une synthèse de ces approches.

Nous allons continuer maintenant avec le middleware d'adaptation. Il est constitué de quatre parties : mécanismes d'adaptation, découverte de ressources, adaptations orientés Interactions Homme-Machine (IHM) et stockage de données.

4.5 MIDDLEWARE D'ADAPTATION

Le middleware d'adaptation doit fournir des mécanismes et des services de base pour l'adaptation. Ces services de base doivent être hautement configurables et pouvoir dynamiquement être ajoutés/retirés.

	Context Model			Context Aggregation
	Object-role	Spatial models	Ontology based models	
Active Object Model	X			
CARE	X	X		
CoBrA			X	
Context Toolkit	X			X
COSMOS	X			X
A. Frank		X	X	
GAIA	X			X
Nearest Neighbor	X			
SOCAM			X	X

TABLE 2 – Comparaison des middlewares de contexte

De plus, la gestion de ces services est sous le contrôle de la plateforme d'adaptation et transparente à l'application. Les mécanismes d'adaptation sont les services fondamentaux du middleware d'adaptation. Ils sont en lien direct avec le modèle d'application et la méthode de développement. Par exemple, nous verrons que si le middleware d'adaptation fournit un mécanisme de micro-noyau, le modèle d'applications doit être un modèle à composants ou un modèle hybride. Si le mécanisme est basé sur AOP (Aspect Oriented Programming), la méthode de développement doit être AOP.

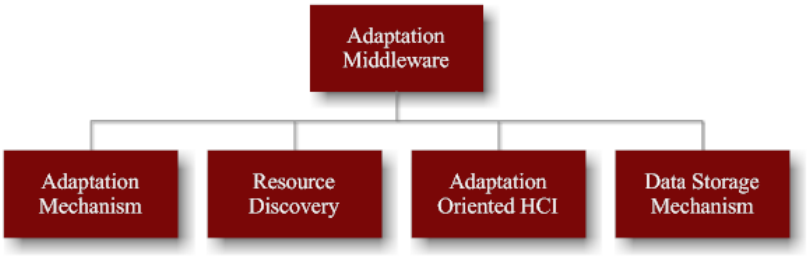

FIGURE 16 – Technologies des middlewares d'adaptation

La figure 16 présente ces mécanismes et services d'adaptation. Nous allons les décrire en détail par la suite en commençant par les mécanismes d'adaptation.

4.5.1 *Mécanismes d'adaptation*

Cette partie de la technologie est relativement mature. La plateforme d'adaptation supervise les applications en utilisant le plus souvent des technologies de réflexivité et de chargement dynamique de modules. Selon la définition de Bobrow et al [25], la réflexivité implique la capacité d'un système logiciel d'observer et de raisonner sur son propre état ainsi que de modifier sa propre exécution. Il y a deux catégories de technologies de chargement dynamiques de modules : micro-noyau et AOP (cf. fig.17)

La technologie micro-noyau utilise l'expérience des systèmes d'exploitation micro-noyau. Dans [23] est présenté le concept de "operating system as application program". Seules les fonctions de base sont chargées dans le noyau, les autres seront chargées au runtime comme des modules indépendants. En accord avec cette définition, il existe plusieurs plateformes à chargement dynamique de modules la plus connue

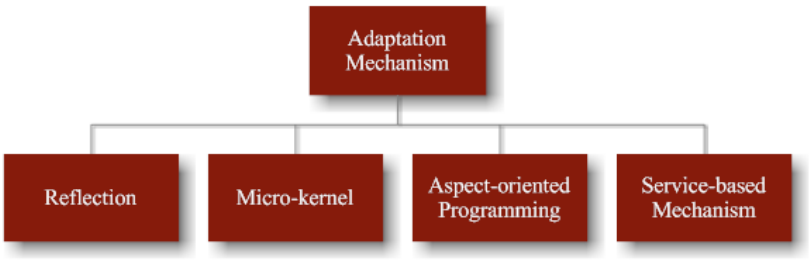

FIGURE 17 – Mécanismes d'adaptation

étant OSGi [3]. Pour l'informatique ubiquitaire, plusieurs plateformes sont basées sur cette technologie comme MUSIC [110], BASE [17] et MundoCore [2].

La technologie AOP [78] est une approche d'ingénierie logicielle qui permet la séparation des préoccupations comme la qualité de service, la sécurité, la tolérance aux pannes. Le développement AOP contraint les développeurs à prédéfinir des points de coupe, de jonction et des greffons. Un point de jonction est un point dans le programme où des comportements additionnels peuvent être ajoutés. Un point de coupe est la description d'un point de jonction. Un greffon est un morceau de code qui peut être lancé avant ou après un point de jonction. Les greffons sont tissés dans le programme quand un point de jonction correspond à un point de coupe. Le tissage est le mécanisme d'intégration des greffons. Il peut se produire soit à la compilation soit à l'exécution. WComp [135] est un middleware adaptatif basé sur AOP. Il utilise un concept spécifique appelé Aspects d'Assemblage pour l'Adaptation.

D'autre part, la technologie basée services, est très répandue pour réaliser des middlewares d'adaptation. Les systèmes orientés services offrent de la flexibilité pour la gestion et l'intégration dynamique de nouveaux périphériques. Ces systèmes sont indépendants des langages, des plateformes logicielles et du matériel. La description d'un service ne concerne que ce que ce service utilise, jamais comment il est réalisé. Ces capacités correspondent exactement aux besoins de l'informatique ubiquitaire, à savoir l'hétérogénéité, la scalabilité, la sécurité, la mobilité et la découverte [135]. La technologie orienté service est utilisée dans les middlewares d'adaptation sous différentes formes comme les web services, les services REST, les services CORBA et les services ad'hoc. CAPPUCINO [108], VieDAM [92] et SAMProc [123] sont des middlewares basés web services. DoAmI [6] uniquement des services CORBA tandis que Gaia [107] et Aura [**?**] utilisent des services ad'hoc.

On rencontre aussi des solutions hybrides comme SCA [97]. CAPPUCINO est basé sur SCA tandis que WComp est basé sur une architecture comparable appelée SLCA [135] qui mélange SCA et AOP. Le tableau 3 présente une synthèse de ces approches.

4.5.2 *Découverte de ressources*

Pour les applications distribuées adaptatives dans lesquelles le code peut être distribué sur des périphériques distants à l'exécution et où les changements de position géographique de l'utilisateur déclenchent des changements de contexte d'exécution et des ressources de l'environnement, la découverte de

	Micro Kernel	AOP	Service-based
Aura			X
AxSel			X
BASE	X		
CAPPUCINO		X	X
DoAmI			X
Gaia	X		
Kalimucho	X		
Mundo Core	X		
MUSIC	X	X	X
RCSM	X		
SAMProc			X
VieDAME			X
WComp		X	X

TABLE 3 – Comparaison de middlewares d'adaptation

ressources devient nécessaire. La découverte de ressources inclut la découverte de ressources matérielle et la découverte de services (cf. figure.18).

Les services de découverte de matériel sont essentiellement utilisés par le wifi et le bluetooth (SDP – Service Discovery Protocole) pour découvrir les ressources matérielles de l'environnement.

La découverte de service utilise le matériel environnant et internet pour découvrir les services. Quand un utilisateur change de lieu géographique, on découvrira les services fournis par le matériel environnant ou on utilisera ce matériel pour se connecter à internet pour réaliser cette découverte. C'est pourquoi, le service de découverte utilise un service de localisation de l'utilisateur. Pour ce faire, il peut utiliser ou combiner plusieurs technologies (wifi, GPS, 3G, etc.) pour déterminer la position de l'utilisateur. On parle de découverte basée sur la localisation. D'autre part, le service de découverte exploite internet pour découvrir des services web, on parle alors de découverte orientée web.

4.5.3 *Adaptation Orientée Interactions Homme-Machine*

Les déplacements de l'utilisateur conduisent à des variations de contexte durant l'exécution de l'application. C'est pourquoi, la plateforme doit fournir un service d'IHM qui découple la logique métier et l'interface. Dans l'objectif de permettre la migration de l'application sur différents périphériques, le service d'IHM doit utiliser des technologies indépendantes du périphérique et de l'OS. La technologie actuellement la plus utilisée dans ce but est HTML.

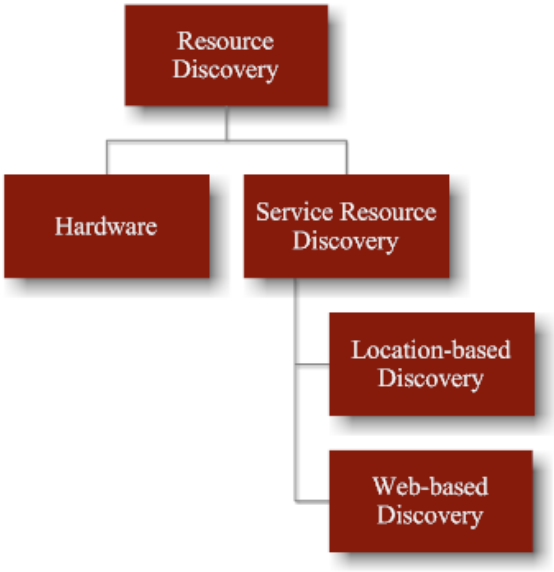

FIGURE 18 – Technologies de découverte de ressources

4.5.4 *Stockage de données*

Les applications ont besoin de stocker et de partager leurs données ainsi que de pouvoir faire des recherches dans les données stockées. Dans un environnement distribué, comment garantir qu'une application puisse rapidement accéder aux données à tout moment et en tout lieu ? Généralement, il y a trois approches possibles : les systèmes de fichiers, les bases de données et le stockage sur le cloud. Les systèmes de fichiers sont utilisés dans Gaia [107]. Les bases de données sont une bonne solution pour différentes raisons en particulier parce qu'elles incluent des mécanismes d'accès et de sécurité. Au contraire, le stockage sur le cloud nécessite un accès internet et la sécurité des données reste un problème. Toutefois, il peut être adapté pour les plateformes auto-adaptatives (cf fig.19).

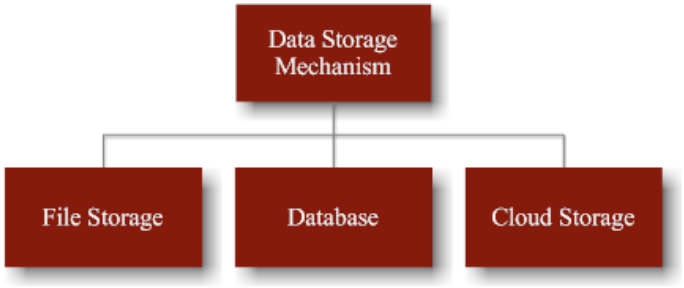

FIGURE 19 – Mécanismes de stockage des données

Nous avons vu les quatre parties composant un middleware d'adaptation. Nous allons maintenant nous intéresser aux plateformes d'adaptation et à leurs techniques.

5

PLATEFORMES D'ADAPTATION

C'est le cœur du système d'adaptation. Elles peuvent proposer plusieurs types d'autonomies allant d'un fonctionnement totalement automatique jusqu'à un fonctionnement contrôlé par interventions humaines [98]. Elles peuvent être distribuées ou centralisées. Elles prennent en charge l'analyse et la prise de décision d'adaptation en fournissant une analyse intelligente, des heuristiques de planifications et des services de décision. En accord avec leurs définition et leurs rôle, les plateformes d'adaptation sont composées de deux parties qui sont l'analyse pour l'adaptation et la décision d'adaptation (cf. fig.20).

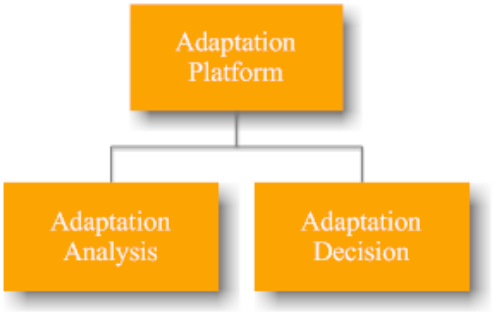

FIGURE 20 – Plateformes d'adaptation

Nous allons présenter les techniques utilisées dans l'analyse pour l'adaptation dans la partie 5.1 et celles pour la décision d'adaptation dans la partie 5.2.

5.1 ANALYSE POUR L'ADAPTATION (RAISONNEMENT SUR LES SITUATIONS)

Quelle est la situation courante ? C'est l'information essentielle que nous devons connaître et qui nous guidera pour prendre une décision. Avant de prendre une décision, le cerveau humain analyse l'environnement, les objectifs, etc. On parle d'analyse de problème qui consiste en l'identification de la situation. Ce processus aide notre cerveau à prendre une décision [76]. La plateforme d'adaptation simule ce processus pour permettre au système de prendre des décisions d'adaptation. Le raisonnement sur les situations est un domaine de recherche très vaste. Dans des applications sensibles au contexte, les chercheurs définissent une situation comme une interprétation sémantique externe des données des capteurs [47]. Conformément à cette définition, il y a deux catégories de techniques : celles basées sur les spécifications et celles basées sur l'apprentissage (cf. fig.21). Ces méthodes permettent de construire des modèles de situations qui pourront être utilisés par le système d'adaptation (cf. fig.22).

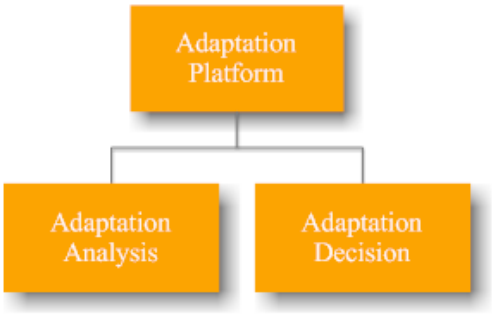

FIGURE 21 – Analyse pour l'adaptation

5.1.1 *Techniques basées sur les spécifications*

Autrefois, les relations entre les informations de contexte brut et les situations étaient faciles à établir parce qu'il y avait peu de capteurs. Les approches basées sur les spécifications sont essentiellement des techniques d'identification de situation. Il existe plusieurs approches basées sur des règles logiques comme les modèles de logiques formelles [82] et les modèles de logique spatio-temporelles [39] qui permettent des raisonnements efficaces. Elles ont été largement appliquées en raison de leurs puissantes possibilités de raisonnement basées sur leurs ontologies de représentation. Récemment, les raisonnements à base d'ontologies ont été intensément utilisés. La logique formelle coordonnée à des ontologies de modèles de contexte permet de fournir un vocabulaire de concepts standards et des relations sémantiques aboutissant à l'inférencee de situations. Pour tenir compte de l'incertitude, les techniques habituelles basées sur la logique doivent être associées à des techniques probabilistes [62] ou à de la logique floue et la théorie de Dempster-Shafer [5, 71, 88].

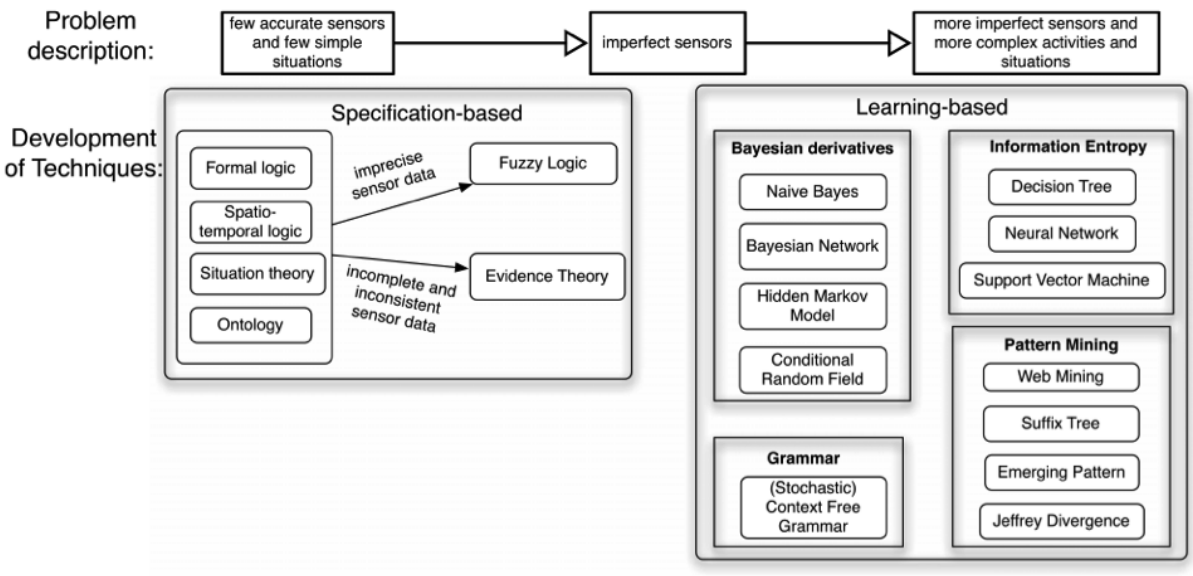

FIGURE 22 – Accroissement de la complexité de la situation principale [150]

5.1.2 *Techniques par apprentissage*

Il y a quatre familles de techniques par apprentissage, les modèles Bayesiens, les modèles à base de grammaires, les modèles entropiques et les fouilles de données. Les modèles Bayesiens sont très connus et comportent deux types d'approches. L'un est le modèle de codage de relations causales comme Naïve Bayes [101, 131] et les réseaux Bayesiens [58]. L'autre est un modèle d'encodage de relations temporelles comme Dynamic Bayesian Network [139], Heiden Markov Models [147, 66] et Conditional Random Fields [137, 140]. Les approches sur les grammaires sont utilisées pour représenter la sémantique structurelle complexe des processus dans des situations hiérarchiques [91, 113, 136]. Les arbres de décision [15], les réseaux de neurones [149], les machines à vecteur de support (ou SVM – Support Vector Machine) [100, 73] sont construits sur l'entropie de l'information. En tout état de cause, les techniques par apprentissage actuelles nécessitent de grandes quantités de données de tests pour construire un modèle et en estimer les paramètres [134].

5.2 DÉCISION D'ADAPTATION

La décision d'adaptation est généralement divisée en approches statiques, dynamiques et par intelligence artificielle. L'approche statique nécessite de prédéfinir et implémenter les logiques d'adaptation. Du fait que différentes applications ont différentes préoccupations, les logiques d'auto-adaptation sont aussi très différentes. Les middlewares d'adaptation peuvent ne pas être capables de prédéfinir les logiques d'adaptation pour toutes les applications possibles. Selon le type des middlewares, choisir une façon statique de générer la logique d'adaptation suppose souvent que l'application gère elle-même cette logique d'adaptation. Cela signifie que le concepteur de l'application doit créer pour chaque application la logique d'adaptation. Le middleware fournit seulement la connaissance du contexte et les mécanismes d'adaptation. Selon la littérature, il y a six approches différentes qui sont : 1) la logique d'adaptation basée sur les algorithmes, 2) la logique d'adaptation basée sur des politiques d'adaptation, 3) la logique d'adaptation basée sur la planification, 4) la logique d'adaptation basée sur des automates, 5) la logique d'adaptation basée sur des graphes, 6) la logique d'adaptation basée sur de l'intelligence artificielle (cf. fig.23).

La logique basée sur les algorithmes est réalisée par une chaîne d'algorithmes. Après exécution, l'algorithme fournit une solution de décision ou un plan d'adaptation dans un temps donné. L'algorithme peut être changé dynamiquement au runtime. Le framework Dynaco offre ce type d'adaptations dynamiques. André et al proposent un algorithme pour faire de l'adaptation maître/esclave dynamiquement à partir du framework Dynamico. Cet algorithme est basé sur la description de patrons de comportement et dépend de l'état du patron et du système distribué dans un objectif de qualité de service [7].

La logique d'adaptation basée sur des politiques d'adaptation fournit un guide pour la décision et les actions. Les services de gestion de la politique sont constitués d'un dépôt de politiques, d'un ensemble de points de décision pour interpréter les politiques et d'un ensemble de points d'application de politiques [145].

Kephart et Walsh abordent différents types de politiques qui peuvent être exploités dans des systèmes d'adaptation comme la politique de but, la politique d'action (dans la forme ECA) et la politique d'utilité

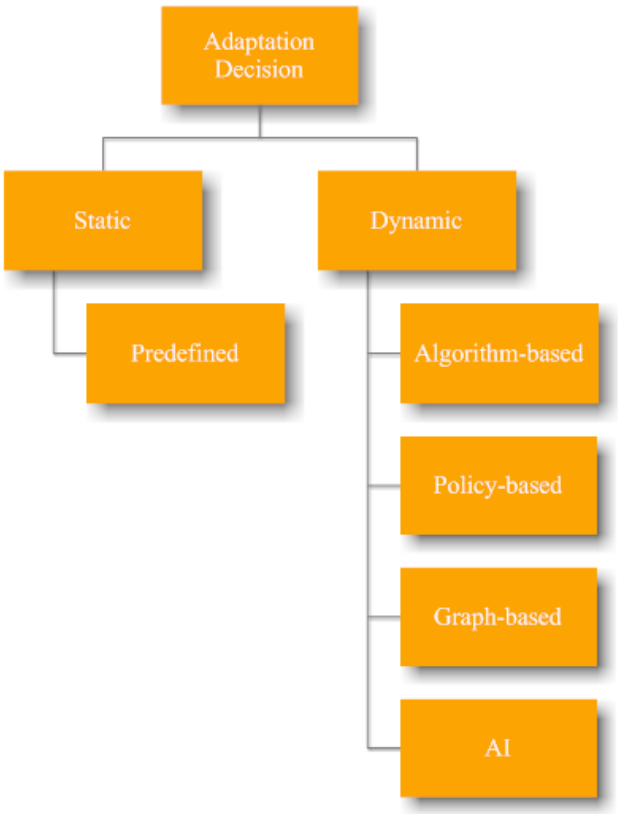

FIGURE 23 – Décision d'adaptation

[75]. En raison des changements dynamiques de situations dans l'informatique pervasive, la politique doit aussi changer dynamiquement. Donc, Lutfyyia et al ont proposé une technique de contrôle conjectural pour couvrir les besoins des systèmes d'adaptation. Chi et al ont proposé une technique de gestion de politiques basée sur des règles ECA pour leur système d'adaptation UbiStart qui accepte des changements au runtime et des mises à jour en ligne [124]. Il existe de nombreux systèmes d'adaptation qui adoptent la gestion par politiques pour la décision et la planification de l'adaptation comme par exemple [74, 12, 116].

La logique d'adaptation basée sur la planification. En 1995, Russel et Norvige indiquent qu'un moteur d'adaptation basé sur la planification doit recourir à de la planification continue plutôt qu'à de la planification circonstancielle ou à de la re-planification [112]. La planification dans les systèmes d'adaptation présente deux aspects : la planification d'observation et la planification d'adaptation [98]. Un planificateur d'observation décide quand et où adapter en fonction des informations observées (informations de contexte de haut niveau). Un planificateur d'adaptation détermine quel plan d'adaptation doit être mis en œuvre et quand il peut l'être en fonction de la situation identifiée. « it refers to the reconfiguration capability of an application to changing operating conditions by exploiting knowledge about its composition and quality of service (QoS) metadata associated to its constituting services [54]. MADAM est un middleware base sur la planification [49]. Le projet MUSIC a adopté ce middleware et l'a étendu.

La logique d'adaptation basée sur les graphes utilise des graphes mathématiques pour résoudre des problèmes de dépendances de déploiement de composants qui permettent de décider quels nœuds peuvent être chargés sur un périphérique en fonction des ressources disponibles. Par exemple, AxSeL [65] est un middleware d'adaptation basé sur les graphes. Une application d'AxSeL est représentée par un graphe

général bidimensionnel et flexible de dépendance où les nœuds représentent les services et les composants. AxSeL utilise un algorithme de coloration du graphe de dépendances avec des informations contextuelles sur les nœuds du graphe pour procéder à une décision de déploiement sous certaines contraintes.

La logique d'adaptation basée sur de l'intelligence artificielle utilise une technologie par apprentissage. Elle peut être utile dans un processus de décision d'adaptation en raison de ses possibilités de planification riche et de ses possibilités de raisonnement par apprentissage. Toutefois, les systèmes actuels basés sur de l'intelligence artificielle rencontrent quelques problèmes comme celui de la garantie de la qualité. Néanmoins, quelques approches sont d'ores et déjà utilisées dans des systèmes d'adaptation comme l'algorithme A*, les heuristiques, la classification naïve bayésienne, les réseaux bayesiens, etc. Par exemple, CADeComp [11] utilise un algorithme A* pour réduire la complexité du problème de placement des composants. Cet algorithme permet d'approcher une solution optimale. Arsahd et al utilisent un système auto-réparable [9]. Maes a proposé un modèle basé sur les objectifs pour la sélection des actions [84]. Tesauro et al [133], Weyns et al [146] travaillent sur système multi-agents constitués d'agents intelligents. Certains systèmes d'adaptation adoptent également des techniques comme les algorithmes génétiques et les algorithmes d'apprentissage.

Certains chercheurs insistent sur l'intérêt de l'apprentissage par renforcement (Reinforcement Learning - RL). Amoui et al considèrent que le RL peut être une option prometteuse pour le choix dynamique des actions [4]. Dowling pense que ça peut être utilisé pour décentraliser les logiciels collaboratifs d'auto-adaptation [48]. Tesauro pense que le RL a le potentiel pour obtenir de meilleures performances par rapport aux méthodes traditionnelles tout en requérant moins de connaissances du domaine [132]. Il existe plusieurs méthodes adoptées par les systèmes d'adaptation pour parvenir à la décision et à la planification d'adaptation. Par exemple, la théorie de la décision, la théorie de l'utilité, les processus de décision Markoviens et les réseaux Bayesiens.

5.3 SYNTHÈSE

La décision d'adaptation est la partie la plus importante du système. Elle traite du problème des plans d'adaptation, de l'observation, du placement des composants et de la mise en œuvre de la décision d'adaptation. Elle a besoin d'une analyse de l'adaptation pour identifier la situation actuelle et décider de l'adaptation en fonction de cette situation (cf. tableau.4).

	Static	Algorithm-based	Policy-based	Planning-based	Graph-based	A.I.-based
AxSel					X	
CADeComp						X
CAPUCCINO			X			
Dynaco framework		X				
GAIA			X			
Kalimucho	X		X			
MUSIC				X		
UbiStar			X			
WComp	X		X			

TABLE 4 – Synthèse des technologies pour la prise de décision d'adaptation

6

MODÈLES D'APPLICATIONS

On distingue trois catégories de modèles d'applications : ceux basés composants, ceux basés services et les modèles hybrides (cf fig.24). La section 6.1 présentera les modèles basés composants, la section 6.2 ceux basés services et la section 6.3 les modèles hybrides.

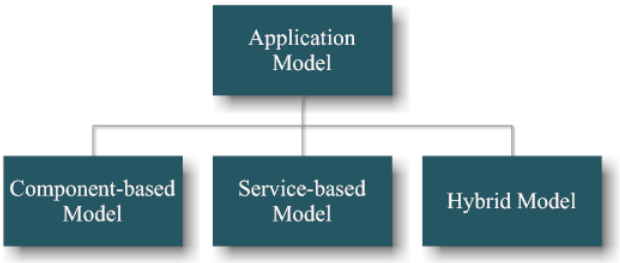

FIGURE 24 – Modèles d'application

6.1 LES MODÈLES BASÉS COMPOSANTS

Szyperski et al. [130] définissent le concept de composants comme : "a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". Un composant peut être constitué d'autres composants au travers d'interfaces bien définies qui spécifient ce qu'il requiert et fournit. Une configuration logicielle est une composition de composants logiciels. Dans l'informatique pervasive, adapter un logiciel basé composants signifie changer la configuration logicielle. Ce modèle d'applications fournit plusieurs avantages comme la réutilisabilité, la configurabilité dynamique et le déploiement dynamique. C'est pourquoi c'est un modèle très répandu dans les logiciels pervasifs. Selon la technique de réflexion, il y a deux types de modèles composants (cf. fig.25)

Réflexion par conteneurs. Les composants sont inclus dans des conteneurs qui les supervisent. Le composant encapsule une logique métier, et le conteneur fournit les propriétés fonctionnelles. Ce modèle peut offrir un couplage faible entre la logique métier et les propriétés non fonctionnelles comme par exemple le contrôle de la qualité de service et le contrôle des communications. Le modèle Osagaia de Kalimucho [27] est un modèle classique à base de conteneurs (cf. fig.26).

Réflexion par composant. Il y a un conteneur unique pour tous les composants du système. Ce conteneur supervise les composants à travers une technique de réflexion. L'application est constituée par une

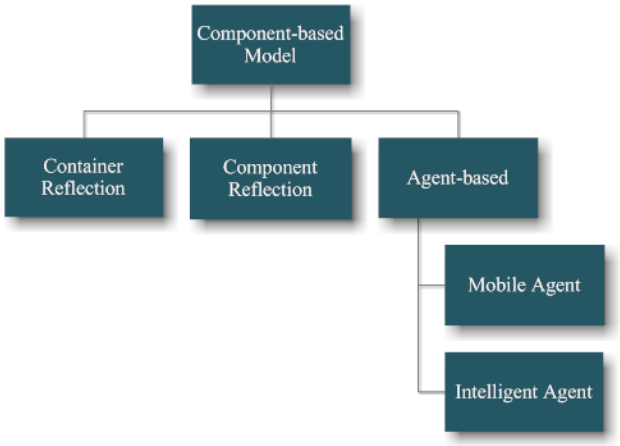

FIGURE 25 – Technologies pour les modèles basés composants

composition de composants et les propriétés non fonctionnelles sont gérées globalement par le système. EJB [128] et .NET [89] sont des approches classiques de plateforme à réflexion par composants.

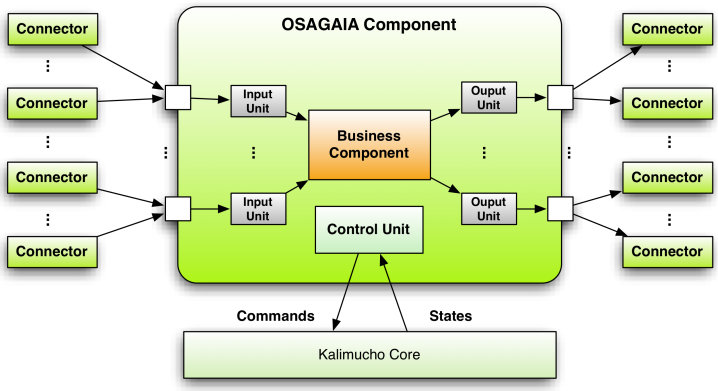

FIGURE 26 – Modèle de composants OSAGAIA

Modèles basés agents. C'est un modèle de composants basé sur la théorie des agents. Russell et Norving définissent un agent comme "anything that can be viewed as perceiving its environment threw sensors and acting upon that environment threw effectors". Selon cette définition, un agent intelligent est un agent qui a un comportement d'intelligence artificielle et qui peut interagir de façon autonome avec son environnement. Un agent mobile est un agent qui a la possibilité de se transporter lui-même d'un système à une autre au travers d'un réseau [79]. Ces modèles basés agents rencontrent un obstacle commun qui est la garantie de qualité. Il existe plusieurs projets basés agents pour l'ubiquité comme MDAgent [152], SpatialAgent [118], UbiMas [13] et G-net [148].

6.2 LES MODÈLES BASÉS SERVICES

Un service dans le domaine des technologies de l'information est une entité (un composant, une application) fournissant des fonctionnalités qui sont utilisées par d'autres entités du système ou d'autres systèmes et même par des êtres humains. Le modèle orienté service met l'accent sur l'interface. Il met en

œuvre un couplage faible entre les interfaces et les implémentations du service. Il existe plusieurs projets qui utilisent un modèle de service ad'hoc comme SOCAM, Gaia et Aura [**?**]. Les services web sont les modèles de services les plus répandus actuellement. Le standard W2C utilise WSDL (Web Service Description Language) comme langage de description et SOAP (Simple Object Access Protocol) pour communiquer. Ce type de modèles permet de changer dynamiquement l'implémentation des services au runtime (cf. fig.27)

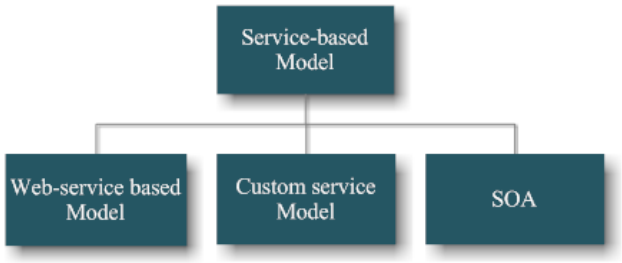

FIGURE 27 – Modèles basés services

Les chercheurs proposent plusieurs processus orientés services autonomes. Ainsi, Verma et Sheth [141] proposent les "autonomic web processes" (AWP) qui sont des processus orientés web services tandis que SAMProc est un middleware pour des processus mobiles auto-adaptatifs [123]. Ce middleware permet à l'application de changer de lieu et de comportement pendant son cycle de vie. Il utilise un langage de description de type BPEL pour décrire le modèle de web service appellé SAM.

6.3 LES MODÈLES HYBRIDES

L'approche hybride propose un modèle basé composants pour fournir des services. C'est un bon moyen de construire des systèmes d'adaptation compatibles avec tous les types d'applications. Cela signifie que le système est basé sur un modèle de service tandis que l'implémentation des services est basée sur un modèle de composants. Les modèles basés composants peuvent fournir une granularité d'adaptation fine et les modèles basés services peuvent fournir une bonne hétérogénéité et scalabilité. Le modèle SLCA de WComp [135] et le module Plastic [10] utilisent de telles approches.

6.4 SYNTHÈSE

Entre les niveaux application et plateforme, peuvent être utilisés différents modèles d'application, par exemple, la plateforme MUSIC est construite par un modèle de composant OSGi et fournit aux applications des services SOA. Ainsi, les applications peuvent être construites sur différents modèles (cf. tableau.5)

La section suivante s'intéressé à la dernière branche de la taxonomie qui est l'ingénierie logicielle. Il présentera différentes techniques utilisées aujourd'hui pour développer des systèmes d'adaptation.

	Component-based	Service-based	Agent-based	Hybrid
Aura		X		
AxSel	X			
CAPUCCINO	X	X		X
DoAmI		X		
G-net			X	
GAIA	X			
Kalimucho	X			
MAS			X	
MDAgent			X	
MUSIC	X	X		X
Plastic model				X
SAMProc		X		
SOCAM		X		
SpatialAgent			X	
UbiMAS			X	
VieDAME		X		
WComp	X	X		X

TABLE 5 – Comparaison des modèles d'application

7

INGÉNIERIE LOGICIELLE

Les systèmes d'adaptation peuvent être implémentés par différentes techniques d'ingénierie logicielle. La programmation par objets est la technique la plus populaire d'ingénierie logicielle. Les patrons de conception GoF sont une méthodologie générale pour la programmation par objet. Les programmations orientées composants et orientées service sont les techniques de programmation les plus répandues actuellement. Elles fournissent plusieurs caractéristiques avancées pour la réutilisation de logiciels, la scalabilité et l'extensibilité. L'ingénierie dirigée par les modèles (IDM) fait référence à l'utilisation systématique de modèles comme premiers artefact pour l'ingénierie des systèmes [52] (cf. fig.28)

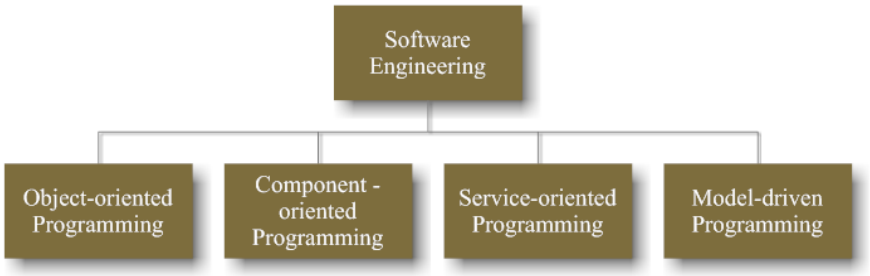

FIGURE 28 – Ingénierie Logicielle

7.1 PROGRAMMATION PAR OBJETS

La naissance de la programmation par objets peut être située en 1960. Il s'agit d'un paradigme de base de la programmation. La programmation par objets associée à la conception par patrons permet d'obtenir des solutions flexibles et réutilisables pour le développement de systèmes adaptables. La conception par patrons est une méthode générale d'ingénierie logicielle pour représenter des solutions bien connues à des problèmes classiques et récurrents dans la conception de logiciels [121].

7.2 PROGRAMMATION ORIENTÉE COMPOSANTS

La programmation orientée composants est également appelée ingénierie logicielle basée composants. Elle peut être associée à une conception par modèles de composants pour implémenter des systèmes adaptables. C'est un paradigme de programmation bien connu qui peut être exploité pour développer des

systèmes logiciels flexibles et extensibles [130]. Un système à base de composants peut s'adapter lui-même en changeant la composition des composants. Comme nous l'avons dit précédemment, cette technique peut être couplée à SOA pour obtenir une meilleure hétérogénéité et flexibilité. OSGi est la plateforme dynamique orientée composants la plus connue.

7.3 PROGRAMMATION ORIENTÉ-ASPECTS

Dans l'informatique ubiquitaire, la technique AOP peut être utilisée pour encapsuler l'adaptation décrite par des aspects. A l'exécution, les aspects peuvent être dynamiquement tissés dans le code de l'application pour réaliser l'adaptation. AOP permet d'implémenter des actions d'adaptation à grain fin, à un niveau inférieur à celui des composants [57, 115]. AspectJ est le framework le plus populaire pour Java. JACK est un framework AOP dynamique qui utilise des mandataires pour dynamiquement modifier ou ajouter des points de jonction [102].

7.4 PROGRAMMATION ORIENTÉE SERVICE (SOP)

SOP fonctionne avec un modèle de conception basé sur les services ou une de ses variantes pour réaliser le système d'adaptation. Actuellement, les chercheurs utilisent cette technologie pour permettre de découvrir des services et d'interagir avec eux en fonction de leur localisation. Elle est efficace pour des services utilisateurs interagissant pour des applications ubiquitaires ou des services ayant des besoins spécifiques de localisation ou de temps [22]. SOP peut être implémenté en Java avec le framework JINI [129], avec le framework .NET [89], par des web services, etc.

De plus, la technologie des web services offre la flexibilité de composition, d'orchestration et de chorégraphie [103].

7.5 MODEL DRIVEN DEPLOYMENT (MDD)

Schmidt définit l'ingénierie définie par les modèles comme "a promising approach to address platform complexity – and the inability of surge-generation language to alleviate this complexity and express domain concepts effectively – is to develop model-driven engineering (MDE) technologies". MDE fournit un haut niveau d'abstraction du système qui peut directement être appliqué à diverses plateformes. Model Driven Architecture (MDA) est une approche MDE proposée par L'OMG [95]. Le modèle abstrait est indépendant de la plateforme, il est appelé PIM (Platform Independent Model). Le PIM peut être dérivé vers un ou plusieurs PSM (Platform Specific Model) qui sont liés à une plateforme spécifique.

Dans la partie suivante de ce chapitre, nous allons tenter de classifier les différentes approches énoncées précédemment.

8

CLASSIFICATION DES APPROCHES

Nous avons décrit précédemment les technologies actuelles de l'informatique ubiquitaire. Nous allons maintenant présenter quelques approches connues selon trois classifications différentes.

Dans ce but nous avons dressé les tableaux de classification des systèmes d'adaptation présentés en tableau 6, 7 et 8.

Il est à remarquer que l'auto-adaptation est hors de cette étude. Certaines approches, comme SamPROC, ne sont ni supervisées ni auto-adaptatives parce qu'elles ne constituent pas un système d'adaptation complet mais seulement un middleware d'adaptation.

Les systèmes d'adaptation peuvent également être classés selon le mode d'adaptation. Adaptation basée sur l'architecture, sur le paramétrage ou sur les aspects (cf. tableau.7). L'adaptation basée sur l'architecture s'intéresse au changements de structure au niveau des composants logiciels ou des services ainsi qu'à la topologie d'interconnexion de ces services ou composants.

L'adaptation basée sur les paramètres porte sur l'intérêt des politiques d'adaptation et des paramètres d'entrée pour configurer les logiciels principalement en termes de QoS. L'adaptation orientée aspect utilise le tissage dynamique de code pour modifier des parties du code source d'un système en cours d'exécution [37].

Les systèmes d'adaptation peuvent aussi être classifiés en fonction de leur middleware : context-middleware, adaptation-middleware et adaptation-platform (cf. tableau.8). Nous avons proposé cette classification parce que les projets actuels sont uniformément appelés middleware pour informatique ubiquitaire alors qu'ils sont parfois très différents. Les context-middleware ont été décrits en section 4.2 (page 23), les adaptation-middleware en section 4.5 (page 28) et les adaptation-platforms en chapitre 5 (page 33) (cf. tableau.8)

	Supervised	Centered	Distributed
Aura	X	X	
AxSel	X	X	
CADeComp	X		X
CAPUCCINO	X		X
DoAmI		X	
GAIA	X	X	
Kalimucho	X		X
MUSIC	X		X
SAMProc		X	
SOCAM	X	X	
VieDAME		X	
WComp	X	X	

TABLE 6 – Classification des plateformes d'adaptation I

	Architecture-based	Parametric-based	Aspect-oriented-based
Aura	X		
AxSel	X		
CADeComp	X		
CAPUCCINO	X	X	X
DoAmI		X	
GAIA	X	X	
Kalimucho	X		
MUSIC	X	X	X
SAMProc		X	
SOCAM		X	
VieDAME		X	
WComp	X	X	X

TABLE 7 – Classification des plateformes d'adaptation II

	Context middleware	Adaptation middleware	Adaptation platform
Aura	X	X	
AxSel		X	X
CoBrA	X		
COSMOS	X		
CADeComp	X	X	X
CAPUCCINO	X	X	X
DoAmI		X	
GAIA		X	X
Kalimucho	X	X	X
MUSIC	X	X	X
SAMProc		X	
SOCAM	X	X	
VieDAME			X
WComp		X	X

TABLE 8 – Classification des plateformes d'adaptation III

Deuxième partie

CONTRIBUTION

9

ONTOLOGIE DE CONTEXTE DE KALIMUCHO (KALICONTO)

Dans ce chapitre nous présentons notre modèle de contexte basé sur l'ontologie. Pour commencer, nous présenterons le noyau du modèle de cette ontologie à partir d'un scénario. Ensuite, nous détaillerons toutes les ontologies de domaine de KaliCOnto. Chaque section indique l'objectif visé avant de donner la description détaillée de l'ontologie avec ses sous domaines. Enfin, nous aborderons les ontologies d'application de KaliCOnto en commençant par l'architecture du conteneur d'ontologie dans un environnement dynamique et Mobile. Nous détaillerons ensuite ces ontologies d'application, leur composition et les réutilisations des ontologies de domaine.

9.1 OBJECTIFS

L'objectif principal est de construire un modèle de contexte pour supporter les fonctionnalités de notre plateforme d'adaptation sensible au contexte, les applications utilisant la plateforme et les raisonnements. Un modèle de contexte destiné à la plateforme d'adaptation doit contenir l'état contextuel d'utilisation : environnement physique, environnement informatique, contexte des utilisateurs. Il doit être extensible par ces applications. La plateforme d'adaptation l'utilise pour identifier les situations d'adaptation (cf. chapitre 12). Elle doit y trouver des connaissances de donnée contextuelle, de l'état d'exécution de la plateforme, etc. Ces connaissances aident aux déploiements de la plateforme et de l'application.

C. Bettini et al. [21] identifient sept exigences pour un modèle de contexte qui sont : hétérogénéité et mobilité, relation et dépendance, pertinence du temps, imperfection, raisonnement, utilisabilité des formalismes de modélisation, efficacité d'approvisionnement du contexte. Les ontologies répondent parfaitement à ce type d'exigences (cf. tableau1 dans l'état de l'art, page 27).

Elles offrent des qualités comme : l'interopérabilité, la présentation sémantique et la possibilité de raisonnement logique. M. Poveda-Vaillaon et al. [105] indiquent qu'un modèle de contexte basé sur des ontologies permet de représenter des connaissances de contextes complexes et de fournir une sémantique formelle à la connaissance du contexte ainsi que de prendre en charge le partage et l'intégration des informations de contexte.

C'est pour toutes ces raisons que nous avons choisi de construire notre modèle de contexte sur des ontologies. L'implémentation de ce modèle de contexte utilise le langage "Web Ontology Langage" (OWL) [1]

1. OWL : http ://www.w3.org/TR/owl-features/

qui est recommandée par W3C [2]. Dans les sections suivantes nous présentons en détail la construction du modèle de contexte : KaliCOnto.

9.2 ONTOLOGIE DE NOYAU DU MODÈLE DE CONTEXTE : KALICONTO

Dans le figure 29 nous rappelons l'architecture de la plateforme Kalimucho-A, et la positionnement du KaliCOnto dans la plateforme. Nous présentons la conception de ce modèle de contexte dans cette section.

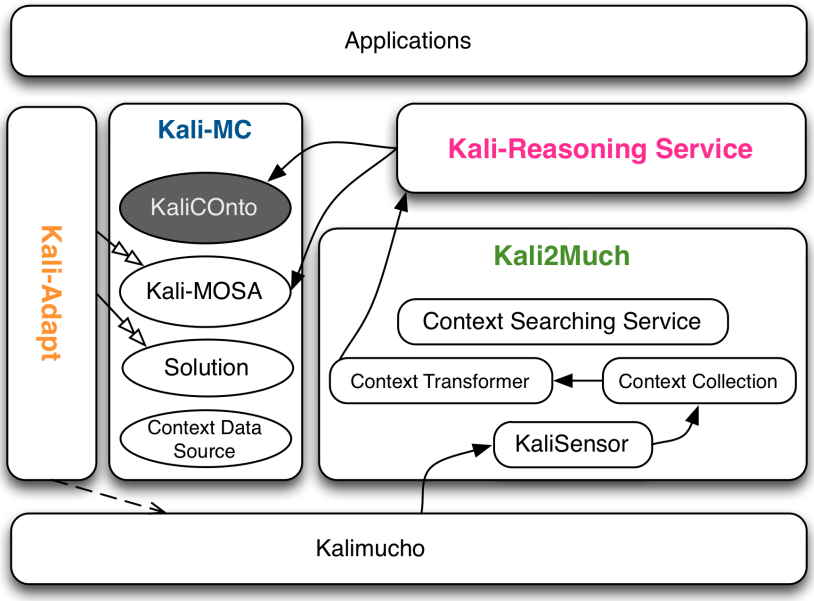

FIGURE 29 – Positionnement de "KaliCOnto" dans la plateforme

9.2.1 *Scénario dans un environnement sensible au contexte (context-aware)*

Considérons, à titre de scénario d'exemple, un utilisateur qui utilise des ressources informatiques pour ses activités quotidiennes dans un environnement sensible au contexte.

Olivier est enseignant à l'université. Le matin avant de partir au travail, Olivier utilise sa tablette pour consulter les actualités et communiquer avec sa famille à distance par vidéo chat. Quand il arrive à l'université, il utilise son PC de son bureau pour ses activités de recherche. Il a accès à certains services de l'université pour travailler (accès aux ressources de calcul de l'université). L'après-midi, il a une conférence dans une autre université. Il prend la voiture de fonction pour s'y rendre et utilise l'ordinateur intégré à la voiture pour la navigation. Sur le lieu de la conférence les participants peuvent se connecter à un vidéo projecteur pour leurs présentations.

Ce scénario montre que les changements d'activité ou de géolocalisation de l'utilisateur vont causer des changements de contexte. Quand il se déplace d'un lieu à l'autre, les environnements physiques autour

2. W3C : http ://www.w3.org/TR/2012/REC-owl2-overview-20121211/

de lui sont différents. Les ressources informatiques qui lui sont accessibles peuvent changer, par exemple en raison des connectivités réseau disponibles. Quand il passe d'une activité à une autre, les ressources informatiques changent également. Par exemple, dans le scénario, quand il est dans son bureau, les ressources de l'université lui sont accessibles pour ses activités professionnelles. Ces ressources matérielles ou logicielles sont différentes de celles dont il dispose quand il est chez lui.

Pour décrite les contextes d'un utilisateur dans un environnement sensible au contexte nous avons identifié les concepts suivants : "ContextEntity", "User", "ComputingResource" et "Environment" décomposé en "ComputingEnvironment" et "PhysicalEnvironment". Nous utilisons ces concepts comme "backbone" de notre ontologie du noyau du modèle. La section suivante présent ces concepts en détail et leur relations.

9.2.2 *Ontologie de noyau*

Dans cette section nous présentons notre conception de modèle de contexte en démarrant par le plus haut niveau d'abstraction. Nos objectifs sont :

— d'avoir un état du contexte actuel de l'utilisateur dans son domaine d'adaptation

— de permettre l'identification des situations d'adaptation

Notre ontologie de noyau du modèle (core ontology) est décrite par la figure 30 :

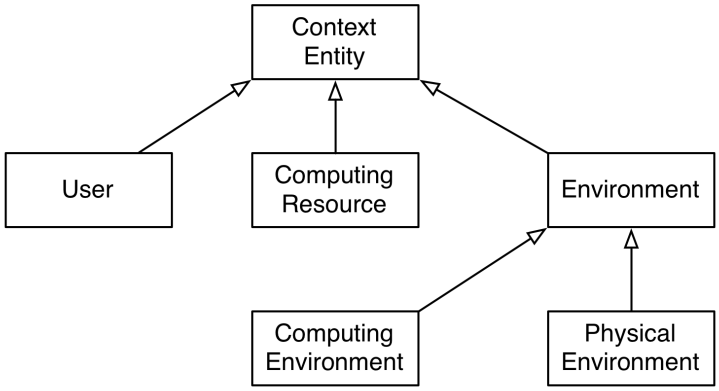

FIGURE 30 – Ontologie du noyau KaliCOnto

Cette ontologie décrit les éléments fondamentaux d'un domaine d'adaptation d'un utilisateur. Elle contient les concepts d'Entité de Contexte ("ContextEntity"), d'Utilisateur ("User"), d'Environnement ("PhysicalEnvironment", et "ComputingEnvironment"), et de Ressource de calcul ("ComputingResource"). Tous les éléments de ce modèle sont des sous catégories du concept "ContextEntity". Le centre de ce modèle est l'Utilisateur puisque son but est de décrire le contexte de l'utilisateur, l'environnement physique où il se situe et les ressources auxquelles il a accès.

Toutes ces informations contextuelles sont liées avec au temps et au lieu. Un utilisateur est toujours dans un environnement physique ("PhysicalEnvironment") et les "ComputingEnvironments" sont liés à un lieu. L'utilisateur a accès à certaines ressources dans un "Computing Environment". L'utilisateur peut

faire des activités qui sont supportées par des "ComputingResources" qui lui sont accessibles (ressources informatiques matérielles et logicielles).

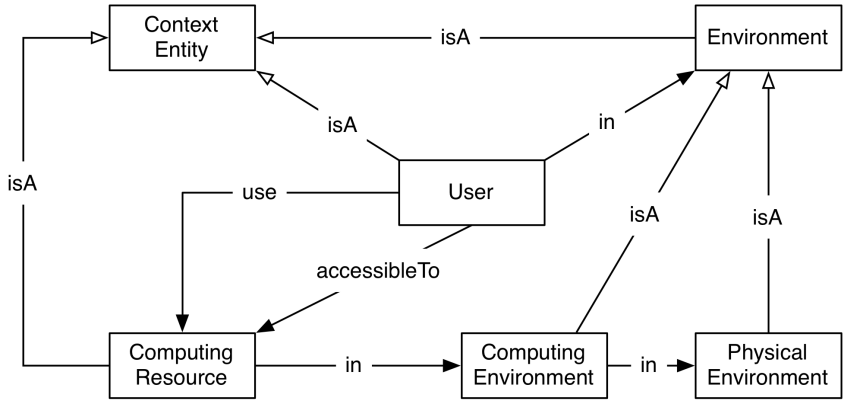

FIGURE 31 – Ontologie de noyau KaliCOnto : relations

Nous avons développé le modèle de contexte générique basé sur une ontologie appellé Kalimucho Context Ontology (KaliCOnto). Nous avons conçu ce modèle par réutilisation d'ontologies connues, par exemple : W3C Time Ontology (OWL-TIME) [70], SWEET Ontology (Unit) [46], MIO Ontology [105], FOAF [30], SOUPA [36], etc. C'est une multi-ontologie qui représente le niveau général du modèle de contexte KaliCOnto.

Nous allons maintenant présenter les ontologies de domaine du KaliCOnto, leurs objectifs et descrire chaque domaine.

9.3 ONTOLOGIES DE DOMAINE DE KALICONTO

Dans cette section nous présentons chaque domaine du modèle KaliCOnto selon l'ordre suivant : "ContextEntity", "User", "Environment" et "ComputingResource".

9.3.1 *Domaine "ContextEntity"*

9.3.1.1 *Objectif et analyse*

Cette ontologie décrit ce qu'est une information de contexte en s'intéressant à l'annotation sémantique, la valeur, l'unité, la représentation de données, la structure de données, etc. Il s'agit d'un méta-modèle de données contextuelles. Nous réutilisons le méta-modèle de contexte qui a été décrit dans la chapitre 10 (page 83) pour construire cette ontologie. L'objectif est de répondre aux questions suivantes :

Q1 Quelles entités de contexte sont représentées ?

Q2 Quelle est la structure de données ?

Q3 Quelle est la sémantique de chaque donnée contextuelle ?

Q4 Quelles sont les unités possibles pour une donnée contextuelle ?

Q5 Quelle est l'unité utilisée pour une donnée contextuelle ?

Q6 Quelles sont les représentations possibles pour une information de contexte donnée ?

Q7 Quelle est la représentation utilisée pour une information de contexte ?

Dans section suivante, nous présentons notre ontologie d'entité de contexte.

9.3.1.2 *Description*

Comme nous le présentons dans le chapitre 10, à un "ContextEntity" est associé un "ContextScope" (cf. figure.32) auquel est associée une "ScopeDescription". Cette "ScopeDescription" décrit les caractéristiques du "ContextScope" et possède une relation avec une "Representation" et avec une "DataStructure" (cf. figure.33). Une "ScopeDescription" peut avoir plusieurs "Représentations" mais une seule "DataStructure". Une "Representation" est associée à un "RepresentationTranslator" qui permet de traduire cette "Representation" en une autre. La "DataStructure" est composée de "ContextItem" ayant chacun un "ContextItem" et une "DataSignification" associés à une "Unit" et un "ContextValue".

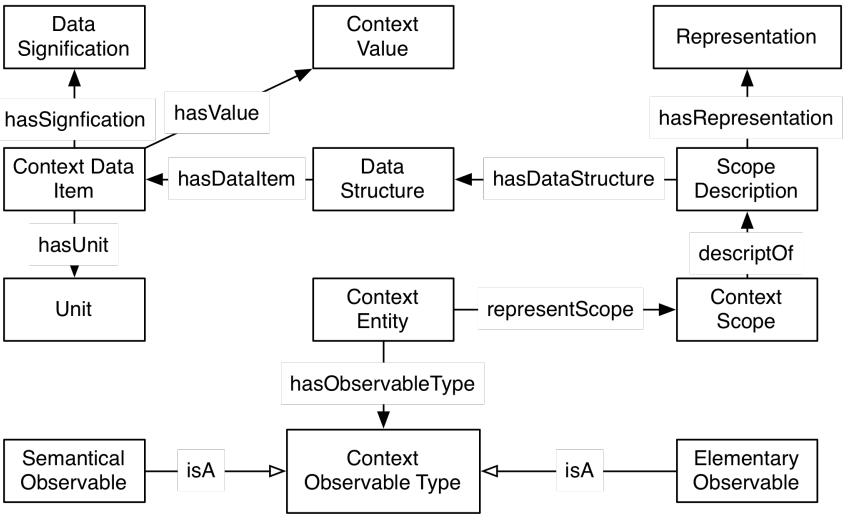

FIGURE 32 – Ontologie de description de "ContextEntity"

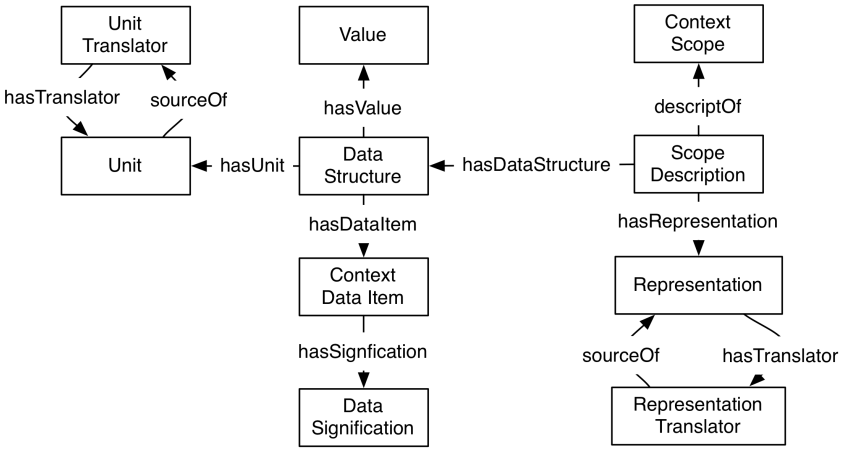

FIGURE 33 – Détail de l'ontologie de description de Contexte : partie "ContextScope"

Cette ontologie est utilisée pour décrire la structure des données contextuelles. Pour une donnée spécifique, nous pouvons utiliser sa description pour retrouver des connaissances associées à ce type de donnée. Par exemple, pour une donnée GPS, nous pouvons lancer une requête dans l'ontologie pour en récupérer les représentations possibles ainsi que pour trouver les traducteurs de représentation ("Representation-Translator") pouvant être utilisés.

Dans la section suivante, nous présentons le domaine "User", et l'ontologie "Kali-User Knowledge".

9.3.2 *Domaine "User"*

9.3.2.1 *Objectif et analyse*

Cette partie de l'ontologie décrit la situation actuelle de l'utilisateur et son profil. Ces informations sont étroitement liées à des informations à caractère spatio-temporel (lieu, temps, mobilité). La plateforme d'adaptation a besoin de savoir quels appareils l'utilisateur est en train d'utiliser. Nous avons identifié les questions de compétence suivantes :

- Q1 Qui est-ce ?
- Q2 Où est-t-il (elle) ?
- Q3 Est-il (elle) en mouvement ?
  - Q3.1 En quelle façon (voiture, à pieds, en vélo, etc.)
  - Q3.2 Avec quelle vitesse
  - Q3.3 En extérieur ou intérieur
  - Q3.4 Dans quelle direction
  - Q3.5 Quelle est sa destination ?
- Q4 Quel est (sont) les matériel qu'il (elle) utilise ?
  - Q4.1 Selon quelle modalité d'interaction ?

Ces quatre questions correspondent aux quatre sous domaines du modèle d'ontologie : Connaissance de l'utilisateur, Localisation de l'utilisateur, Mobilité de l'utilisateur et Matériel en interaction.

La connaissance de l'utilisateur se présente sous la forme d'un profil d'utilisateur, par exemple, son nom, son âge, ses comptes numériques. La localisation de l'utilisateur correspond à sa géolocalisation, ce peut être une coordonnée GPS collectée par son mobile ou un lieu nommé se trouvant dans l'ontologie "Place" de KaliCOnto (cf. section.9.3.3.2).

L'utilisateur peut se déplacer avec ses appareils mobiles. Cette information va être utilisée pour identifier des solutions optimisées en rapport avec la situation actuelle de l'utilisateur (plus de détails seront donnés dans le chapitre suivant). La mobilité de l'utilisateur peut être détectée par les appareils mobiles par GPS, par changement de réseaux wifi, etc.

L'utilisateur interagit également avec ses appareils. Il peut communiquer avec plusieurs dispositifs en même temps. Par exemple, il utilise son portable pour accéder à son serveur de stockage chez lui au travers d'Internet. Il peut aussi connecter ses appareils à distance. Notre système est intéressé par l'identification des appareils en interaction directe avec l'utilisateur. Ces informations vont aider la plateforme à identifier les situations de l'utilisateur et à trouver des solutions d'adaptation si nécessaire.

9.3.2.2 *Description*

Dans cette partie nous décrivons l'ontologie "User" appelée "Kali-User". La figure 34 montre la structure de cette ontologie et les relations importantes.

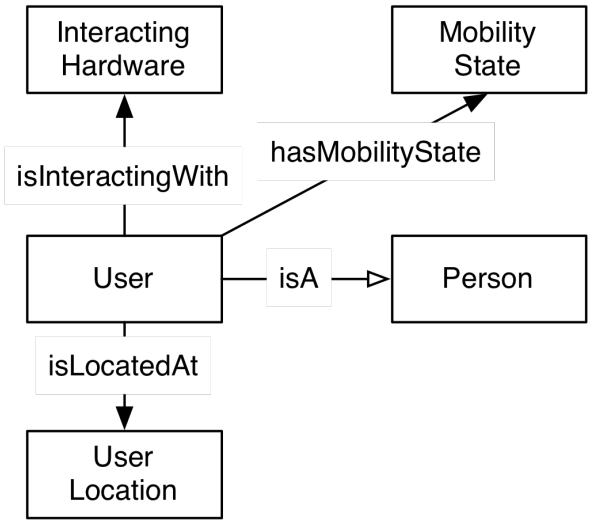

FIGURE 34 – Ontologie "Kali-User"

Cette ontologie contient quatre sous domaines (cf. figure.34) : Connaissance de l'utilisateur, Localisation de l'utilisateur, Mobilité de l'utilisateur et Matériel en interaction. L'utilisateur se trouve dans ("isLocatedAt") un lieu ("UserLocation"). Un lieu ("UserLocation") correspond à une géolocalisation ("isLocationOf") de l'utilisateur. L'utilisateur a ("hasMobilityState") un état de mobilité ("MobilityState"). Il peut interagir avec ("isInteractingWith") des matériels ("InteractingHardware"). Nous allons maintenant décrire ces quatre sous domaines.

Nous réutilisons l'ontologie de description d'utilisateur FOAF [30]. FOAF est une ontologie largement utilisée dans différents domaines. "FOAF intègre trois types de réseaux : les réseaux sociaux de collaboration humaine, d'amitié et d'association ; le réseaux de représentation qui décrivent une vue simplifiée d'un univers de bande dessinée sur le plan factuel et les réseaux d'information qui utilisent la liaison basée sur le Web pour partager les descriptions publiées indépendamment de cet inter-monde connecté.". La classe "User" dans notre ontologie est une sous classe (rdfs : subClassOf) de "foaf : Person".

La localisation de l'utilisateur ("UserLocation") est une sous classe (rdfs : subClassOf) de "Place" de KaliCOnto (cf. section.9.3.3.2). Les relations "isLocatedAt" et "isLocationOf" sont transitives. Par exemple, l'utilisateur est dans la salle de réunion numéro 23 et cette salle se situe dans l'IUT de Bayonne donc l'utilisateur est aussi à l'IUT de Bayonne.

La mobilité de l'utilisateur décrit un état contienant par défaut l'une des valeurs suivantes : "Mobile" et "Immobile". Quand l'utilisateur se déplace, son état de mobilité va changer (cf. figure.35). L'utilisateur a une vitesse de déplacement ("MovingSpeed"), une direction de déplacement ("MovingDirection"), une destination ("Destination") et un mode de déplacement ("WayOfMoving"). "MovingSpeed" est un "ContextScope" (cf. section. 9.3.1) qui contient un seul "ContextItem" qui est une vitesse. "MovingDirection" est un "ContextScope" qui contient une seule valeur entre 0 et 360 degrés nord. "WayOfMoving" est le mode de

déplacement de l'utilisateur, par exemple, en voiture, en vélo ou à pieds. "Destination" est le lieu "Place" où l'utilisateur va. Pour avoir toutes ces informations, nous avons conçu nos méchanismes de acquisition (10) et de raisonnement (11) pour enrichir l'ontologie KaliCOnto.

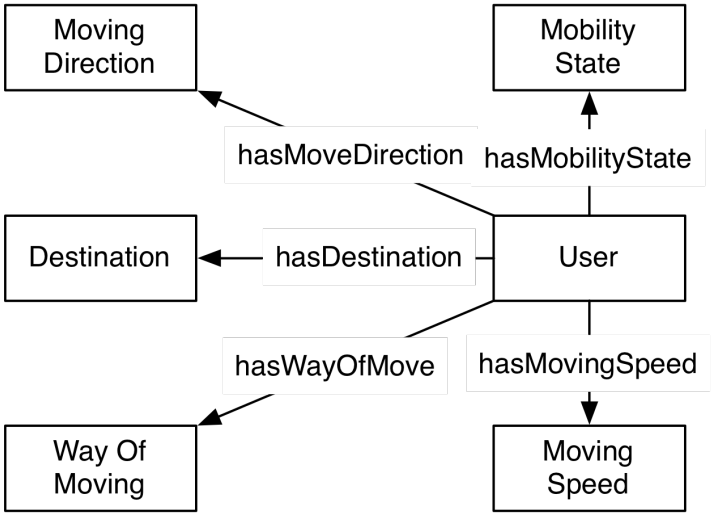

FIGURE 35 – Vue partielle de l'ontologie "Kali-User" : "MobilityState"

Nous avons appliqué le pattern "Objet avec état" [3] (cf. figure.36) pour construire l'état de mobilité. Le but de ce pattern est de permettre de modéliser les différents états d'un objet et les restrictions relatives à cet objet selon ses différents états.

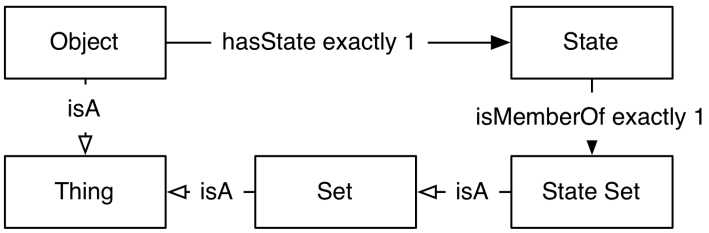

FIGURE 36 – Partie : "Object With State"

L'utilisateur n'a qu'un seul état de mobilité ("MobilityState") (cf. figure.37). Un état de mobilité doit correspond à l'une des valeurs suivantes : "Mobile" ("Walking", "DrivingVehicle", "RidingBike"), et "Immobile".

L'utilisateur interagit avec des dispositifs qui sont des "InteractingHardware". "InteractingHardware" est une sous classe de "Hardware" dans KaliCOnto (cf. section.9.3.4.3). Dans cette ontologie nous distingons deux relations avec les matériels en interaction (cf. figure.38) : l'utilisateur interagit avec ("isInteractingWith") des matériels au travers de ("isInteractingThrough") des modalités ("Modality"). Ces relations permettent de retrouver les périphériques actuellement en interaction avec l'utilisateur mais également de savoir comment ils interagissent avec l'utilisateur.

Dans cette section nous avons présenté l'une des ontologies de noyau de KaliCOnto : l'ontologie d'utilisateur qui contient les informations de l'utilisateur, sa géolocalisation, sa mobilité, et les matériels avec

3. Ontology design patterns : http ://ontologydesignpatterns.org/wiki/Submissions :Object_with_states

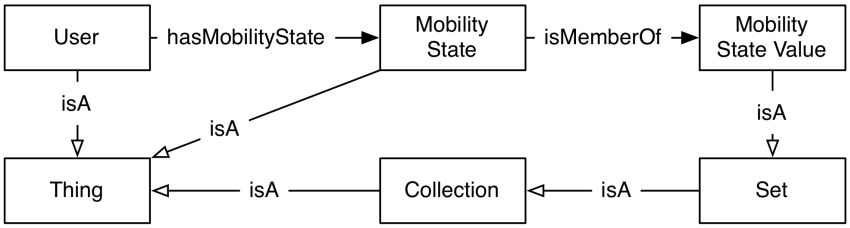

FIGURE 37 – Mobilité de l'utilisateur

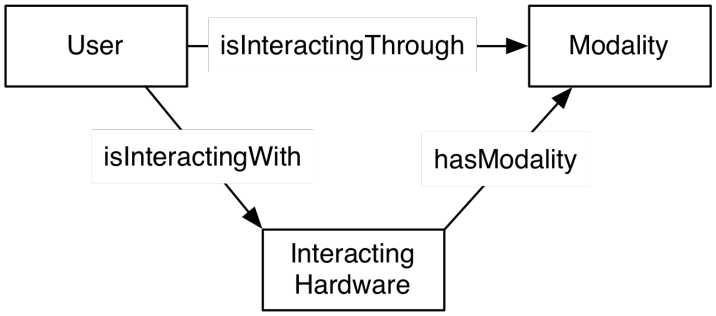

FIGURE 38 – Ontologie "Kali-User" : Matériels en interaction

lesquels il interagit. La partie suivante présente une autre ontologie de noyau de KaliCOnto : l'ontologie d'environnement ("Environment").

9.3.3 *Domaine "Environment"*

9.3.3.1 *Objectif et analyse*

Selon le dictionnaire français LAROUSSE [4], le mot "environnement" signifie : "Ensemble des éléments (biotiques ou abiotiques) qui entourent un individu ou une espèce et dont certains contribuent directement à subvenir à ses besoins."

Un utilisateur de notre plateforme est d'un individu qui est entouré par un ensemble d'éléments physiques et d'éléments informatiques. Le cadre de vie d'un utilisateur est constitué par les éléments physiques (lumière, géolocalisation, etc.) et les ressources informatiques (smartphone, PC, etc.) auxquels il peut accéder et qui lui rendent des services dans ses activités quotidiennes. L'ontologie d'environnement a pour but de décrire ces éléments et leurs relations.

Nous avons identifié l'ensemble de questions suivant pour nous aider à développer cette ontologie d'environnement :

Q1 Comment décrire l'environnement de l'utilisateur ?

Q2 Quelles sont les conditions environnementales actuelles de l'utilisateur ?

Q3 Quelles sont les ressources informatiques disponibles dans un lieu donné ?

4. http ://www.larousse.fr/

Q4 Quelles sont les ressources informatiques qui peuvent être accédées par l'utilisateur dans un lieu donné ?

Q5 Quels sont les environnements informatiques qui interagissent actuellement avec l'utilisateur ?

9.3.3.2 *Description*

L'environnement contient deux types d'environnements : l'environnement physique et l'environnement de calcul. L'environnement physique (EP) est réservé aux applications sensibles au contexte physique comme, par exemple, une application gérant une climatisation. Quand la température d'une salle est supérieure à 28 degrés, l'application active le dispositif de climatisation. Dans cette partie de l'ontologie, on doit répondre aux questions Q1 et Q2. Il existe déjà des ontologies sur cette thématique, par exemple (SWEET [46], ENVO [31], CoDAMos [106], mIO ! [105], etc.). Nous réutilisons une ontologie d'environnement dans un modèle de contexte pour environnement mobile. Cette ontologie est une partie du réseau d'ontologie mIO !. Elle modélise les connaissances de l'environnement physique et les conditions environnementales (humidité, luminosité, bruit, etc.) Dans le monde physique tous les contextes sont liés à un temps précis, un intervalle du temps et un lieu. Nous héritons de l'ontologie "OWL-Time" [5] qui a créée par Hobbs et Pan [70] pour décrire la notion du temps. Pour la localisation, nous réutilisons l'ontologie "GeoRSS-Simple" [6] pour constituer notre ontologie de lieu ("Place"). "GeoRSS-Simple" nous permet de décrire les géométries de base – "gml :Point", "gml :Polygon", "gml :LigneString", "gml :LinearRing" et "gml :Envelope". Elle nous permet également d'ajouter des annotations sémantiques en utilisant la classe "gml :_Feature" (cf. figure.39). L'ontologie "Place", contient deux classes : "Place" et "NamedPlace" qui héritent de "gml :_ Feature". "Place" est une localisation simple, c'est une "gml :_Feature" qui situe dans un "gml : _Geometry" ("box", "point", "polygon", etc.). "NamedPlace" est une "gml : _Feature" qui doit avoir un "gml :featurename" et un "gml :featuretypetag".

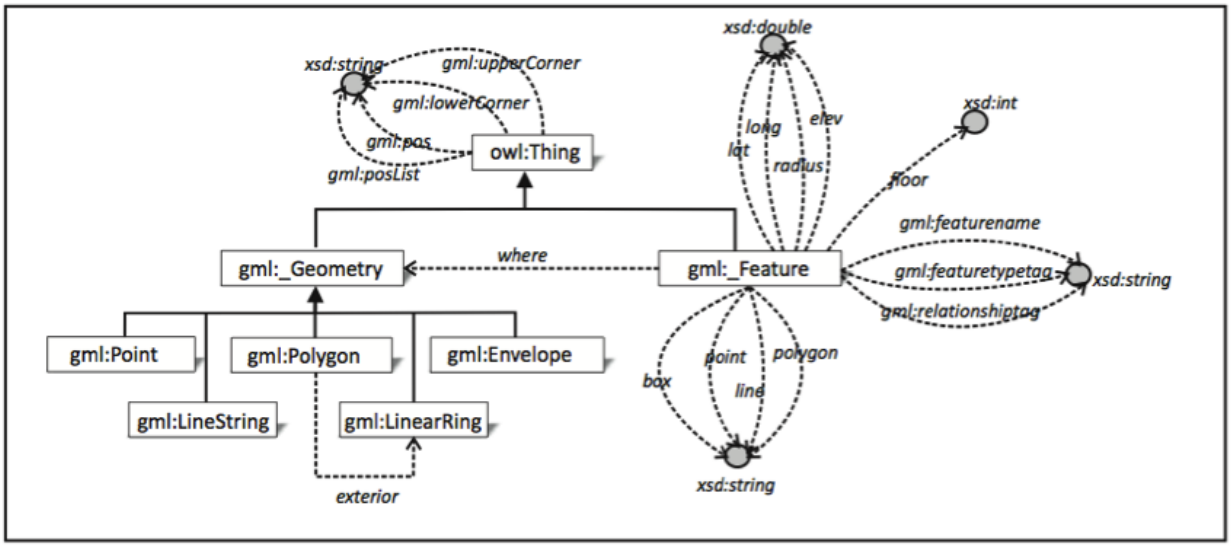

FIGURE 39 – Ontologie "GeoRSS-Simple" [90]

5. OWL-Time : http ://www.w3.org/TR/owl-time/
6. GeoRSS-Simple : http ://www.georss.org/

L'ontologie d'environnement est conçue pour les applications sensibles au contexte qui peuvent l'étendre selon leurs besoins.

L'Environnement de calcul (EC "ComputingEnvironment") représente un lieu spécifique contenant des ressources pour exécuter des applications. Par exemple, un lieu désigné par "AtHome" contiendra des ressources pouvant être partagées par les membres d'une famille ou leurs invités pour tous types d'activités. Tandis qu'un lieu désigné par "AtWorkingPlace" représentera des ressources auxquelles l'utilisateur a accès dans le cadre de ses activités professionnelles.

Chaque environnement peut offrir des informations sur ses ressources dans l'ontologie. Cela veut dire que quand l'utilisateur entre dans un "ComputingEnvironment", les informations des ressources informatiques vont s'ajouter dans le "ComputingResource" (cf. section.9.3.4) du modèle KaliCOnto. La détection de changements d'environnement est réalisée par des chaînes de raisonnement qui détectent soit la présence dans un lieu précis (les informations de ressource sont prédéfinies dans l'ontologie), soit par la connexion à l'hôte d'accueil. Normalement cette connaissance est fournie par un hôte d'accueil (un serveur relativement fixe dans le lieu et accessible) de cet environnement. Ce type de connaissance facilite l'optimisation de l'utilisation des ressources environnementales.

A l'instant t, l'utilisateur peut interagir avec différents environnements de calcul. L'environnement de calcul est diffère de l'environnement physique qui est lié à la localisation actuelle de l'utilisateur (les éléments physiques qui entourent l'utilisateur). L'environnement de calcul peut être distant, par exemple, l'environnement de calcul "Home" peut interagir avec l'utilisateur au travers de l'Internet même quand celui-ci est hors de chez lui. L'environnement de calcul "Home" n'est pas géographiquement lié à l'utilisateur, bien qu'étant à distance il "entoure" l'utilisateur. C'est pour cette raison qu'un l'utilisateur peut avoir plusieurs environnements de calcul en même temps.

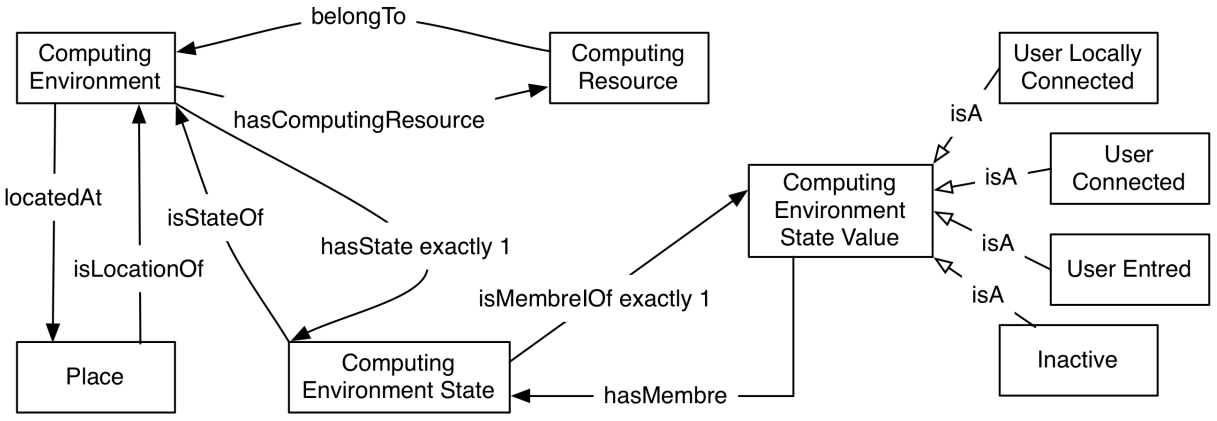

FIGURE 40 – Ontologie du "ComputingEnvironment"

Le "ComputingEnvironment" dispose de ressources informatiques ("ComputingResource") et est localisé en un lieu ("Place") (Q3). Parmi les ressources informatiques d'un environnement une partie est accessible par l'utilisateur et une partie peut ne pas l'être. Ces informations sont contenues dans l'ontologie de ressources ("ComputingResource"). En combinant ces deux ontologies nous pouvons retrouver la réponse à la question Q4. Le "ComputingEnvironment" a un état "ComputingEnvironmentState" qui correspond au mode d'interaction de l'utilisateur. Cet état peut prendre quatre valeurs (Q5) : "UserLocallyConnected",

"UserConnected", "UserEntered", et "Inactive". "UserLocallyConnected" signifie que l'utilisateur est localisé dans le même lieu que le "ComputingEnvironment" et qu'il est connecté à au moins une ressource de cet environnement. "UserConnected" signifie que l'utilisateur est connecté à au moins une ressource d'un environnement distant. "UserEntered" veut simplement dire que l'utilisateur est dans le même lieu qu'un environnement mais qu'il n'est connecté à aucune de ressource de cet environnement. "Inactive" signifie que l'utilisateur n'est ni connecte avec l'environnement, ni situé dans le même lieu. Pour représenter cet état dans l'ontologie nous avons réutilisé le pattern "Objet avec état" (cf. figure.36).

9.3.4 *Domaine "ComputingResource"*

9.3.4.1 *Objectif et analyse*

L'objectif de cette ontologie est d'avoir une connaissance de toutes les ressources du domaine d'adaptation, c'est à dire le leurs descriptions statiques et dynamiques durant l'exécution.

Toutes les ressources ont des descriptions intrisèques (description statique) et des états actuels d'exécution (description dynamique). Ces descriptions identifient des types (catégories) de ressources. Les états actuels sont des états dynamiques qui représentent les fonctionnements des ressources et les environnements ("PhysicalEnvironment" et "ComputingEnvironment") où les ressources se sont situées.

Cette partie de l'ontologie correspond à une taxonomie de ressources orientée adaptation parmi lesquelles on trouve les ressources matérielles (HardwareResource), les ressources logicielles (SoftwareResources), les ressources réseau (NetworkResources) et les ressources d'énergie (PowerResource). Elle représente également les relations entre l'exigence logicielle, la présence et la qualité d'exécution des ressources.

9.3.4.2 *Description*

Nous avons identifié quatre types de ressources informatiques intéressantes pour les adaptations logicielles : "Hardware", "Software", "Power", et "Network" (cf. figure.41).

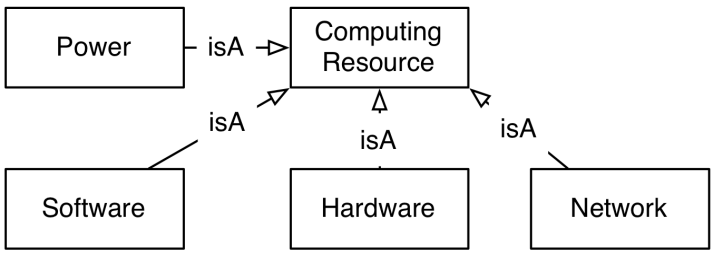

FIGURE 41 – Ontologie "ComputingResource" : relation entre les quatre types de ressources

L'ontologie "ComputingResource" contient quatre parties (cf. figure.42) : "HardwareResource", "SoftwareResource", "PowerResource", et "NetworkResource". Chacune correspond à une taxonomie de ressource, une taxonomie de description et une taxonomie d'état d'exécution.

Nous réutilisons le design pattern : "Description Pattern"[7] (cf. figure.43) pour construire la partie description des ressources de notre ontologie "ComputingResource". Ce pattern permet au concepteur de

7. Description Pattern : http ://ontologydesignpatterns.org/wiki/Submissions :Description

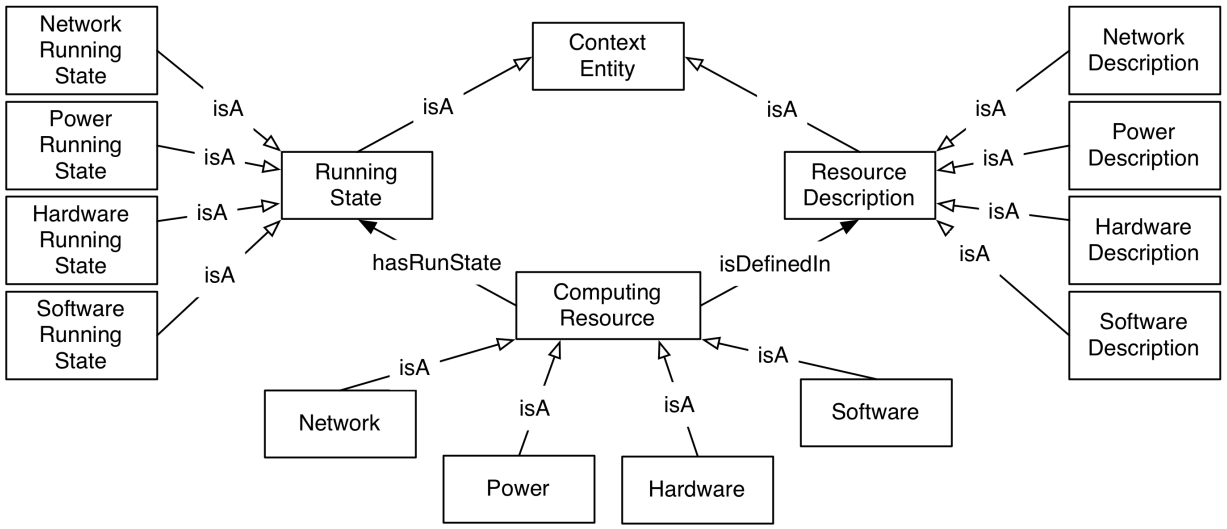

FIGURE 42 – Structure de l'ontologie "ComputingResource"

représenter un contexte (descriptif) ainsi que les éléments qui le caractérisent et ceux qui sont impliqués dans ce contexte.

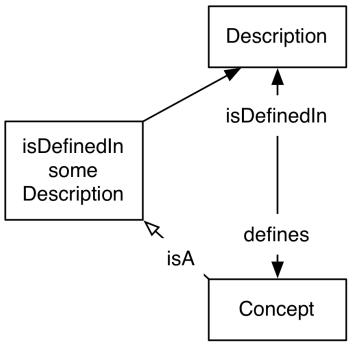

FIGURE 43 – Pattern "Description pattern"

L'ontologie "ComputingResource" décrit les relations entre les quatre types de ressources (cf. figure.44). "Hardware" a besoin d'énergie pour fonctionner. Donc "Hardware" a une relation "*consumeEnergy*" avec "Power" tandis que la relation inverse est "*provideEnergy*". L'existence d'un "Hardware" est une condition nécessaire pour exécuter les "Software". Les "Hardware" sont aussi des lieux pour stocker les fichiers et les ressources nécessaires à l'exécution du "Software". Ceci est exprimé par les relations entre "Hardware" et "Software" qui sont : "*run*"/"*runAt*", "contain" et "use". Le "Hardware" peut être connecté à un réseau et/ou fournir un point d'accès à un réseau ce qui est exprimé par les relations entre "Hardware" et "Network" qui sont : "*connectTo*" et "*provideOf*"/"*isProvidedBy*".De même la relation est "use" indique qu'un "Software" utilise un "Network" pour communiquer.

9.3.4.3 *ONTOLOGIE PARTIELLE DE "KALICONTO" : "HARDWARE"*

Les ressources matérielles ("Hardware") ne concernent que les matériels informatiques. Ces matériels ont soit leur propre source d'énergie (batterie ou secteur) soit dépendent de la source d'énergie d'un autre

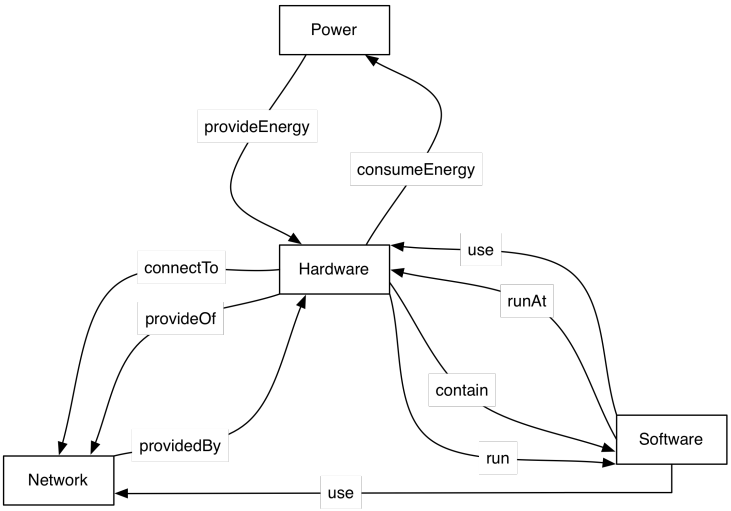

FIGURE 44 – Relations entre les quatre types de ressources informatiques

matériel (dispositif connecté par USB par exemple). Un matériel équipé d'une unité de calcul (CPU) et de mémoire (RAM) et ayant un système exploitation est appelé "Device". Généralement les hôtes peuvent être connectés à différents types de matériels périphériques, par exemple, un clavier, une imprimante ou un écouteur Bluetooth. Ces périphériques peuvent soit être connectés par câble (USB, Thunderbolt, HDMI, etc.) soit par des technologies sans fils (WIFI, Bluetooth, etc.). Les périphériques utilisant des connections sans fil sont des matériels ayant leur propre source d'énergie. Au contraire, les périphériques utilisant des connections par câble sont des matériels sans source d'énergie propre.

Nous définissons par ailleurs le concept de "KaliDevice" comme un hôte qui peut exécuter la plateforme Kalimucho. Ce sont des hôtes qui peuvent être manipulés par notre plateforme d'adaptation pour effectuer des déploiements de composants Kalimucho.

Les périphériques ne sont pas pris en compte pour exécuter des logiciels. En revanche un hôte peut offrir des ressources utilisables pour l'adaptation pendant l'exécution. Les hôtes non KaliDevices peuvent offrir des délégations de services logiciels (sous forme de Web Services) ou offrir des services matériels (par UPnP). Les KaliDevices peuvent offrir un accès à toutes les ressources gérées par la plateforme Kalimucho.

Comme les hôtes ont des ressources de CPU, de mémoire et peuvent avoir des ressources de stockage et d'énergie, nous devons disposer d'une description statique et d'informations de présence de ces quatre types de ressources. Nous avons également besoin de proposer une description des connectivités d'un hôte et de son OS.

Les ressources périphériques sont de deux types : Build-in et Plugin. Le type Build-in désigne des périphériques intégrés dans l'appareil, tandis que le type Plugin désigne des périphériques pouvant être connectés par un câble ou par des technologies sans fil. Les périphériques Plugin ont un statut connecté ou déconnecté ce qui n'est pas le cas des périphériques Build-in. Les informations liées aux ressources de type périphériques sont : le niveau d'énergie (si connecté par sans fils), l'état de connexion (bon, mauvais, déconnecté, connecté), l'état de fonctionnement (activé, désactivé, dysfonctionnement), le type de connexion, le type de périphérique (Build-in ou Plugin), et la fonctionnalité (imprimer, capter une température, capter un mouvement, E/S pour l'utilisateur, etc.).

9.3.4.4 *ONTOLOGIE PARTIELLE DE "KALICONTO" : "SOFTWARE"*

Aux ressources logicielles sont associés une taxonomie logiciel, les services disponibles pour l'utilisateur, les descriptions de logiciel, et leur l'état de fonctionnement (cf. figure.45).

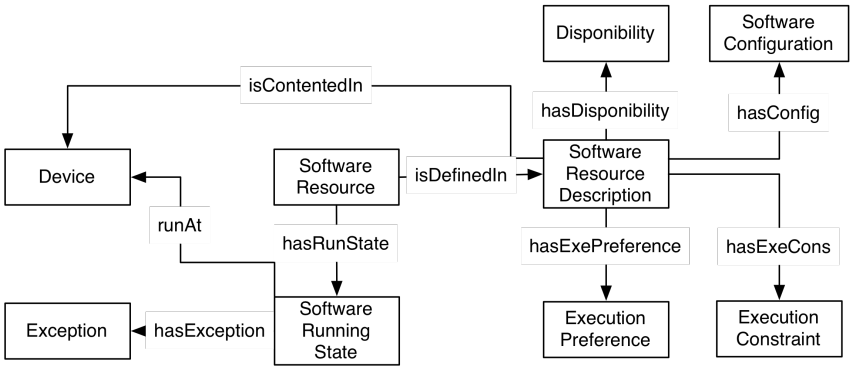

FIGURE 45 – L'ontologie "ComputingResource" : "SoftwareResource"

"SoftwareResource" contient les sous classes : "OS", "Plateforme", "Middleware", "Application", "Service", "Component", etc. (cf. figure.46).

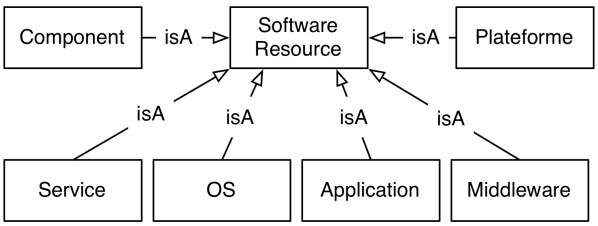

FIGURE 46 – L'ontologie "SoftwareResource"

Nous étendons cette ontologie de "SoftwareResource" au sous domaine spécifique de la plateforme Kalimucho et l'appelons "KaliSoftware" (cf. figure.47).

Cette ontologie représente les relations entre les concepts logiciels de la plateforme Kalimucho. Elle contient également les instances de chaque composant exécuté dans la plateforme pendant le runtime. Dans la section suivante, nous présentons la partie d'ontologie relative au réseau "Network".

Nous désignons les applications Kalimucho par "KaliApp". Une KaliApp est composée de services et possède une configuration d'application. Une KaliApp contient des services de Kalimucho que nous appelons "KaliService" et des services délégués comme, par exemple, des Web Services, des applications Android, etc. Les services délégués ne bénéficient pas des possibilités d'adaptation de la plateforme. Un KaliService est composé de composants Osagaia et de Connecteurs Korrontea. Chaque KaliService fournit ses propres compositions en composants correspondant aux différentes configurations liées aux différentes QdS.

En effet un KaliService peut avoir différents niveaux de QoS correspondant à différentes architectures qui sont représentées dans l'ontologie. Un composant Osagaia peut être connecté à plusieurs connecteurs et chaque connecteur est lié à un port d'entrée ou de sortie. Chaque port peut être connecté à plusieurs

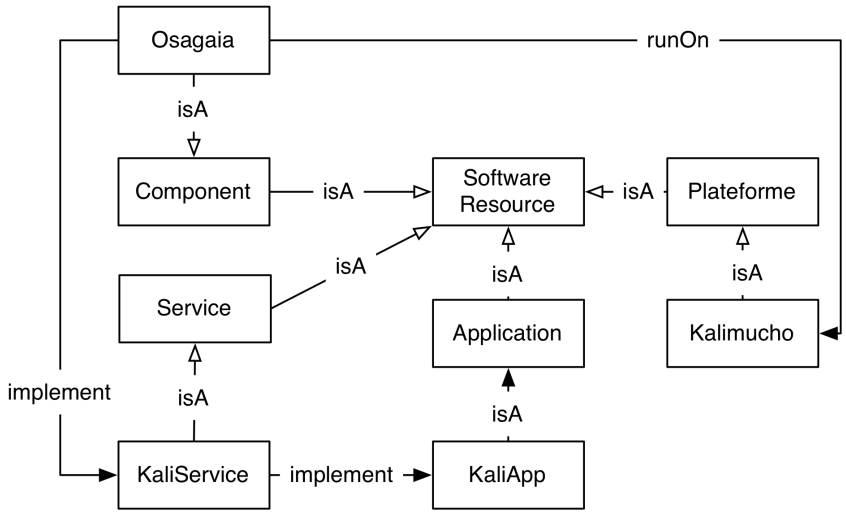

FIGURE 47 – L'ontologie de "KaliSoftware"

connecteurs. Lorsqu'il s'agit d'un port d'entrée il reçoit les données issues de chacun des connecteurs tandis qu'un port de sortie duplique la même donnée sur chacun des connecteurs. (cf. 1.1.1)

Chaque composant de Kalimucho a son exigence d'exécution ("*ExecutionConstraint*" cf. figure.46). Par exemple, un composant intéractive (e.g. GUI), il faut qui soit déployé sur un hôte qui est activé et l'utilisateur qui entrain d'intéracter avec. Pour nous, tous les composants intéractives ont ce même exigence d'exécution par rapport l'intéraction de l'utilisateur.

Chaque composant de Kalimucho peut avoir son preférence d'exécution ("*ExecutionPreference*" cf. figure.46). Par exemple, un composant qui contient l'interface graphique de vidéo chat, il prefere de s'exécuter sur "une machine où équipe un grand écran". Ce type de preférence d'exécution est décrit en sémantique. "Une machine où équipe un grand écran" signifie que un hôte qui a equipé un écran qui est en taille "grand". Donc la taille du écran peut être 24 pouce, peut être 56 pouce, pour le composant sont les même. Durant l'exécution, s'il y a un hôte qui équipé un grand écran ajout dans le domaine, et aussi l'utilisateur est entrain d'intéracter avec (i.e. à cause de exigence d'exécution), dans cette situation, le composant va être déplacé par la plateforme vers l'hôte.

Avec l'exemple précédant, s'il existe pas un hôte qui équipé d'un grand écran, le composant fonctionne tousjour. Par contre, si l'utilisateur n'intérage avec aucun hôte, le composant ne peut pas être déployé. Les exigences d'exécution indiquent dans quelle situation, le composant ne peut pas fonctionner. Ils sont doit être précise, par exemple, un hôte qui équipe un écran, mémoire libre est plus grand que 30Mo. Nous ne pouvons pas dire que un composant ne fonctionne pas dans des hôtes qui ont un peu de mémoire libre. Parce que "un peu" peut tranduire différament avec différent des hôtes. Un hôte a 8Go de mémoire, 200Mo libre, nous pouvons dire c'est "un peu de libre". Mais pour un hôte qui a 512Mo de mémoire, 200Mo libre, c'est pas "un peu". Donc nous utilisons des conditions très précise pour décrire les exigences d'exécution. Dans l'autre main, les preférences d'exécution indiquent les situations idéales pour exécuter le composant. Nous utilisons la façon sémantique pour décrire les preférences d'exécution. Parce que le développeur du composant ne peut pas prévoir dans quelle environnement d'exécution précise que le composant va être exécuté. Par exemple, un composant prefere d'exécuter sur des hôtes qui équipent

d'un écran 53 pouce. Dans un environnement donnée, il y a que un hôte qui équipe d'un grand écran de 42 pouce. Dans cet environnement, la plateforme ne déclanchera pas une adaptation pour déplacer ce composant sur l'hôte qui équipe le grand écran. En résultat, le composant ne peut jamais bénéficier les ressources disponibles. Au contraire, en utilisant la façon sémantique pour décrire ces preférences d'exécution, simplifie le développement et ainsi que l'adaptation.

Les deux facteurs sont très importants pour la prise de décision d'adaptation. C'est ces conditions conduira la déploiement des composants. Nous expliquerons dans le chapitre 13.

9.3.4.5 *ONTOLOGIE PARTIELLE DE "KALICONTO" : "NETWORK"*

Une ressource réseau est caractérisée par sa présence dans un "Environment" c'est à dire si elle est découverte par les hôtes de l'utilisateur. "NetworkResource" contient deux principaux types de réseau : "LAN" et "WAN" (cf. figure.48).

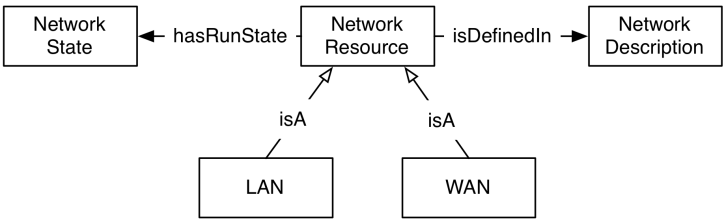

FIGURE 48 – L'ontologie "NetworkResource"

Cette ontologie contient aussi des connaissances sur l'accès à Internet dans un domaine d'adaptation. "Internet" est une "NetworkResource" de type "WAN" (cf. figure.49) et est définie par une "InternetResourceDescription".

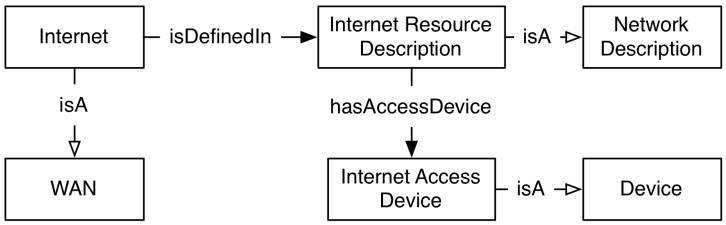

FIGURE 49 – Connaissance d'accès Internet

"InternetResourceDescription" contient la relation "hasAccessDevice" qui désigne un point d'accès Internet d'un domaine d'adaptation. Un domaine d'adaptation peut avoir plusieurs points d'accès Internet. Un "InternetAccessDevice" est un hôte qui peut offrir un accès Internet. La fonctionnalité d'accès Internet peut être éventuellement désactivée à un instant donné.

Nous présentons maintenant la dernière partie de KaliCOnto : celle relative à l'énergie ("Power").

9.3.4.6 *ONTOLOGIE PARTIELLE DE "KALICONTO" : "POWER"*

Une source d'énergie ("PowerResource") est caractérisée par un "PowerRunningState" et une "PowerDescription" (cf. fugire.50). Nous distinguons deux types de "PowerResource" : "Battery" et "AC" auxquelles est associée une caractéristique de stabilité ("Statbility").

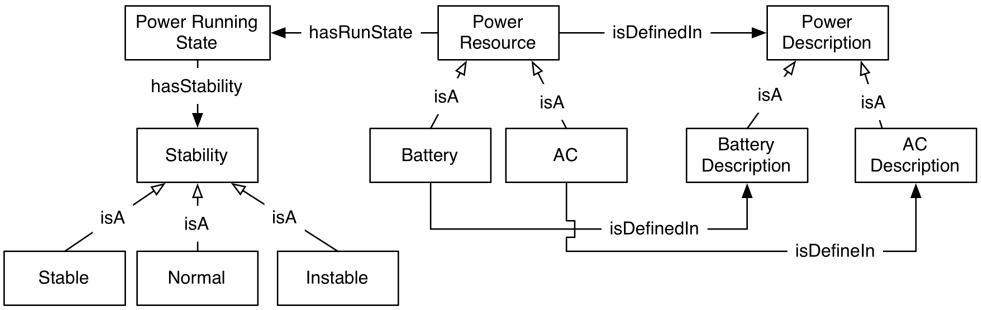

FIGURE 50 – L'ontologie "PowerResource"

Pour terminer nous décrirons les ontologies d'application du modèle KaliCOnto. Elles sont extensibles par les applications de Kalimucho et seront utilisées par la plateforme pour l'adaptation.

9.4 ONTOLOGIES D'APPLICATION DU MODÈLE KALICONTO

Dans cette section nous présentons les ontologies d'application du modèle KaliCOnto. Les ontologies d'application sont des ontologies qui va être exécuté durant le runtime de la plateforme et les applications. Ils sont assemblés par les ontologies de domaine. Dans un environnement dynamique et mobile, nous ne pouvons pas centraliser le conteneur d'ontologie. En effet, si un hôte perdait la connexion avec le conteneur d'ontologie, la plateforme ne pourrait plus fonctionner. Nous proposons donc une architecture adaptable permettant à la plateforme de continuer fonctionner.

Pour bénéficier des possibilités d'adaptation de la plateforme, le conteneur d'ontologie doit être implémenté en composant de la plateforme Kalimucho. La plateforme d'adaptation peut alors le gérer selon les disponibilités des ressources de l'utilisateur.

Nous avons séparé les connaissances de contexte en deux parties : la connaissance du contexte d'un hôte et la connaissance du contexte du domaine d'adaptation (les connaissances partagées dans le domaine). Les connaissances de contexte d'un hôte contiennent les connaissances de ressources informatiques (description et état d'exécution) et les connaissances d'interactions de l'utilisateur. Ceci garantit que la plateforme dispose du minimum de connaissances pour fonctionner. Les connaissances du domaine d'adaptation constituent les connaissances globales d'un domaine d'adaptation. Ce sont les connaissances de l'utilisateur, celles de l'environnement physique et celles de l'environnement informatique.

Nous avons donc deux types de conteneurs d'ontologie dans la plateforme : "Device Ontology Container" et "Domain Ontology Container" (cf. figure.51). Les implémentations de ces deux types de conteneurs utilisent la même librairie : "OWL API"[8]. Dans un domaine d'adaptation, il n'existe qu'un seul "Domain Ontology Container". Pour chaque "KaliDevice", il a un "Device Ontology Container". Le "Domain On-

8. OWL API : http ://owlapi.sourceforge.net/

tology Container" peut migrer dans les "KaliDevices" selon leurs capacités et la disponibilité de leurs ressources.

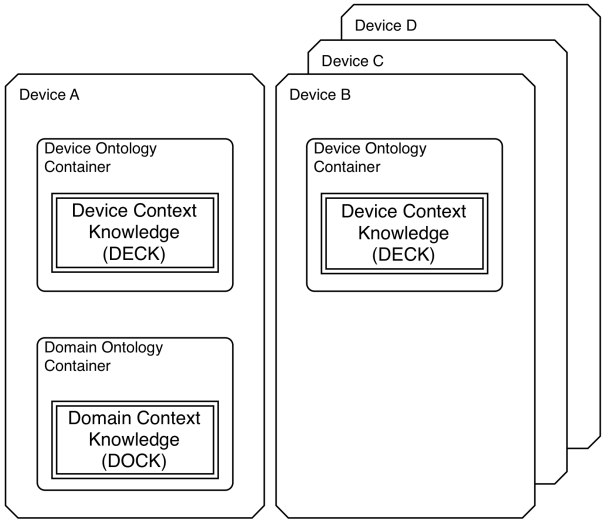

FIGURE 51 – Architecture des conteneurs d'ontologie

Le "Device Ontology Container" contient les états d'exécution et les descriptions des ressources qui appartiennent à l'hôte ("KaliDevice") . Quand le "KaliDevice" entre dans un domaine d'adaptation, il ajoute sa description matérielle au "Domain Ontology Container" (cf. figure.52).

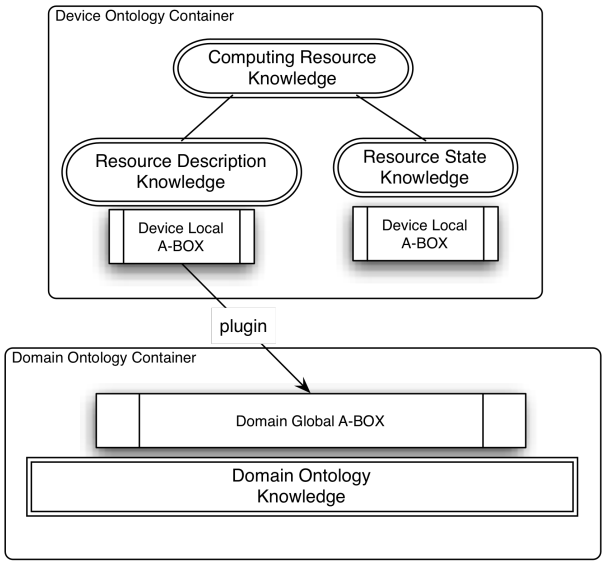

FIGURE 52 – Nouvel hôte ajouté au domaine d'adaptation

Le "Device Ontology Container" contient un "DEvice Context Knowledge" (DECK) qui est constitué de deux ontologies : "Computing Resource Knowledge" et "Kali-User Knowledge" qui représentent les ressources de l'hôte. Le "User Knowledge" ne contient que les interactions entre l'utilisateur et cet hôte (cf. figure.53). Nous supposons que chaque hôte fournit un fichier .owl d'ontologie qui contient toutes les connaissances descriptives de matérieux et de logicielles. Ce fichier peut-être construit par human ou

peut-être construit automatiquement par des services logicielles. Mais ce sujet n'est pas dans la cadre de la thèse.

Enfin, le "Domain Ontology Container" contient un "DOmain Context Knowledge" (DOCK) qui renferme les descriptions des ressources de tous les hôtes du domaine, les connaissances sur l'environnement physique, la localisation de l'utilisateur ainsi que ses modes d'interaction et son état (cf. figure.38). Le "Computing Resource Knowledge" du DOCK ne contient que la description des caractéristiques des hôtes et leur disponibilité.

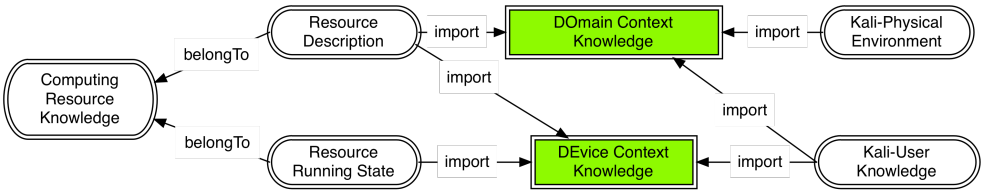

FIGURE 53 – Les importations de l'ontologie "Domain Context Knowledge" et de l'ontologie "Device Context Knowledge"

9.5 CONCLUSION

Pour conclure ce chapitre, nous pouvons classer notre modèle de contexte selon les types d'ontologie proposés par [55, 61] (cf. figure.54). KaliCOnto peut se classer dans trois catégories : Ontologie d'application, Ontologie de domaine et Ontologie générique.

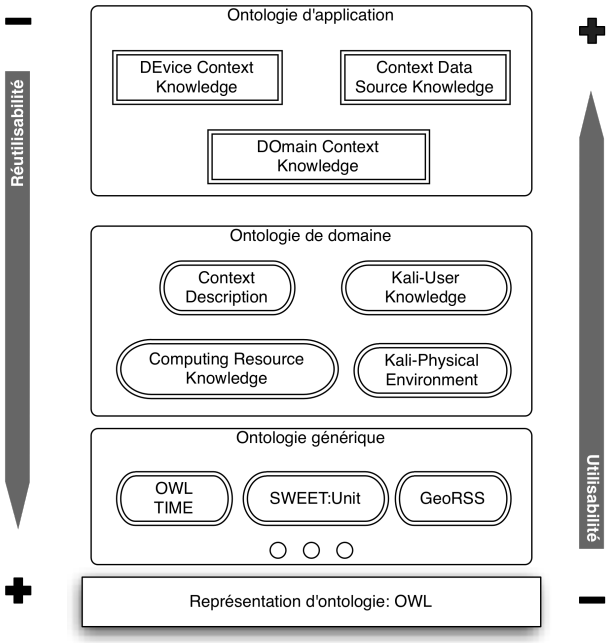

FIGURE 54 – Catégorisation de KaliCOnto

Ontologie d'application : Elles sont construites pour certaine applications, dans notre cas, elles sont construites pour la plateforme mais peuvent être étendues par les applications supportées par la plateforme.

Ontologie de Domaine : Elles sont construites pour certaines zones ou domaines dont elles modélisent les aspects généraux et récurrents. Elles sont préparées pour une utilisation dans des applications spécifiques. Dans KaliCOnto, nous considérons les domaines : utilisateur, ressource informatique, environnement physique et données contextuelles.

Ontologie Générique : Ce sont des ontologies qui décrivent des domaines plus génériques. C'est à dire des domaines universels ou des modèles indépendants du domaine. Par exemple, le temps, la géolocalisation, etc.

Utilisabilité et Réutilisabilité : Il est à noter que plus une ontologie est générale plus elle peut être réutilisée mais moins elle est directement utilisable dans une application donnée.

KaliCOnto est alimenté par la plateforme et aussi par les applications qui mettrent leur information contextuelles de haut niveau. Que la plateforme met les informations qui lui permettront de faire des reconfigurations contextuelles. La façon de créer ces informations de haut niveau, nous avons choisi par chaine de raisonnement (cf. chapitre.11) La chapitre 11 aborde les raisonnements sur le contexte et les mécanismes pour réaliser ces raisonnements. Chaque hôte aliment son "DEvice Context Knowledge" (DECK) par le contexte middleware Kali2Much (cf. chapitre.11). Le "DOmain Context Knowledge" (DOCK) est alimenté par tous les hôtes du domaine. Les détails sont dans le chapitre 11.

La chapitre suivant, nous présentons notre Middleware de gestion du contexte. Ce middleware gére les contextes de bas niveau. Il permet les consommateurs de contexte faire des recherches des informations de contexte par le service de recherche de ressource contextuelle.

10

MIDDLEWARE DE GESTION DU CONTEXTE (KALI2MUCH)

Le rôle essentiel du middleware de gestion du contexte est de permettre la récupération des informations contextuelles de tout type. En ce qui concerne le contexte environnemental, l'utilisation de capteurs dans le domaine du "*Mobile Computing*", soulève une problématique spécifique liée au dynamisme. En effet, les capteurs peuvent disparaître ou apparaître à tout moment de façon imprévisible. Le service doit alors être capable de gérer ce dynamisme. Ceci pose aussi le problème de la complétude des données récupérées en raison de la qualité des communications. De plus, le service est également responsable de distribuer les données captées à ses consommateurs. Dans un domaine d'adaptation, l'utilisateur peut accès des ressources partagés, par exemple, une caméra fixe dans une bâtiment, ou un capteur de température, etc. Ces ressources partagées peuvent aussi accès par les autres utilisateurs. Kali2Much propose une solution de duplication des données pour ce type de partage.

Dans le "*Pervasive*" et le "*Mobile Computing*", les appareils et les applications sont généralement dynamiquement répartis et distribués. Par conséquent les consommateurs peuvent dynamiquement demander l'acquisition des données issues de capteurs distants. Le service doit donc être capable de gérer la redirection de la transmission des données en cours d'exécution.

Dans le monde des capteurs, il n'existe pas d'API universelle. Par exemple, le même type de capteurs peut avoir des API différentes pour l'acquisition des données et pour la configuration du capteur. De même, les façons d'acquérir les données peuvent être différentes, par exemple, Android OS ne permet qu'un seul moyen d'acquisition des données du capteur accéléromètre : par un "Listener" (une méthode de "Call Back") tandis que l'API des SunSpots [1] permet deux modes d'acquisition : soit par un "Listener", soit directement à la demande par les méthodes de type "get*()". Par ailleurs, la diversité de capteur génère une grande variété des données et de leurs types. Ceci pose un problème d'hétérogénéité tant pour l'acquisition des données issues des capteurs que pour leur configuration.

10.1 OBJECTIF

Pour cacher l'hétérogénéité, le service doit offrir une interface unifiée pour tous les consommateurs de contexte. L'interface que nous proposons permet aux consommateurs d'acquérir les données soit en mode "*Query*", soit en mode "*Notify*". Pour le mode "*Notify*", l'interface offre des méthodes de configuration de la notification, par exemple, lors de changements de la valeur mesurée, à une fréquence donnée ou en fonction de conditions sur les données définies par le consommateur. Elle permet également aux consom-

1. SunSpots : http ://www.sunspotworld.com/

mateurs de n'acquérir qu'une partie des données captées, par exemple : un GPS, récupère une position qui contient trois valeurs : latitude, longitude et altitude alors qu'un consommateur peut n'être intéressé que par la seule altitude et un autre par la latitude et la longitude.

Le service doit pouvoir transmettre tout type de données issues de tout type de capteurs. Dans ce but nous avons défini un méta-modèle assurant la compatibilité avec les différents modèles de contexte que nous utilisons. Ce méta-modèle doit également permettre la détection des incomplétudes de données.

Le service pourra alors distribuer les données captées à n'importe quel destinataire et la redirection sera possible en cours d'exécution dans la mesure où il offre un mécanisme configurable de distribution dynamique. La Figure 55 présente la composition du middleware "**Kali2Much**" et sa position dans l'architecture globale de notre middleware appelé **Kalimucho-A**.

En raison des problématiques que nous avons évoquées, le service "ContextCollection" du middleware doit cacher l'hétérogénéité des capteurs, gérer le dynamisme de mobilité des matériels et des logiciels et signaler les incomplétudes de données.

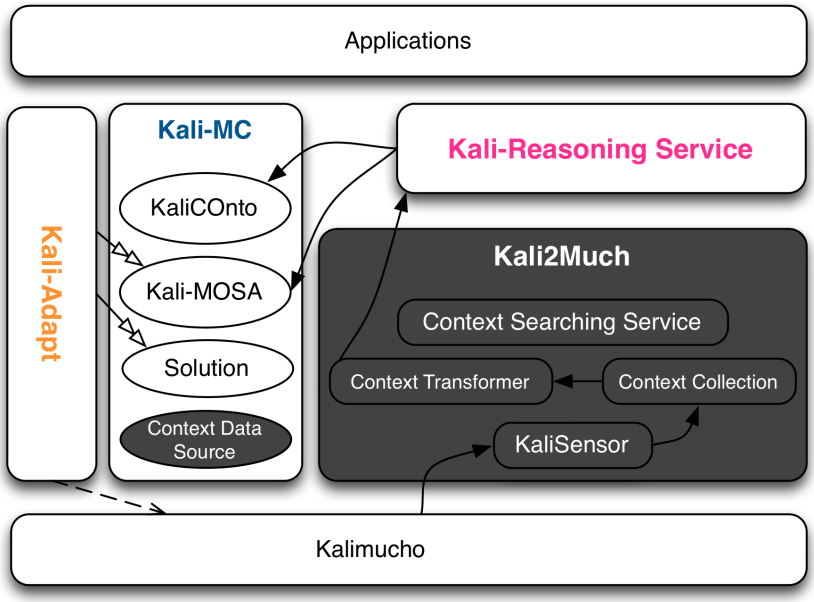

FIGURE 55 – Middleware De Gestion Du Contexte (KALI2MUCH)

10.2 "SENSOR"

10.2.1 *Définition*

Un "Sensor" représente un capteur physique (par exemple un capteur de géolocalisation par GPS) ou un capteur logique (par exemple un composant logiciel de mesure de la charge CPU). Un "Sensor" fournit une API pour obtenir les données captées et pour configurer les notifications (par exemple envoyer les mesures toutes les trois secondes ou signaler un dépassement de seuil). Un "Sensor" ne peut mesurer qu'un seul type de données. Cette règle simplifie la tâche de configuration par type de donnée et permet d'atteindre une granularité fine des configurations par le choix des "Sensor"(s). Si un capteur physique

sur un appareil supporte plusieurs mesures plusieurs "Sensor"(s) seront instanciés. Le rôle d'un capteur se borne à effectuer les mesures puis à envoyer les données à son ou ses clients ou à permettre à son ou ses clients de venir les récupérer. L'obtention des données peut se faire selon deux modes d'observation ("ObservationMode") :

— À la demande ("Query" ou "Pull")

— Par notification ("Notify" ou "Push")

En mode "Query", c'est l'utilisateur du capteur qui interroge le capteur pour récupérer les données par le biais d'une interface de "Query" (une interface de "Query" contient un ensemble de méthodes). En mode "Notify" c'est le capteur qui envoie les données au client conformément à sa configuration.

10.2.2 *Modes d'observation*

Le méta-modèle présenté à la figure 56 correspond aux différents modes d'observation que nous avons retenus.

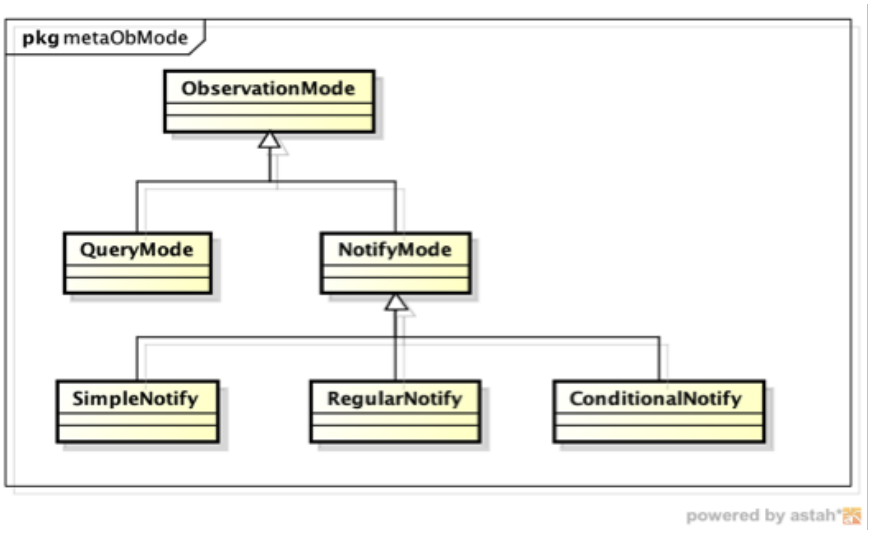

FIGURE 56 – Meta-modèle de "ObservationMode"

Si le mode "Query" est, par nature, simple le mode "Notify" se décompose, quant à lui, en trois modes :

Le mode "*SimpleNotify*" correspond à un fonctionnement dans lequel le capteur va notifier les données au client à chaque nouvelle mesure. Ceci veut dire que le capteur effectue des mesures en continu et les transmet au client. C'est le mode de fonctionnement que propose, par exemple, Android pour les capteurs d'environnement (température, accélération, etc.). L'espace de temps séparant deux mesures peut ne pas être constant.

Le mode "*RegularNotify*" correspond à un fonctionnement dans lequel le capteur va notifier les données au client à une fréquence choisie par celui-ci. Ce mode permet au client d'éviter de devoir traiter un trop grand nombre de mesures redondantes (par exemple quel est l'intérêt de disposer d'une mesure de température ambiante chaque dixième de seconde ?). Il permet également d'effectuer certains types de calculs comme des taux d'accroissement qui ne peuvent se faire que si l'on dispose d'une échelle de temps pour les mesures.

Le mode "*ConditionalNotify*" correspond à un fonctionnement dans lequel le capteur ne transmettra d'information que lorsque certaines conditions définies sur les mesures seront vérifiées. Par exemple, lorsque la position GPS du périphérique est à l'extérieur d'une zone donnée. Les conditions peuvent être liées entre elles par des opérateurs logiques (ET, OU, NON) pour finalement produire un résultat binaire (TRUE ou FALSE), si cette valeur est "TRUE" le "ContextCollector" va envoyer une notification à son consommateur ; sinon, il ne fait rien.

Pour la réalisation du "Sensor" nous avons conçu un composant logiciel supportant des fonctionnalités spécifiques pour les multi-consommateurs et ainsi nous avons pu proposer une API unifiée. Dans la section suivante nous présentons ce composant logiciel.

10.3 KALISENSOR : CAPTEUR EN COMPOSANT COMPOSITE

10.3.1 *Objectif*

L'objectif est de définir un capteur (Un KaliSensor) ayant les propriétés décrites précédemment (Sensor) mais, en outre, capable de supporter un fonctionnement multi-utilisateurs. Ceci signifie que son API doit supporter la multi-notification, la multifréquence et les multi-conditions. La plupart des capteurs physiques ne proposent pas ce type de fonctionnement. Il faudra donc, pour avoir une interface unifiée et permettre l'utilisation des capteurs dans un environnement de type "*Pervasive Computing*", que notre middleware prenne en charge ce problème. C'est ce que nous présenterons en détail dans la section "ContextCollector".

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

Ainsi un KaliSensor est un composant composite constitué d'un Sensor et d'un "ContextProvider" selon le schéma de la Figure 57.

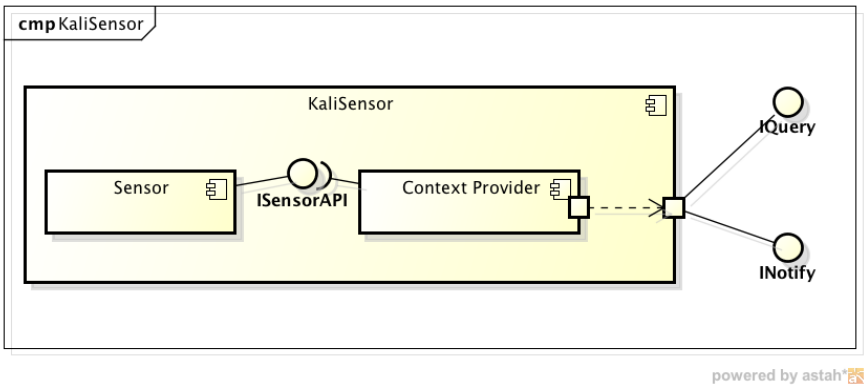

FIGURE 57 – Structure du composant logiciel "KaliSensor"

Nous allons maintenant décrire plus en détail la partie "ContextProvider" et la circulation des données dans ce composant.

10.3.2 *"ContextProvider"*

Le "ContextProvider" est un composant logiciel lié au capteur physique ou logique tel que défini par "Sensor". Ceci signifie qu'à un modèle de "Sensor" peuvent être associées différentes implémentations de "ContextProvider". Le "ContextProvider" fournit 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". La distribution ainsi que la redirection dynamique des données captées vont être gérées par le noyau de la plateforme "Kalimucho", voir article [41] pour de plus amples détails.

Pour disposer d'une API unifiée qui supporte tous les types de données captées par des capteurs hétérogènes il faut définir un méta-modèle de données. La section suivante présente ce méta-modèle.

10.3.3 *Méta-données pour la transmission unifiée*

Le "ContextProvider" fournit deux interfaces "*IQuery*" et "*INotify*" correspondant aux deux modes d'observation possibles. Ces deux interfaces utilisent le même méta-modèle de données (cf. figure.62) que celui utilisé lors de la transmission (cf. figure.58). Ce modèle définit deux méta-modèles d'informations :

— Méta-modèle des informations sur les capteurs, par exemple : l'identification, le type de capteur, le type de donnée mesurée, etc.

— Méta-modèle des données captées

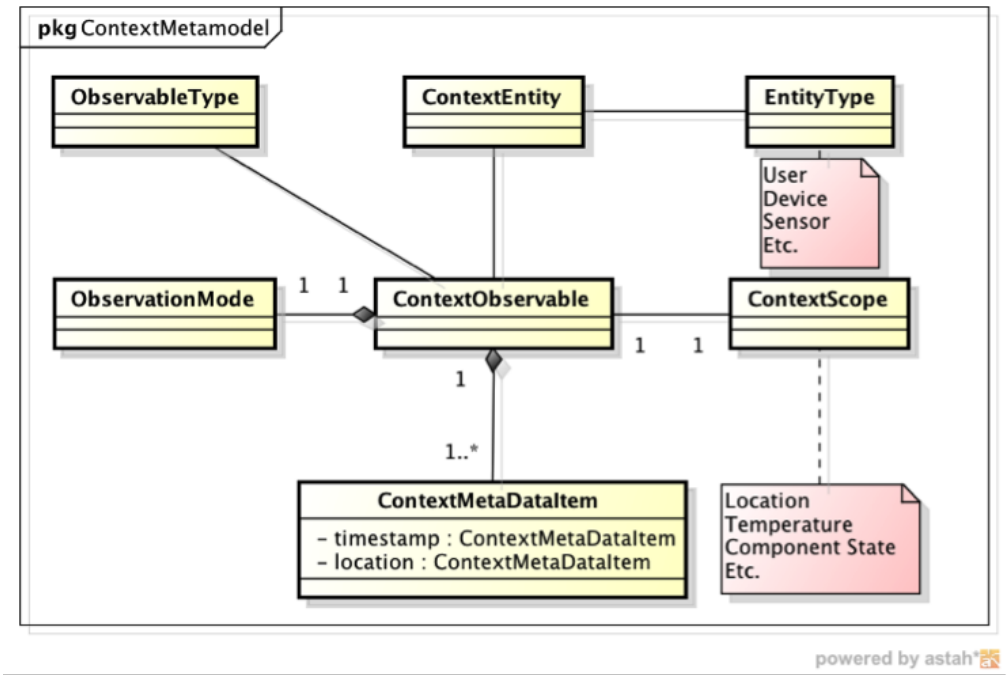

FIGURE 58 – Méta-données du "ContextProvider"

Le "ContextObservable" représente la vue logique d'un capteur. C'est un "ContextEntity" avec 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 contenir 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 supportés.

Le "ContextObservable" peut supporter un ou plusieurs "ObservationMode" qui correspondent aux modes d'observation précédemment définis.

Le "ObservableType" permet d'indiquer si les données sont produites directement par un "Sensor" ou si elles ont déjà fait l'objet d'une interprétation par le Raisonneur de Contexte (il s'agit d'un composant logiciel capable de raisonnement pour interpréter des informations de contexte brutes qui sera décrit en chapitre 11). Par exemple, une température d'ambiance produite par un "Sensor" sera de 28° C tandis qu'après interprétation par le Raisonneur de Contexte elle sera, pour l'utilisateur X, "chaude". De même la coordonnée GPS "43.47774 ,-1.508791" correspond à l' "IUT de Bayonne". Un "ObservableType" peut être de deux types : "ElementaryObsType" et "InterpretedObsType".

Toutes les informations présentées ci-dessus sont conformes au méta-modèle des informations de capteur (cf. figure 62). Celles mesurées par un "Sensor" seront de type "ElementaryObsType".

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. La date est nécessaire car, lors de l'envoi de données par réseau, l'ordre peut en être inversé et le timestamp permet au récepteur de remettre les données dans l'ordre. De plus, l'analyseur de contexte peut avoir besoin de ces informations pour inférer et raisonner sur de nouvelles données. La localisation permet également à l'analyseur d'inférer et de raisonner. Sans elle une donnée peut perdre son sens, par exemple, une température de 28 degrés, ne veut rien dire si l'on ignore où elle a eté mesurée. Le timestamp et la localisation sont, elles-mêmes, des instances de "ContextMetaDataItem" qui est défini par : "Value", "Unit", "Representation" et "ContextDataSignification" (cf. figure.59).

"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.

10.3.4 *Les interfaces "IQuery" et "INotify"*

Ces deux interfaces unifiées du KaliSensor permettent l'acces aux données contextuelles mesurées.

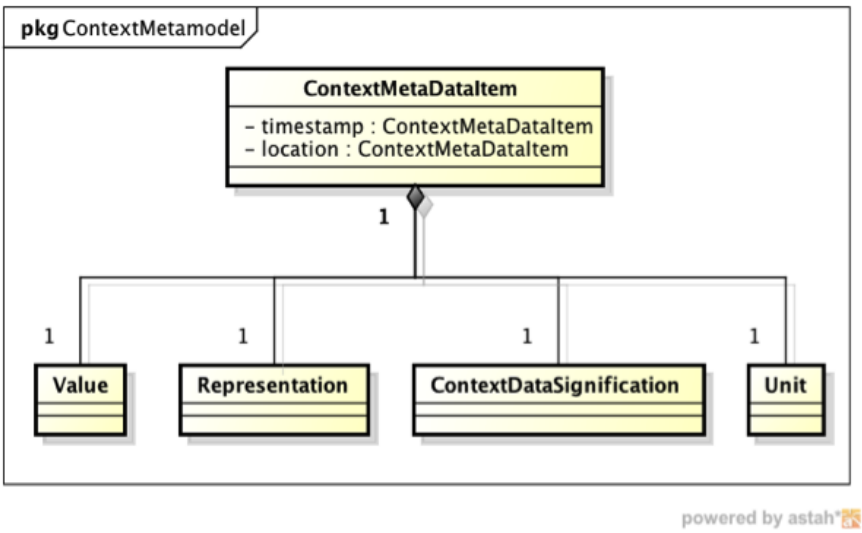

FIGURE 59 – Méta-modèle de "ContextMetaDataItem"

Les consommateurs d'un KaliSensor peuvent accéder aux données par l'interface "IQuery". Cette interface contient les cinq méthodes suivantes :

```
ContextObservable getObservableInfo() ;
ContextScope getObservableContextScope() ;
List<ContextMetaDataItem> getObservableDataModel() ;
List<ContextMetaDataItem> getData() ;
ContextMetaDataItem getDataItemByImplication(DataSignification imp) ;
```

"getObservableInfo", "getObservableContextScope" et "getObservableDataModel" sont les méthodes communes aux deux interfaces. Elles permettent d'obtenir les informations du capteur.

Exemple : un consommateur veut connaître la position géographique courante à partir d'un GPS. Dans un premier temps il peut connaître les champs de données du KaliSensor associé au GPS grâce à la méthode "getObservableContextScope". Cette méthode retourne un objet "ContextScope" qui contient un nom de "scope" (coordonnées GPS, Température, Luminosité, etc.) et une liste de "ContextDataSignification" qui donnent la signification de chaque champ. Dans cet exemple le nom est ["GPS Coordinate"] et la liste contient les trois éléments suivants :

```
[<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, le consommateur utilise la méthode "getObservableDataModel" pour obtenir un modèle de données lui permettant de vérifier le type d'unité et 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=<"degree">,
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=<"degree">,
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>>]
```

Le KaliSensor permet également l'acquisition de données par notification par l'interface "INotify". Cette interface contient les neuf méthodes suivantes :

```
ContextObservable getObservableInfo() ;
ContextScope getObservableContextScope() ;
List<ContextMetaDataItem> getObservableDataModel() ;
boolean subscribeNotification(ContextDataSignification [] dataScopes, ISensorDataListener listener);
boolean unsubscribeNotification(ContextDataSignification [] dataScopes, ISensorDataListener listener);
int getMinFrequency();
boolean isRegular();
int getCurrentFrequency();
void setFrequency(int frequency);
```

Outre les méthodes "getObservableInfo", "getObservableContextScope" et "getObservableDataModel" communes aux deux interfaces, elle offre des méthodes permettant au client de s'inscrire pour recevoir des notifications simples ou régulières. Dans la mesure où le fonctionnement en multi-utilisateurs n'est pas fourni par le matériel, la fréquence de fonctionnement du capteur sera calculée en fonction de la fréquence maximale définie par les clients. Bien entendu, cela signifie que, selon les possibilités de vitesses de fonctionnement offertes par le capteur, les fréquences de notification des clients ne pourront pas être respectées avec une totale exactitude. L'accès aux données est assuré par le composant logiciel "ContextCollector" présenté dans la section suivante.

10.4 "CONTEXTCOLLECTOR"

10.4.1 *Objectif*

Les objectifs du "ContextCollector" sont le support du fonctionnement en mode multi-utilisateurs et la gestion des conditions de notification. Pour supporter le multi-utilisateurs, il faut que le "ContextCollector" supporte la multi-notification, la multifréquence, et les multi-conditions de notification par les clients. Dans notre proposition chaque utilisateur peut demander une ou plusieurs notifications au même capteur. Chaque notification peut être de type "SimpleNotify", "RegularNotify" ou "ConditionalNotify" et correspond à une instance de "ContextCollector" différente.

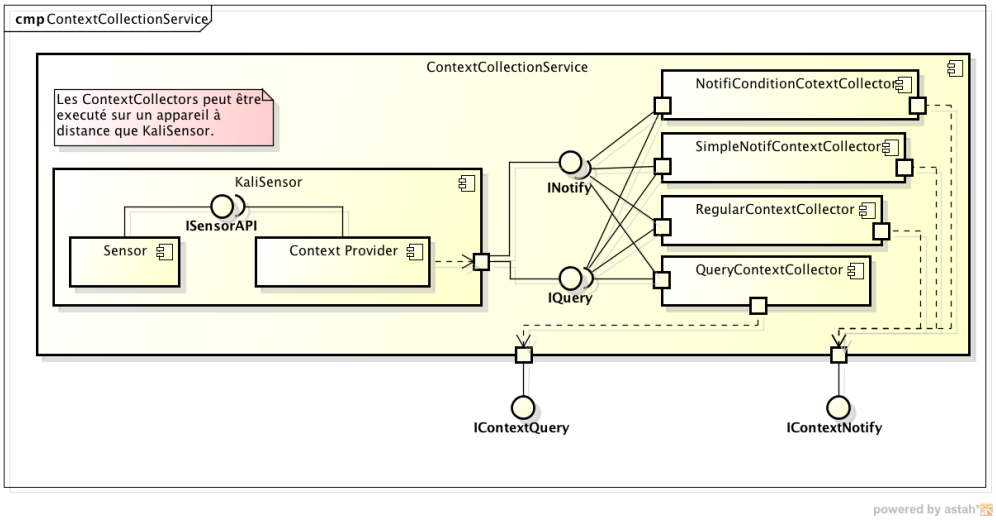

FIGURE 60 – Les quatre types de "ContextCollector"

Le "ContextCollector" est un composant logiciel configurable selon les demandes des clients (cf. figure.60). Chaque instance de "ContextCollector" est configurable par son client soit en mode "Query", soit en mode "SimpleNotify" (le client sera notifié lors d'un changement de valeur), soit en mode "RegularNotify" avec sa propre fréquence, soit enfin en mode "ConditionalNotify" avec ses propres conditions de notification. Nous proposons un langage d'expression des conditions permettant aux clients 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", Value
```

Ce langage permet de définir des conditions complexes facilitant la notification précise d'événements de contexte. Par exemple : un client peut demander à être notifié quand la latitude dépasse 45° et que la longitude est inférieure à 11° mais que l'altitude n'est pas de 20 mètres. Nous pouvons représenter ceci dans notre langage par :

```
["Latitude" >= "45"] AND ["Longitude" <= "11"]
AND ["Attitude" != "20"]
```

10.4.2 *Entrées et Sorties*

Le "ContextCollector" utilise le "ContextProvider". Il va recevoir les données envoyées par le "ContextProvider". Ces données sont conformes au méta-modèle (ContextMetaDataItem) telles que présenté en section 10.5 (page 83). Le "ContextCollector" utilise le même méta-modèle pour transférer les données aux utilisateurs. Il propose deux interfaces pour les utilisateurs : "IContextQuery" et "IContextNotify". Le mode Query "IContextQuery" a les mêmes méthodes que "IQuery" dans le "ContextProvider". "IContextNotify" offre, en plus des méthodes de "INotify", des méthodes permettant à l'utilisateur de s'inscrire (et se désinscrire) en indiquant la fréquence de notification souhaitée ou une condition de notification.

Si un "Sensor" ne fournit qu'un seul mode d'acquisition, par exemple uniquement le mode "Query", c'est le "ContextCollector" qui se charge de mettre en œuvre les autres modes pour un utilisateur qui aurait besoin fonctionner en mode "Notify".

10.4.3 *Utilisation*

Un "ContextCollector" ne peut utiliser qu'un seul "KaliSensor" qui ne produit qu'un seul type de donnée contextuelle. Pour satisfaire les besoins de tous les consommateurs, chaque consommateur de "ContextCollector" peut s'attacher autant de "ContextCollector" qu'il le veut. C'est à dire que chaque "ContextCollector" gère une configuration d'acquisition, et donc que si un consommateur a besoin de plusieurs types d'acquisition, il peut créer autant de "ContextCollector" que de configurations.

Reprenons l'exemple de la géolocalisation qui a été utilisé dans la section "ContextProvider" et dont le schéma est décrit dans la figure.61. Il y a quatre consommateurs de contexte, chacun a son propre "ContextCollector" et les quatre "ContextCollector" sont différents. Ils représentent quatre possibilités de configuration d'une instance de "ContextCollector". Le consommateur 3 (appelé Consommateur Local) se situe sur la même machine que le "Sensor" alors que les consommateurs 1 et 2 sont distants. Le "Sensor" est un GPS d'un téléphone fonctionnant sous "Android". Il fonctionne en mode "Notify" avec une fréquence configurable.

Consommateur 1 : il a besoin d'une notification conditionnelle, comme celle présentée à la section 10.4 (page 81). Dès que la condition est vérifiée, la notification se produira à chaque seconde. Il a également besoin de fonctionner en mode "Query". Le "LocationContextProvider" répondra à sa demande.

Consommateur 2 : il a besoin d'une notification régulière. Il demande une fréquence de notification de 20 secondes.

Consommateur 3 : il veut observer le déplacement de ce téléphone. Il demande donc une notification simple. Il sera notifié à chaque modification de la position.

Le "NotificationConditionContextCollector" va recevoir une notification toutes les 20 secondes et chacune va déclencher une vérification de la condition. Si elle est vérifiée, il notifiera le consommateur.

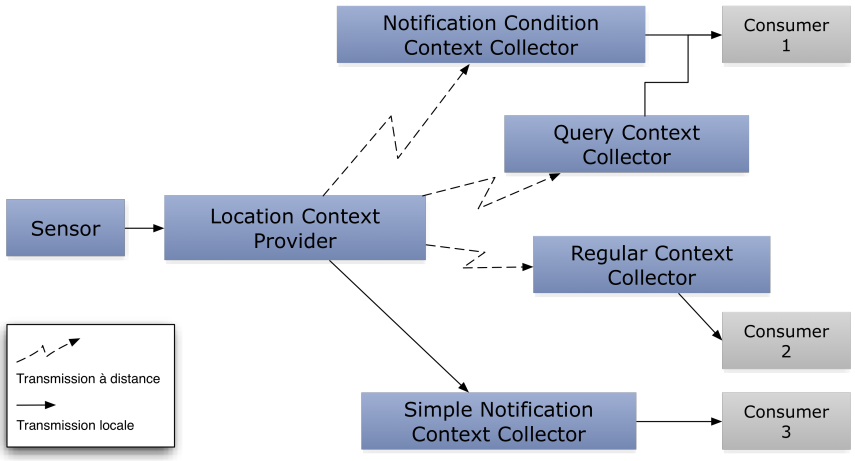

FIGURE 61 – Exemple de "ContextProvider" et de "ContextCollector"

10.5 MÉTA-MODÈLE DE CONTEXTE

Le Méta-modèle de Contexte (cf. figure.62) est un modèle abstrait permettant l'unification des données et leur transport dans "Kali2Much". Le "ContextEntity" est une source de données. Le "ObservableType" indique si les données sont directement issues d'un capteur direct ou ont été interprétées par des "ContextReasoner". À ce niveau, il n'y a que des données brutes c'est à dire que seul le type "ElementaryObsType" sera pris en compte. Le "ContextScope" représente les relations entre les représentations des données et leur structure par des "ContextScope".

Ce Méta-modèle est utilisé pour la transmission des données entre le bas niveau du fournisseur de contexte (les capteurs physiques et logiques) et le consommateur de contexte. Les traitements de ces informations ne sont pas pris en compte à ce niveau.

A ce méta-modèle va correspondre une ontologie contenant la connaissance sémantique et aidant au raisonnement sur le contexte (pour plus de détails voir la partie Contexte Modèle 9, page 51). L'"EntityType" définit l'origine de l'information, par exemple : un utilisateur, un PC, un capteur de température, etc. Un "ContextEntity" est aussi un "ContextObservable" qui est un "ContextType" représentant les types du contexte, par exemple : localisation, température, charge d'un CPU, etc. Un "ContextObservable" a un "ObservationMode" qui correspond aux deux modes d'acquisition : "QueryMode" et "NotificationMode". Il a un "ContextMetaDataItem" représentant la valeur concrète d'une information de contexte. L'unité de la valeur est contenue dans "Unit" et sa forme dans "Representation". Par exemple, une valeur de température peut avoir une "Representation" de type "valeur + unité" (35° C) ou une "Representation" de type "XML" (<Temperature>35</Temperature>) ou de type "RDF" etc. Les "Unit" et les "Representation" possibles sont définis par le "ContextType". Ceci permet une transformation de la représentation par le consommateur du contexte.

Chaque consommateur de contexte possède une connaissance du "ContextScope" qui le concerne. À partir de ces connaissances il peut connaître les structures des données possibles, la portée sémantique de chacune, leur représentation et les relations entre les différentes unités pour pouvoir effectuer une traduction des unités [143]). Par exemple, si un consommateur s'intéresse à la géolocalisation, il a un

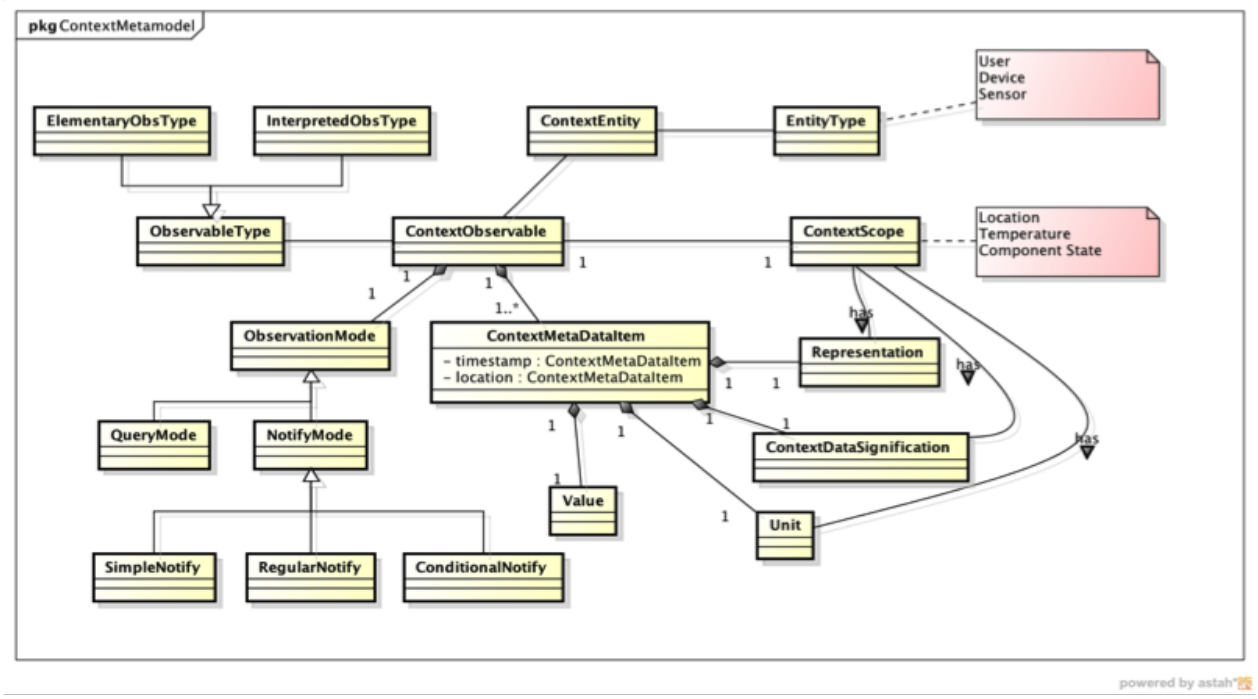

FIGURE 62 – Méta-modèle de Contexte

"ContextScope" de géolocalisation qui contient un triplet "ContextDataSignification", "Representation" et "Unit" pour chaque élément de donnée (cf. figure.63). Donc ce consommateur dispose de trois "ContextDataSignification" : "Latitude", "Longitude" et "Accuracy". Dans cet exemple la composition du "ContextScope" est différente de celle de l'exemple donné à la section 10.5, car ce qui nous intéresse ici c'est la "Latitude", la "Longitude" et l'"Accuracy". Ceci montre que le type de "ContextScope" peut changer selon le consommateur. C'est également vrai pour la représentation des données et pour leur unité. Le service "ContextCollection" utilise un "ContextDataSignification" pour identifier les données de contexte correspondant à la demande du consommateur. Comme le montre la figure 63, le service "ContextCollection" peut fournir des données sous différentes "Representation" ou "Unit" selon la demande du consommateur. La transformation sera faite par le consommateur. En effet, le service "ContextCollection" ne peut anticiper et contenir tous les algorithmes possibles de transformation.

Les consommateurs de contexte sont capables de faire des vérifications et de détecter des données incomplètes (par exemple, lorsque dans des données GPS il manque la valeur de longitude). Ils peuvent également réaliser des vérifications de cohérence sémantique des données.

10.6 CAS D'UTILISATION DU "CONTEXTCOLLECTION" SERVICE

Dans cette section nous allons décrire une application de contrôle de chauffage. Nous utiliserons la position de l'utilisateur et la température de la maison pour piloter le fonctionnement du chauffage de sa maison. Il y a deux "KaliSensor", l'un est un GPS, l'autre est un capteur de température. Le GPS est un capteur intégré dans le portable de l'utilisateur il est donc distant. Le capteur de température est installé dans la maison de l'utilisateur et est donc local. (cf. figure.64)

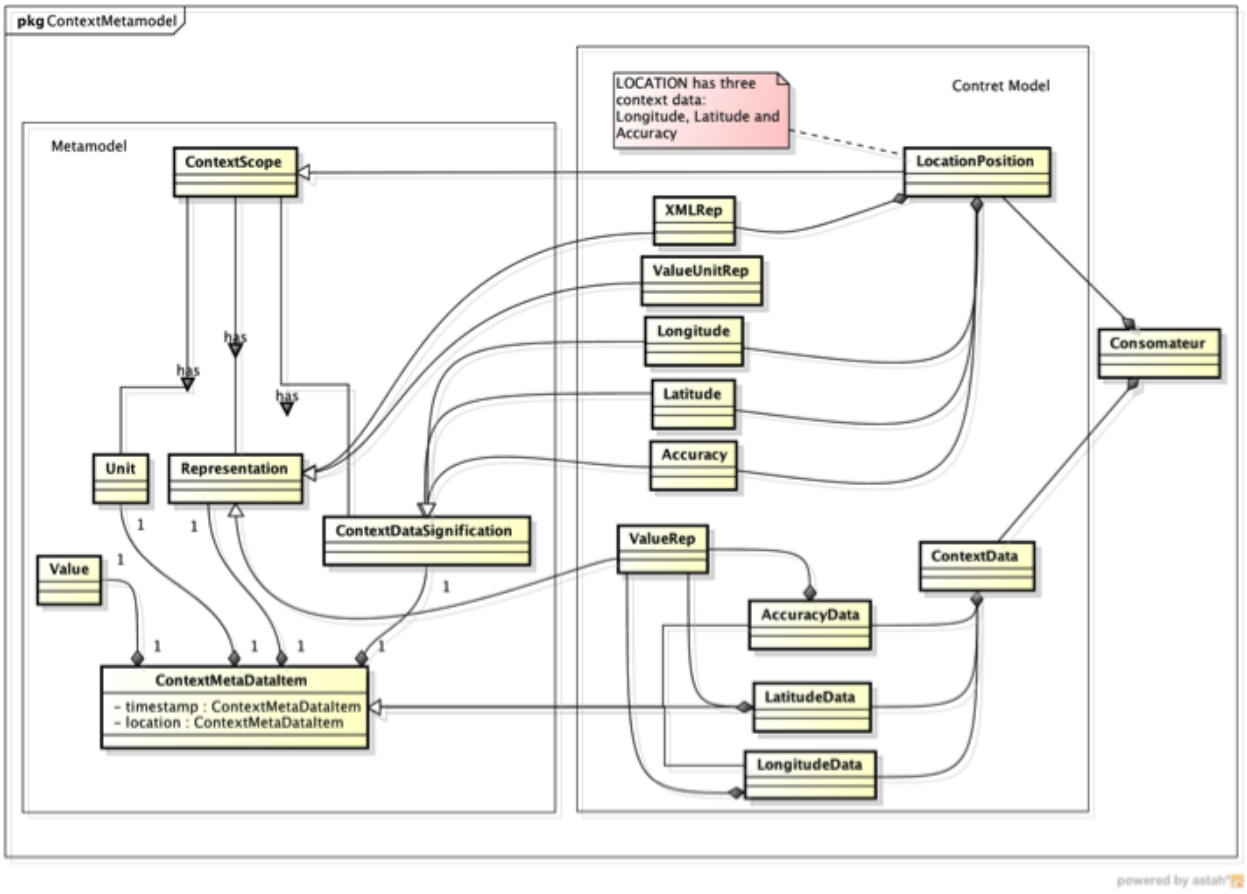

FIGURE 63 – Exemple de "ContextScope" de géolocalisation

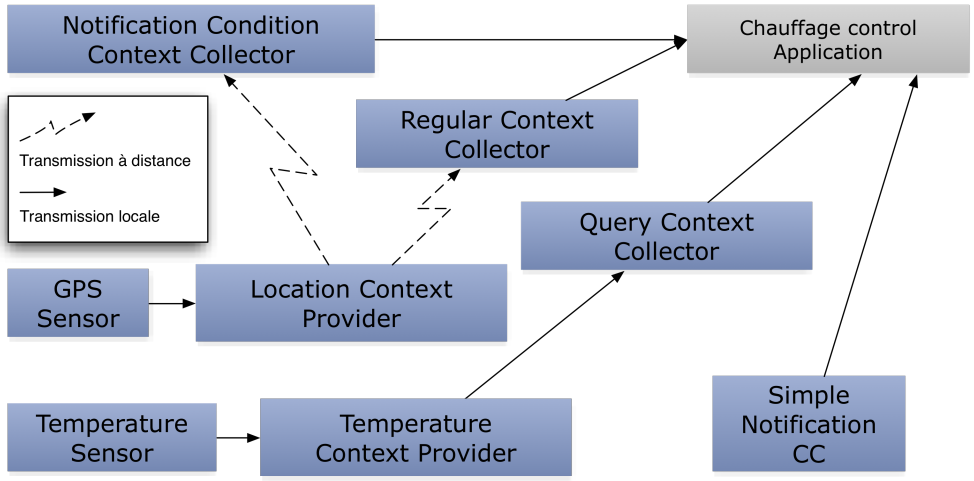

FIGURE 64 – Illustration du scénario

L'application gère le fait que si l'utilisateur s'approche de sa maison et que la température y est inférieure à 18° C, il faut activer le chauffage pour avoir une température plus élevée avant que l'utilisateur n'arrive chez lui. L'application tient compte du mode de déplacement de l'utilisateur : l'application doit recevoir la position de l'utilisateur toutes les minutes dès que l'utilisateur se situe dans un rayon de 10Km de la maison. Ce rayon diminue à 1Km. Selon la direction et la distance de l'utilisateur, l'application va régler la puissance du chauffage.

Cette application utilise un Context Collector (Regular Context Collector sur la figure 64) lié au GPS et lui envoyant la position de l'utilisateur à intervalles réguliers. Elle va alors utiliser un autre Context Collector (Notification Condition Context Collector sur la figure 64) lui aussi lié au GPS et pouvant être programmé pour détecter une condition de notification (l'utilisateur entre dans un cercle de 10km/1km selon son mode de déplacement actuel). Ce "Notification Condition Context Collector" est créé par l'application qui le règle, par exemple, à [(Latitude > 43,487330 AND Longitude > -1.524318) AND (Latitude < 43,469364 AND Longitude < -1.494198)].

Enfin, un Context Collector lié au capteur de température (Query Context Collector sur la figure 64) permet à l'application d'accéder à la température de la maison en mode "Query" pour contrôler le chauffage.

Il est à noter que les liaisons entre le Context Provider associé au GPS (Location Context Provider sur la figure 64) et les Context Collectors (Regular Context Collector et Notification Condition Context Collector sur la figure 64) se font au travers du réseau puisque le GPS est distant. Comme on l'a vu le rôle du Context Provider est non seulement de recueillir les mesures auprès du capteur physique mais également de les transmettre après les avoir formatées selon le méta-modèle d'informations de contexte défini précédemment. Ce transport d'information est entièrement pris en charge par la plateforme Kalimucho au travers de la création d'un connecteur et est donc totalement transparent à l'application qui peut ainsi utiliser indifféremment des capteurs locaux ou distants.

Ce scénario montre l'utilisation du service "ContextCollection". Un consommateur peut créer plusieurs "ContextCollector" pour recevoir des notifications de contexte ou pour récupérer directement des informations de contexte. Chaque "Sensor" est relié à un seul "ContextProvider" et constitue un "KaliSensor" qui peut être distant ou sur le même dispositif que le "ContextCollector". Un consommateur peut ainsi créer ou supprimer des "ContextCollector" dynamiquement.

10.7 SERVICE DE RECHERCHE DE RESSOURCES CONTEXTUELLES

10.7.1 *Objectifs*

Dans les applications ubiquitaires les informations de contexte sont distribuées et sont dynamiquement modifiées dans le temps. Dans les sections précédentes de ce chapitre nous avons proposé une solution pour collecter des informations de contexte. Cette section montre comment rechercher ces informations de contexte dans un environnement pervasif et mobile.

Dans [63], sont présentés cinq types d'informations de contexte :

— Dynamique vs. Statique : Une information de contexte peut changer dans le temps comme une température ou être statique comme le nom d'un utilisateur.
— Flux de données : Une information de contexte peut être issue d'une source continue de données comme un capteur.
— Métadonnées et Qualité de Contexte (QoC) : Une information de contexte peut être liée à des métadonnées. Elle peut être éphémère, imparfaite, incomplète, erronée ou ambiguë. Sa qualité peut changer au cours du temps par exemple selon la précision.
— Relation spatiale et proximité : Une information de contexte peut être spatiale et liée à la proximité. Sa sémantique dépend de la localisation, par exemple la température dans la maison de l'utilisateur X.
— Situation et émotion : Une information de contexte peut représenter une situation liée à d'autres contextes, par exemple 'David est en réunion'. Une information de contexte peut correspondre à une émotion d'utilisateur. De telles informations sont généralement très difficiles à capturer.

L'objectif principal du service de recherche de ressources contextuelles est de trouver des informations de contexte captées par des KaliSensors dans un environnement mobile. Il doit permettre aux services d'interprétation de contexte et aux applications contextuelles de trouver et d'accéder facilement aux informations contextuelles disponibles. Ses objectifs sont donc :

— Trouver des informations de contexte selon différents critères (QoC qualité de Contexte [63] , relation spatio-temporelle, etc.)
— Accepter des flux de données comme type de résultat.
— Permettre de définir des requêtes incluant des conditions de notification.

Grâce à nos modèles de données contextuelles, les situations et les statuts de l'utilisateur sont modélisés à un niveau plus abstrait. Ils sont représentés de façon sémantique dans des ontologies implémentant ces modèles. Ainsi les recherches peuvent être effectuées par "*DL Query*" [2].

10.7.2 *Service de recherche de contexte*

Dans cette section nous présentons notre service de recherche de contexte. D'abord nous expliquons comment ce service fonctionne. Ensuite nous présentons le noyau de ce service et l'ontologie de connaissance des sources de contexte. Pour terminer nous présentons l'implémentation de ce service.

Ce service permet de rechercher n'importe quelle information de contexte dans un domaine d'utilisateur (cf. chapitre 1, section 1.1.2). Le service supporte la recherche de contexte statique, la recherche de contexte dynamique et la recherche de contexte sous forme de flux. Les contextes statiques sont des descriptions comme un profil d'utilisateur, les caractéristiques de l'OS ou du matériel, etc. Ils sont généralement accédés à la demande. Les contextes dynamiques sont des informations mesurées par des capteurs. Leurs valeurs changent et peuvent accédées par les consommateurs à la demande ou en mode de notification (push). Dans ce dernier cas le consommateur peut configurer les paramètres de ces notifications comme, par exemple, la fréquence de notification, la condition de notification etc. Les contextes sous forme de flux

2. OWL-DL Query : http ://www.w3.org/TR/owl-semantics/

sont des flux de données continues qui changent très rapidement comme, par exemple, la vidéo, l'audio, etc. Ils sont envoyés au consommateur en mode "push".

Dans un environnement ubiquitaire et mobile, les sources de contexte sont hétérogènes ainsi que les données captées. Notre service de recherche de contexte prend en compte un mécanisme automatique de transformation des unités et des représentations des données. Le consommateur peut préciser les unités et les représentations souhaitées dans sa requête. Le service va déployer des composants de transformation ("Context Transformer") pour adapter les données de chaque notification ou requête.

Différent capteurs peuvent offrir différents champs de données pour le même type de contexte. Par exemple, certains capteurs GPS peuvent ne fournir que les coordonnées géographiques tandis que d'autres y ajoutent la précision. Un consommateur peut ne pas avoir besoin de la précision mais si le seul capteur disponible est celui qui la fourni, dans ce cas le service de recherche prendra en charge le filtrage des champs de données. Le consommateur indique dans sa requête le format de données souhaité.

10.7.3 *Transformateur de Contexte (Context Transformer CT)*

Il est en charge de traduire les données reçues d'un "ContextCollector" (CC) à l'aide de traducteurs d'unités et de traducteurs de représentation pour renvoyer au consommateur les informations dans la bonne unité et la représentation souhaitée (cf. figure.65).

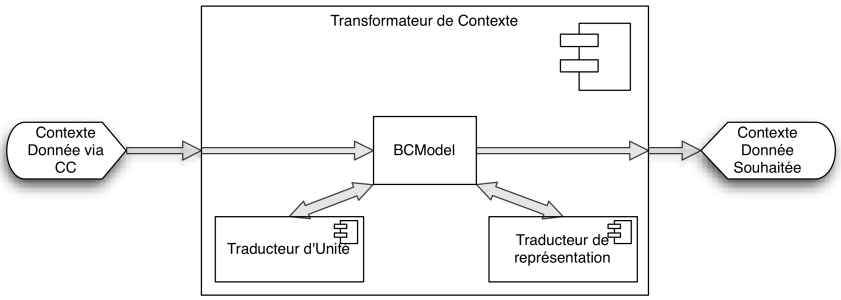

FIGURE 65 – Transformateur de Contexte

Lors du déploiement du composant CT, il faut lui donner le dictionnaire de chaque champ de donnée correspondant à une description de traducteur d'unité et le dictionnaire de chaque champ de donnée correspondant à une description de traducteur de représentation. Le CT se base sur ces deux dictionnaires pour assurer le travail de traduction, lorsque les traductions sont effectuées, les données sont envoyées au consommateur.

10.7.4 *Flux de donnée*

Le service de recherche est piloté par une ontologie (Context Data Source Knowledge, cf. figure.66) qui est enrichie par les gestionnaires d'hôtes (Device Context Manager, cf. chapitre. 11, section 11.5.1). Il utilise également l'ontologie de contexte de la plateforme Kalimucho (KaliCOnto cf. chapitre 9, page 51)

pour trouver les hôtes situés aux lieux indiqués par les requêtes et pour trouver les traducteurs (traducteur d'unité, traducteur de représentation) correspondant aux requêtes.

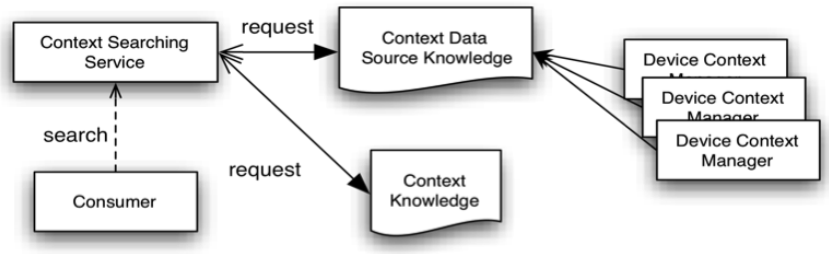

FIGURE 66 – Service de recherche de contexte

Ce service de recherche suit trois étapes principales :

1. Trouver la source de contexte (KaliSensor).
2. Créer un "ContextCollector" connecté au KaliSensor trouvé par la première étape en utilisant les configurations de CC indiquées dans la requête (si la requête est statique le CC sera en mode "Query", si une condition de notification est définie le CC sera en mode "NotifConditionCC", etc.).
3. Créer un "ContextTransfomer" (CT) connecté au CC avec les "Translators" trouvé sur KaliCOnto.

Le service contient deux méthodes, l'une pour les contextes de données statiques, l'autre pour les contextes de données dynamiques et flux. Si la requête est statique le CC, le CT et les "Translators" sont créés lors de la prise en compte de la requête et seront détruits dès que le résultat a été délivré au consommateur. Si la requête est dynamique, le CC, le CT et les "Translators" sont créés comme précédemment mais c'est le consommateur qui a la responsabilité de résilier sa souscription au CT. Le CT sera détruit lorsqu'il n'y a plus de souscripteur.

Nous allons maintenant présenter la base de connaissance de sources de contexte qui est constitué par une ontologie.

10.7.5 *Ontologie de sources de contexte (Context Source Knowledge)*

Cette ontologie décrit le concept "ContextSource" (cf. figure.67). Un "ContextSource" est représenté par un "ContextScope". Le "ContextScope" est importé de notre modèle de contexte (KaliCOnto cf. chapitre 9, page 51). Il représente le type de donnée contextuelle dans notre modèle de contexte (il correspond au ContextScope du méta-modèle). Chaque "ContextScope" a un nom unique dans le modèle de contexte, il suffit donc de fournir dans la requête le nom de ce "ContextScope". Un "ContextSource" peut avoir une valeur de QoC (Qualité de Contexte [63]). Qui pourra servir de critère de recherche. Pour les informations spatiales d'une source de contexte, nous avons identifié deux cas :

1. Capteur intégré à l'hôte
2. Capteur à distance

Dans le premier cas, le capteur est intégré à l'hôte. Ils partagent la même localisation (un capteur connecté à un hôte par un câble USB est considéré comme intégré à l'hôte.) Un tel capteur peut avoir une

géolocalisation et une localisation symbolique – son hôte. Dans le second cas, le capteur est à distance sur un hôte qui peut utiliser la plateforme Kalimucho pour transférer les données. Un tel capteur n'a que sa propre géolocalisation et il ne peut pas partager celle de l'hôte qui fait le transfert de donnée. C'est pourquoi dans le modèle nous définissons le concept de "ContextSource" comme pouvant se situer en un lieu nommé ("Named Place") et/ou se situer sur un hôte disponible. C'est à cela que correspondent les concepts de lieu nommé et d'hôte de notre modèle de contexte.

Un "ContextSource" est offert par un "KaliSensor". Le concept de "KaliSensor" représente les composants logiciels de Kali2Much qui sont des sources de données contextuelles. Il contient des informations de description du composant logiciel qui permettront au service de recherche de leur connecter un Context Collector.

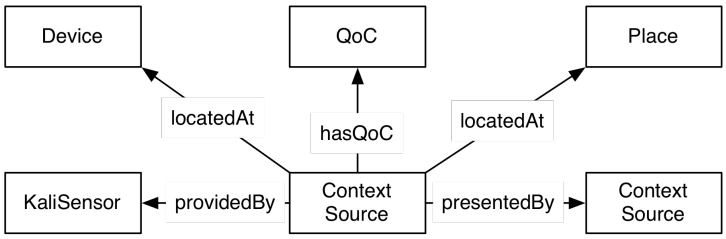

FIGURE 67 – Ontologie de source de contexte (Context Data Source Knowledge)

10.7.6 *Structure de la requête*

La requête est constituée de quatre champs : Quoi, Où, Option de Notification, et Option de Résultat (cf. figure.68). Le champ "Quoi" désigne le type de donnée contextuelle que le consommateur recherche. Dans ce champ, le consommateur peut donner soit une instance de ContextScope trouvé sur KaliCOnto soit le nom d'un ContextScope. La partie "Où" indique le lieu où la donnée doit être produite. Dans ce champ, le consommateur peut donner soit une instance de lieu nommé ("Named Place"), soit le nom d'un lieu, soit le nom d'un hôte, soit enfin une instance d'hôte. L'option de notification "Notification Option" est l'une des quatre configurations de "ContextCollector" possible avec les paramètres associés. Par exemple, pour un "NotifConditionCC", il faut fournir une Condition de Notification dont la définition et l'utilisation ont été décrites dans la partie "ContextCollector" de la section 10.4. Pour un *SimpleNotificationContextCollector*, il faut fournir nul comme valeur. Pour un *RegularContextCollector*, il faut fournir juste la fréquance. Et pour un QueryContextCollector, il faut fournir "QUERY" comme indiquation. Le dernier champ "Résultat Option" contient trois fonctionnalités : Filtrage des champs de donnée, traduction d'unité de donnée et transformation de la représentation du résultat. Les deux premiers font appel à un dictionnaire de signification d'unités. La liste de significations dans ce dictionnaire servira de filtre aux champs de donnée. Les unités correspondant à chaque signification serviront pour trouver les traducteurs d'unités "UnitTranslator". Pour finir, l'instance de représentation permettra de trouver un traducteur de représentation "Representation Translator".

Une fois que la requête est construite, le consommateur va la transmettre au service de recherche. Ce service suivra alors les trois étapes que nous avons décrites précédemment.

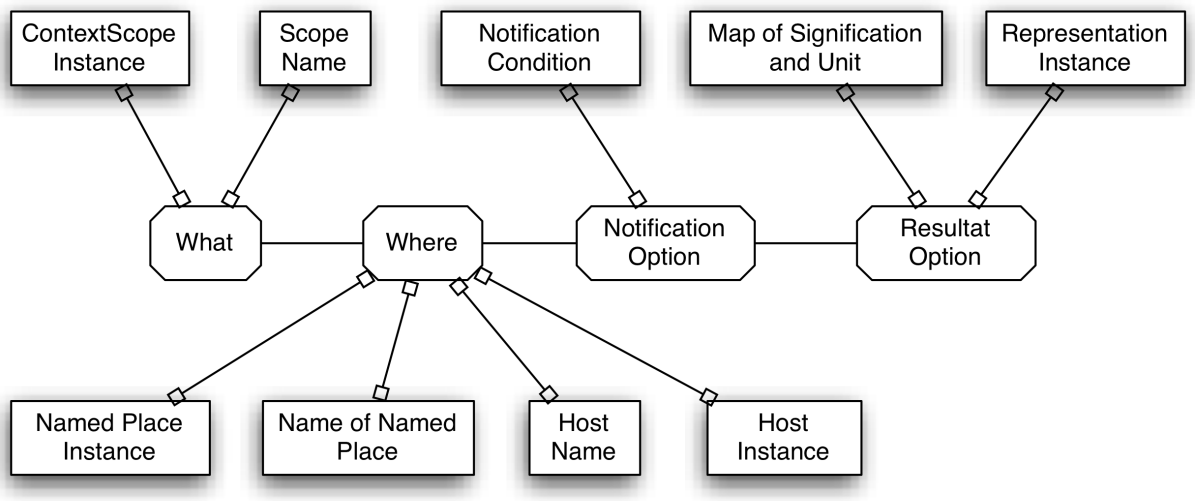

FIGURE 68 – Structure d'une requête de recherche de contexte

Pour la première étape qui consiste à trouver un KaliSensor il utilise le type de donnée contextuelle et le lieu fournis par le consommateur dans sa requête. À partir de ces informations nous pouvons construire une ou plusieurs requêtes en format DL Query (voir exemple suivant). Si les entrées sont des instances de l'ontologie, nous utilisons l'API d'OWL pour récupérer les noms ou les identificateurs pour construire la requête en DL Query. Ce même mécanisme est utilisé dans chacune des trois étapes.

Exemple de requête en DL Query pour trouver un KaliSensor :

```
providOf some (ContextSource  and hasSourceLocation some (ContextSourceLocation and hasLocationName
    value "Room211") and presentOf some ( ContextScope and hasScopeName value "SystemDescriptionScope"
    ))
```

Avant de passer à étape suivante, le service recherchera les "Translators" adéquats, s'il ne les trouve pas le service renverra une erreur au consommateur et terminera le processus de recherche. La figure.69 présente les relations du concept "ContextScope" dans KaliCOnto. Nous utilisons la relation "hasRepresentation" pour trouver une liste de "Representation" valable pour le "ContextScope" demandé et vérifions que la représentation donnée par la requête est valable pour ce "ContextScope". Ensuite nous utilisons le KaliSensor trouvé pour construire une DL Query pour rechercher de la représentation fournie par ce KaliSensor. Cette représentation servira de source pour le "Translator" (il doit y avoir une relation de type "sourceOf" avec ce "Translator") et la représentation donnée par la requête aura comme destinataire le "Translator" (il doit y avoir une relation "has Destination" avec ce "Translator"). Nous pouvons ainsi trouver le "Translator" adéquat. Le processus est exactement le même pour trouver un transformateur d'unités adéquat "UnitTranslator".

La deuxième étape consiste à créer un "ContextCollector" connecté au KaliSensor, comme nous l'avons vu ces deux composants peuvent ne pas se situer sur la même machine. Dès que nous avons trouvé une instance de "KaliSensor", nous utilisons ses informations de description (localisation) pour le connecter. Avant de connecter le KaliSensor au "ContextCollector", il faut configurer ce dernier conformément à la demande qui est indiquée dans la requête (condition de notification).

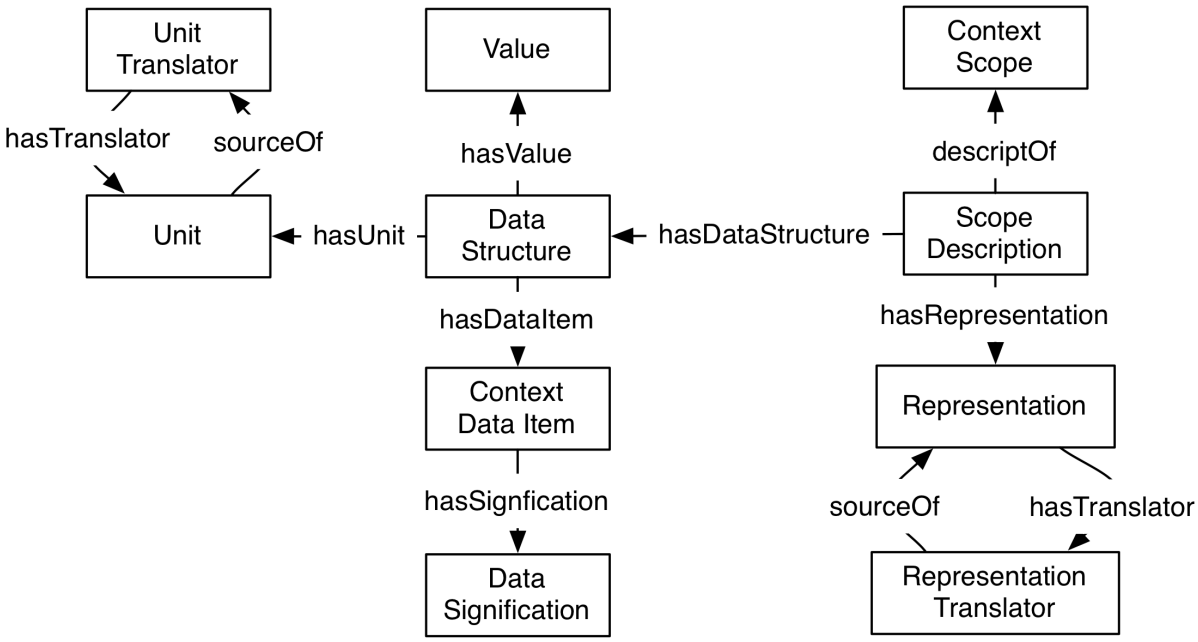

FIGURE 69 – Le Context Scope et ses relations (vue partielle de KaliCOnto)

La troisième étape consiste à créer, à partir des "Translators" découverts à la première étape, un "ContextTransfomer" connecté au "ContextCollector" . Pour créer un "ContextTransfomer" il faut lui donner la description du "ContextCollector" et les descriptions des "Translators". Toutes ces informations ont été préparées dans les deux étapes précédentes. Quand le "ContextTransfomer" est prêt à fonctionner, nous renvoyons au consommateur le résultat de sa recherche. Il s'agit de la description du "ContextTransfomer" lorsque la requête porte sur des données dynamiques ou en flux, tandis que pour les données statiques, le service récupère directement les données auprès du "ContextTransfomer" et les renvoie au consomateur en tant que résultat.

Nous reprenons l'exemple de l'application de contrôle de chauffage (cf. 10.6) pour expliquer l'utilisation du service de recherche de contexte et le rôle de *ContextTransfomer*. L'application utilise le service *ContextCollection* de la plateforme pour acquérir les informations de position de l'utilisateur et la température de la maison.

Grâce au service de recherche de contexte les développeurs d'applications n'ont plus à se préoccuper de la localisation d'un capteur physique mais seulement du type et de la localisation des informations de contexte qu'ils souhaitent recevoir. Ainsi, en cas de non disponibilité du capteur de température habituellement utilisé pour la régulation du chauffage, l'application pourra relancer une requête au service de recherche de contexte qui, si c'est possible, détectera un autre capteur de température dans la maison pouvant remplacer le capteur défectueux.

En pratique, l'application de contrôle de chauffage a besoin de construire trois requêtes pour les trois types d'information : 1) Température de la maison, 2) Distance de l'utilisateur à la maison, 3) Localisation de l'utilisateur. Le résultat dépends de la DL-Query.

Température de la maison :

```
What "Temperature", Where "Home","QUERY",;
```

DL-Query généré par le service :

```
providOf some (ContextSource  and hasSourceLocation some (ContextSourceLocation and hasLocationName
    value "Home") and presentOf some ( ContextScope and hasScopeName value "Temperature"))
```

Localisation de l'utilisateur :

```
What "Location", Where "User","1min",;
```

DL-Query généré par le service :

```
providOf some (ContextSource  and hasSourceLocation some (ContextSourceLocation and hasLocationName
    value "User") and presentOf some ( ContextScope and hasScopeName value "Location"))
```

Distance de l'utilisateur à la maison :

```
What "Location", Where "User","[(Latitude > 43,487330 && Longitude > -1.524318) && (Latitude <
    43,469364 && Longitude < -1.494198)]",;
```

DL-Query généré par le service :

```
providOf some (ContextSource  and hasSourceLocation some (ContextSourceLocation and hasLocationName
    value "User") and presentOf some ( ContextScope and hasScopeName value "Location"))
```

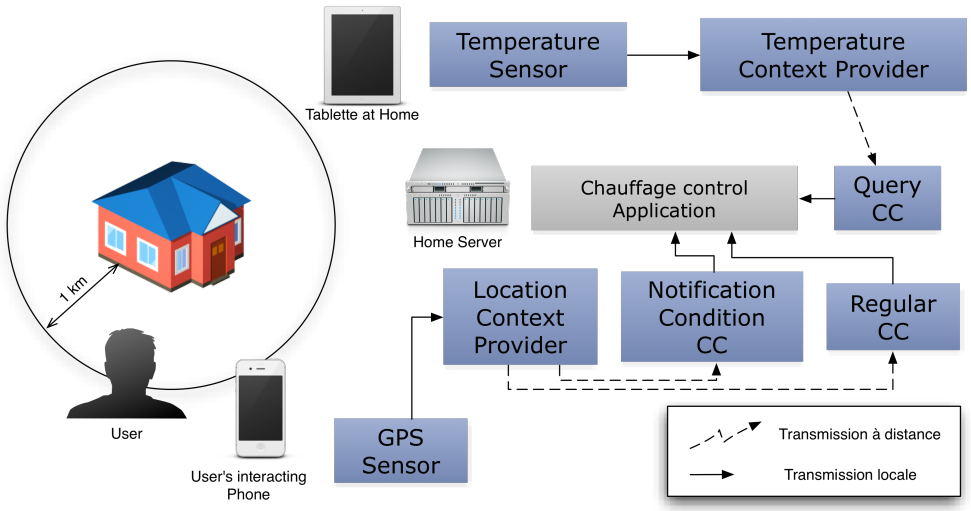

FIGURE 70 – Exemple d'application contrôle de chauffage

Le service de recherche de contexte s'exécute sur une seule machine du domaine de l'utilisateur. Celle sur laquelle se situe le conteneur de DOCK (voir section.9.4). Parce que les critaires de déploiement sont quasiments les même (cf. figure.70). Plus de détail et des exemples voir Appendix.A.5.1. Nous conclusons ce chapitre dans la section suivante.

10.8 CONCLUSION

Dans ce chapitre nous avons présenté notre middleware de gestion de contexte – Kali2Much. L'objectif principal de ce middleware est la collecte des données contextuelles directement mesurées par des

capteurs physiques ou logiques. Ce middleware supporte un fonctionnement multi-consommateurs et permet l'accès à des données captées à distance et le filtrage des champs de données par le consommateur. Il propose aux consommateurs quatre types de "ContextCollector" programmables permettant de réaliser des besoins complexes. L'API unifiée de Kali2Much utilise un méta-modèle pour cacher la complexité et l'hétérogénéité des capteurs de contexte dans un environnement pervasif et mobile.

Dans ce type de l'environnement, en raison de la mobilité et de l'hétérogénéité des dispositifs, il est difficile de prévoir à l'avance quel capteur pourrait être utilisé et à quel endroit il se situe. Au lieu de prévoir des ressources à l'avance, Kali2Much offre un service de recherche qui permet aux consommateurs de trouver les capteurs actuellement disponibles dans un lieu donné. Ce service supporte la recherche de données contextuelles statiques, dynamiques et sous forme de flux. Le langage de requêtes défini permet à Kali2Much de construire automatiquement la chaîne de collecte (KaliSensor + Context Collector + Context Transformer) et de transférer directement les données au consommateur par des connecteurs de la plateforme Kalimucho. Les transformations d'unités et de représentation des données sont automatiquement prises en charge par le service de recherche. Ainsi le consommateur peut se concentrer sur le type de données contextuelles dont il a besoin et leur lieu de production sans avoir à se préoccuper de programmer les capteurs et s'y connecter.

Les données collectées par Kali2Much sont des données contextuelles de bas niveau. Elles sont suffisantes pour configurer les Context Collectors (voir l'exemple du contrôleur de chauffage). En effet, pour configurer les conditions de notification du Context Collector lié aux cordonnées GPS nous devons utiliser des valeurs exactes et précises. Nous ne pouvons pas configurer un Context Collector avec une donnée sémantique, par exemple, "près de chez moi". Toutefois, du point de vue de l'application, il n'est pas très commode d'utiliser des données contextuelles de bas niveau. L'application nécessite généralement moins de précision et plus d'automatisme. Pour disposer de cette souplesse, il faut offrir aux applications des données contextuelles de plus haut niveau d'abstraction. Mais ces données doivent garder des capacités d'interopérabilité et d'inférence. Pour cela nous avons choisi d'utiliser un modèle contextuel sémantique pour simplifier la programmation d'application sensibles au contexte qui sera également utilisé pour la prise de décision d'adaptation dans notre plateforme. Nous présentons ce modèle dans le chapitre suivant.

11

RAISONNEMENT SUR LES INFORMATIONS DE CONTEXTE (KALI-REASON)

Le "Reasoning Chain Engine" (RCE) a pour rôle d'orchestrer les services du raisonneur de contexte (Reasoner Chain) afin de construire un contexte de haut niveau à partir du contexte de bas niveau. Chaque chaîne emploie le service de recherche du contexte (cf. chapitre 10, section 10.7) pour trouver des informations de contexte selon ses besoins de raisonnement. Chaque chaîne reçoit des informations de contexte de bas niveau conformes au méta-modèle (cf. chapitre 10, figure.62, page 84) et va mettre à jour une partie du modèle de contexte "KaliCOnto" (cf. chapitre 9).

11.1 PROBLÉMATIQUE

" *Information from physical sensors, called low-level context and acquired without any further interpretation, can be meaningless, trivial, vulnerable to small changes, or uncertain*" [40]. Nous voyons ici que le problème des informations contextuelles de bas niveau est que les informations collectées directement par des capteurs peuvent être sans signification, triviales, vulnérables à de petits changements ou incertaines. Une façon de résoudre ce problème est une dérivation de plus haut niveau d'informations de contexte à partir de celles capturées directement par les capteurs qui sont, pour nous, des informations de contexte de type d'observation "ElementaryObsType" du méta-modèle, (cf. chapitre 10, figure.62, page 84). Ces informations de haut niveau sont construites par des techniques d'interprétation et de raisonnement logique.

Le service "Reasoning Chain Engine" (RCE) fait face à la problématique de la construction de contextes de haut niveau (type "high-level") à partir d'informations de contexte de type "low-level" au travers d'une chaîne de raisonnement. Chaque modèle de contexte a sa propre logique de raisonnement qui le lie à son domaine spécifique. Il y a donc deux sous problématiques :

1. Comment offrir un service unifié pour toutes les applications ?
2. Comment garantir la construction des chaînes de raisonnement au run-time ?

Le raisonnement sur les informations de contexte est une étape importante pour les applications de type "Context-aware" [26, 45]. Toutefois, le raisonnement peut aussi être une tâche lourde.

11.2 OBJECTIFS

Le service "RCE" doit acquérir les informations de contexte du service "ContextCollection" qu'il doit configurer selon la demande du consommateur. Il doit aussi permettre au consommateur construire des

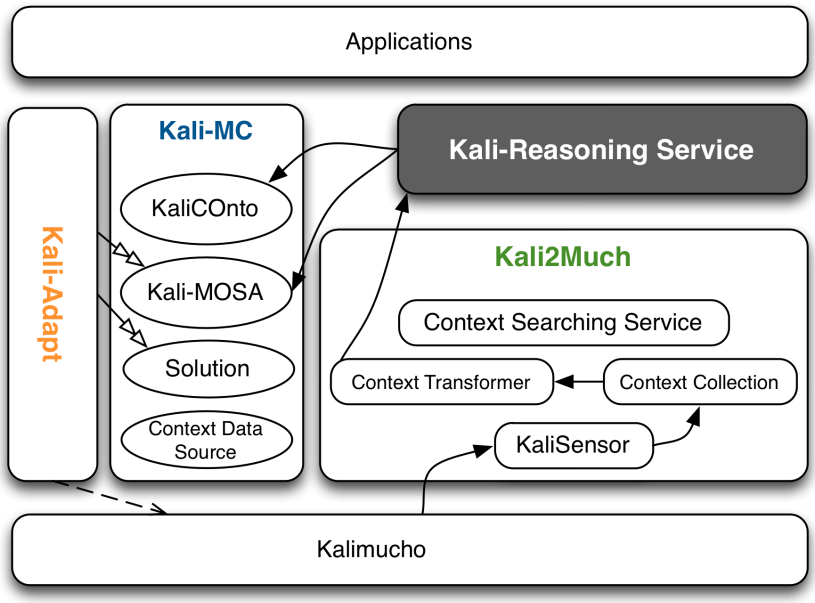

FIGURE 71 – Positionnement du "Reasoning Chain Engine" dans la plateforme

chaînes de raisonnement au run-time, d'être notifié des changements de contexte qu'il spécifie, d'insérer des informations de type "high-level" dans le modèle de contexte étendu par le consommateur et de signaler les incomplétudes des informations de type "low-level" comme les absences de matériel ou de logiciel. Cette abstraction permet ainsi aux applications qui le désirent de, par exemple, raisonner à partir d'un nom de lieu plutôt que de le faire à partir de coordonnées GPS.

Le service "RCE" doit s'inscrire auprès du service "ContextCollection" qui fournit des informations de contexte correspondant à la demande du consommateur. Par exemple : si le consommateur a besoin de la localisation d'un utilisateur, le service "RCE" va s'inscrire après d'un "KaliSensor" qui fournit une cordonnée GPS pour récupérer l'information de géolocalisation qui représente la position actuelle de l'utilisateur.

Pour permettre au consommateur de configurer sa propre chaîne de raisonnement pendant l'exécution, il faut que le service invoque dynamiquement les services de raisonnement (que nous appellerons "Reasoner") pour atteindre l'objectif du raisonnement. Le service doit permettre au consommateur de construire différentes chaînes selon le même modèle en fonction de ses besoins. Chaque information de contexte peut être du type "ElementaryObsType" ou de type "IntrepretedObsType" du méta-modèle. Les services "Reasoner" sont soit fournis par le middleware, soit implémentés par le développeur d'application soit encore des services web. Nous allons illustrer par un exemple l'utilisation de tels raisonnements.

Le consommateur (un consommateur peut être un service d'application ou un service du système) a besoin de savoir quand un utilisateur se dirige vers sa voiture sur son lieu de travail. Une chaîne de raisonnement va être utilisée pour récupérer la position GPS de l'utilisateur via son téléphone portable et la situer sur le plan de son lieu de travail. Elle va pouvoir interpréter cette coordonnée GPS par son nom sur le plan (parking A par exemple). Ensuite elle va utiliser la position GPS de l'utilisateur pour calculer la direction de sa trajectoire. Si la trajectoire indique que l'utilisateur se dirige vers sa voiture, elle va mettre à jour le modèle de contexte pour indiquer que "L'utilisateur quitte son lieu de travail".

Le middleware doit permettre au consommateur de définir ses propres notifications comme par exemple savoir si l'utilisateur va quitter son lieu de travail. Lorsque le consommateur est la plateforme ces notifications lui permettront de prendre des décisions de redéploiement de services de l'application, ainsi, les services énergivores (généralement les services accédant au réseau) peuvent être migrés vers l'ordinateur de sa voiture. Dans ce cas, le consommateur souhaite pouvoir définir la notification suivante : si l'utilisateur sort de sa zone de travail et va vers sa voiture, lever un événement. Cette notification d'événement est réalisée par le "Situation Notification Service" (cf. Apendix.A.5.2).

11.3 SERVICE DE CHAÎNES DE RAISONNEMENT "REASONING CHAIN ENGINE"

En raison des problématiques que nous avons évoquées, le service "RCE" va fournir un service de création de chaînes de raisonnement pendant l'exécution des applications et proposer une façon unifiée de construction d'une chaîne de raisonnement qui se définit comme un ensemble de raisonneurs interagissant entre eux et suivant un raisonnement logique. Chaque raisonneur traite une ou plusieurs informations de contexte qui sont soit de type "Elementary" ou soit de type "Intrepreted".

Une chaîne de raisonnement est essentiellement vue comme une orchestration de services, mais dédiée à l'orchestration de services de raisonnement "Reasoner". Un raisonneur est un service spécifique pour interpréter ou raisonner sur des informations de contexte. Le raisonnement fait appel à une logique métier ("Business Logic") permettant de raisonner sur des informations de contexte de bas et de haut niveau. Nous pouvons donc dire que la gestion du raisonnement est une instance spécifique de la gestion de la logique métier (Business Process Management - BPM). Le BPM est un domaine de recherche qui s'intéresse à "soutenir les processus métier à l'aide des méthodes, des techniques et des logiciels pour concevoir, adopter, contrôler et analyser les processus opérationnels et avec l'implication des humains, des organisations, des applications, de documents et autres sources d'information" [138]

Les logiciels supportant ce type de gestion s'appellent des "Business Process Management Systems" (BPMS). Les BPMS font appel à une ou plusieurs des quatre étapes du cycle de vie d'un BPM (cf. figure.72) : Process Design, System Configuration, Process Enactment, et Diagnosis.

Dans notre cas, nous ne nous intéressons qu'aux Process design, Process enactment et Diagnosis. Nous ne traitons pas l'étape System Configuration qui est spécifique à la gestion des métiers de l'entreprise.

Les avantages d'adopter des travaux existants de ce domaine sont : 1) il existe des standards (BPMN, BPEL, XPDL etc.) ; 2) la facilité de développement (il n'y a pas besoin d'écrire de code car tout peut se faire par des éditeurs graphiques.) ; 3) le dynamisme, la souplesse, l'extensibilité (les langages sont basés sur XML et leur exécution est assurée par un moteur d'exécution qui interprète le fichier XML.)

Dans l'étape de conception, le développeur peut utiliser les éditeurs graphiques **Unified Modeling Language Activity Diagram** (UML AD) [94] ou **Business Process Modeling and Notation** (BPMN) [96]. Les deux sont standards et répandus dans le monde industriel et académique. Notons que dans [111] il est indiqué que "UML ADs are extremely limited in modelling resource-related or organisational aspects of business processes." mais dans le cadre de notre utilisation, ceci ne pose pas de problème.

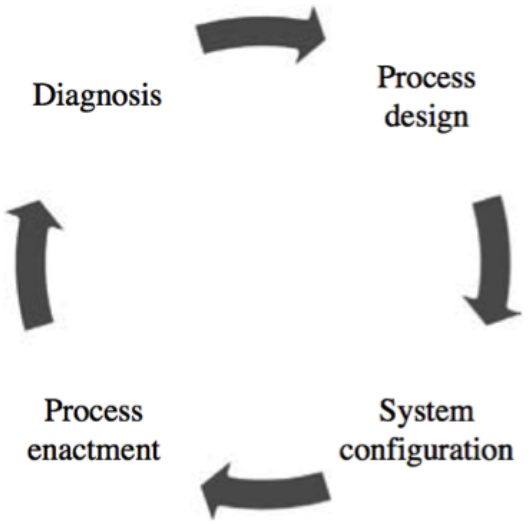

FIGURE 72 – Cycle de vie d'un BPM [138]

Pour l'étape d'exécution, le langage le plus connu est le **Business Process Execution Langage** (BPEL/WS-BPEL) [1] standardisé par l'organisation OASIS. Il été conçu pour orchestrer des web services et est basé sur XML et **Web Service Description Langage** (WSDL) 1.1. BPMN et UML AD peuvent être transformés en BPEL par des outils de transformation [99]. La nouvelle version de BPMN 2.0 (**Business Process Modeling and Notation**) contient une fonctionnalité d'exécution [96]. Dans la spécification de la version 2.0, le mapping entre BPMN et BPEL est assuré sous certaines conditions. BPMN devient un standard de la conception et de l'exécution qui va peut-être remplacer BPEL à l'avenir. BPMN inclut la conception graphique et l'exécution et possède de plus une bonne compatibilité avec BPEL (BPMN peut être traduit en BPEL pour être exécuté dans des moteurs de BPEL). Pour toutes ces raisons nous avons choisi, dans notre middleware, d'utiliser BPMN comme modèle d'orchestration.

Pour l'étape de diagnostic, il existe des standards et des outils comme, par exemple, Business Process Runtime Interface (BPRI) et Business Process Query Language (BPQL). Il existe également des éditeurs graphiques en open source pour concevoir des modèles BPMN et les valider comme : Bonita studio , jBMP et Activiti qui sont les trois plus connus et les plus utilisés dans le monde industriel et sont développés en JAVA.

11.3.1 *Trouver les informations de contexte de bas niveau*

Le service "RCE" utilise le service "ContextSourceSearchingService" pour trouver les informations contextuelles. Ensuite il demande au service "ContextSourceSearchingService" de créer des instances de "ContextCollector" et de "ContextTransformer" selon les besoins. Le service "ContextSourceSearchingService" permet aux clients de chercher des informations contextuelles captées directement par des "KaliSensors" (informations contextuelles de bas niveau) soit par description de contexte ("ContextScope", cf. figure.33 et figure.63), soit par combinaison de différents critères (cf. chapitre.10, section "ContextSourceSearchingService" page 86 pour plus de détails). Une fois que le service "RCE" a trouvé le bon capteur, il

1. BPEL/WS : https ://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=wsbpel

va configurer le "ContextCollector" et le "ContextTransformer". Puis les connecter aux clients qui sont les composants Osagaia (cf. section.6.1 figure.26) constituant les chaînes de raisonnement.

11.3.2 *Construction de la chaîne de raisonnement*

11.3.2.1 *Création de la chaîne*

Pour la création de la chaîne, le développeur d'application peut utiliser un éditeur graphique ou un éditeur de texte pour créer la chaîne sous forme de BPMN. Nous conseillons l'utilisation d'éditeurs graphiques parce qu'ils présentent les avantages d'être visuels, lisibles, de permettre le drag & drop et de générer du code sans fautes de syntaxe.

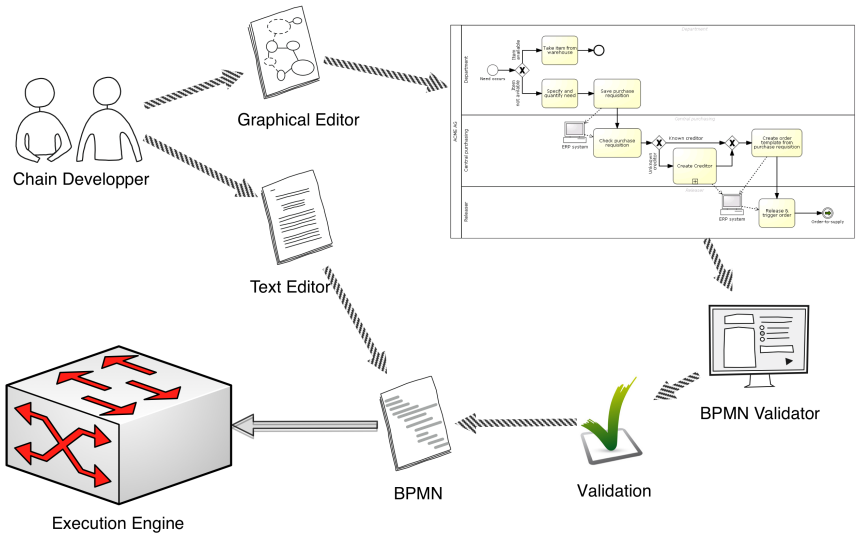

FIGURE 73 – Construction de la chaîne de raisonnement

11.3.2.2 *Entrée et résultat de la chaîne*

Pour chaque service de raisonnement les entrées sont soit les résultats d'un autre service de raisonnement, soit des informations de contexte issues de "ContextTransformer", elles sont toutes du type "ContextMetaDataItem" du méta-modèle. Les résultats de la chaîne sont des données de contexte issues du raisonnement, par exemple, une cordonnée GPS est un lieu nommé ou la température d'une salle pour une personne est "chaude", "froide" ou "tiède". Le dernier "Reasoner" peut avoir recours au service "ContextModelUpdater" pour mettre à jour le modèle de contexte de haut niveau (voir section.11.3.4 Mettre à jour haut niveau contexte modèle).

11.3.2.3 *Validation et Exécution de la chaîne*

Avant l'exécution d'une chaîne dans le moteur d'exécution, il faut que cette chaîne soit validée par le BPMN. Les validations sont intégrées dans les éditeurs graphiques comme Bonita Studio, Activiti etc. L'exécution est réalisée par le moteur de BPMN. Le moteur prend un fichier BPMN comme entrée et orchestre les services indiqués dans ce fichier. (cf. figure.73)

"RCE" lance toutes les chaînes existant dans le système, une chaîne se termine soit par elle-même, soit à la demande de l'application soit quand l'application elle-même se termine (cf. figure.74). Une chaîne peut ne s'exécuter qu'une seule fois et puis se terminer. Elle peut aussi, selon sa logique de raisonnement, s'exécuter en boucle, par exemple, lorsqu'elle attend un type de contexte notifié par un "ContextCollector" pour commencer. Chaque mise à jour de ce contexte va lancer le processus de raisonnement, chaque fin de raisonnement va mettre à jour le modèle de contexte, après la mise à jour on retourne au point initial. Le style d'exécution dépend de la logique de raisonnement, c'est le développeur de la chaîne qui doit gérer et concevoir cette chaîne.

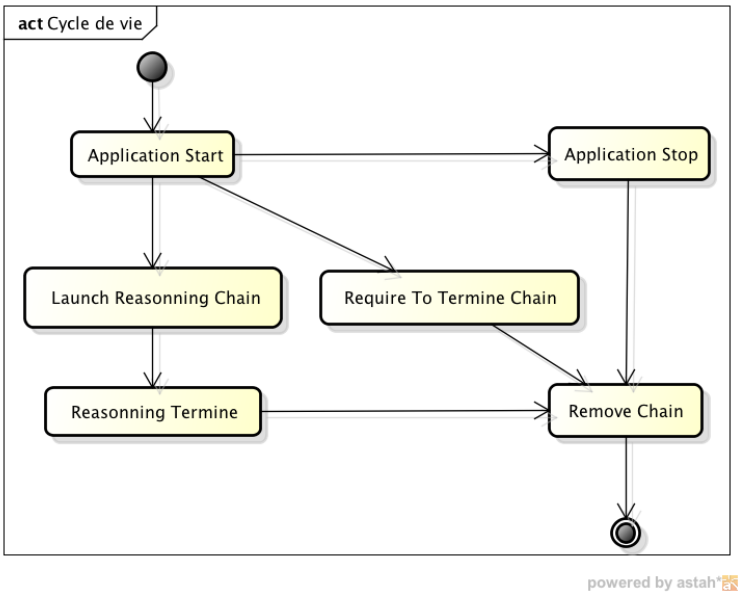

FIGURE 74 – Cycle de vie des chaînes de raisonnement

Dans l'exemple de la section.11.2 page.95, nous utiliserons une chaîne de raisonnement qui contient deux "Reasoners". On suppose qu'il en existe un qui permet de comparer une position GPS avec une zone et retourne le nom de la zone où est située la position. Son entrée est une coordonnée GPS ne contenant que la "Latitude" et la "Longitude". Le second "Reasoner", de positionnement, associe une suite de coordonnées GPS à un nom de zone. Deux "Reasoners" permettent de compléter le traitement : l'un pour calculer la trajectoire du mouvement, l'autre pour calculer la direction de la trajectoire. Le premier "Reasoner" a besoin d'au moins deux positions pour calculer une trajectoire. Il va retourner la direction de la trajectoire (la dernière position associée à une indication de direction). Le deuxième a besoin d'une direction et d'une position. Il va indiquer vers quelle position se dirige l'utilisateur. Pour finir, un dernier "Reasoner" spécifique compare les résultats des "Reasoner" précédents. Si l'utilisateur est sur son lieu de travail et se dirige vers sa voiture, cela signifie qu'il quitte son lieu de travail.

11.3.3 *Différents types de chaînes*

Dans cette section nous présentons trois cycles de vie correspondant aux différents types de chaînes de raisonnement : chaîne type linéaire, chaîne de type boucle et chaîne avec suspension.

Chaîne de type linéaire

Une chaîne de type linéaire commence à s'exécuter depuis un point initial pour se terminer en un point final. Ceci signifie que les chaînes de ce type ne peuvent s'exécuter qu'une seule fois de "Action o" à "Action n" (cf. figure.75) puis se terminent. Dans tout le cycle de vie de l'application, une telle chaîne peut être créée plusieurs fois, mais après chaque fin d'exécution les informations de contexte qu'elle traite ne sont plus prises en compte. Ce type de chaîne est souvent utilisé pour une détection d'activité ou une identification de situation. Par exemple, David, jardinier d'un parc botanique, (cf. Projet ANR MOANO[2]) est assigné à la mission de désherber la zone A du parc. Lorsqu'il prend tous les équipements (contexte n°1) et quitte l'entrepôt (contexte n°2), ceci signifie que la mission est commencée et que David démarre l'activité "désherber". Lorsque David finit sa mission, il quitte la zone A (contexte n°1), retourne à l'entrepôt (contexte n°2) et dépose les équipements (contexte n°3). Il a fini l'activité "Désherber". Dans ce scénario, on peut utiliser deux chaînes indépendantes de type "Linéaire" : une pour détecter le début de l'activité "désherber" et l'autre pour en détecter la fin. Ces deux chaînes sont créées par l'application de gestion de l'entretien du parc botanique dès l'assignation de la mission à David. Quand le début de l'activité de désherbage est identifié, la première chaîne se termine tandis que la seconde se terminera à la fin de cette activité.

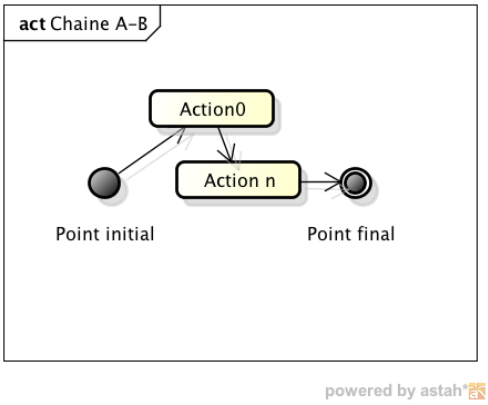

FIGURE 75 – Chaîne de type linéaire

Chaîne de type boucle

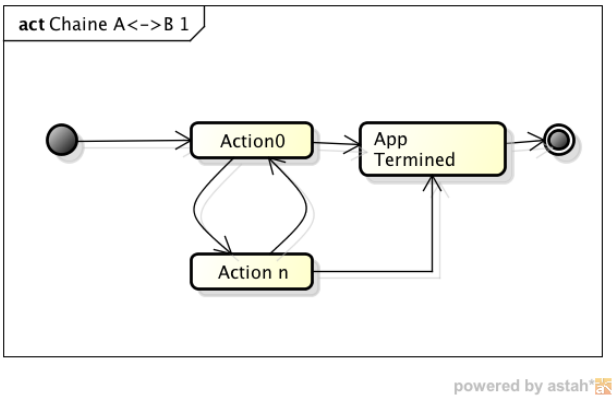

FIGURE 76 – Chaîne de type boucle avec une seule instance

2. http ://moano.liuppa.univ-pau.fr/

La chaîne de type boucle se termine soit à la demande de l'application soit lorsque l'application elle-même se termine. Elle surveille les informations de contexte en permanence. Elle commence à l'action 0, va jusqu'à l'action n puis boucle selon sa logique de traitement. Il existe deux façons de fonctionner pour ce type de chaîne :

— La chaîne boucle avec mais il n'en existe qu'une seule instance (cf. figure.76)

— La chaîne boucle mais il en existe plusieurs instances (cf. figure.77)

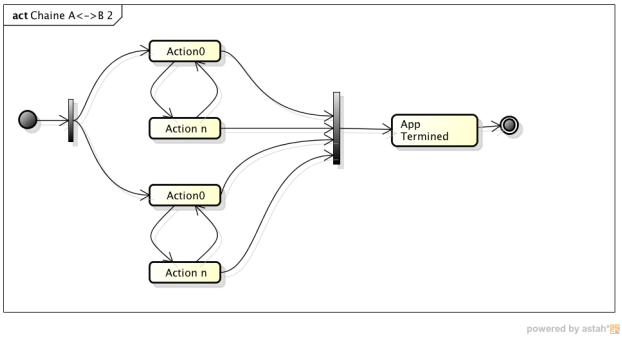

FIGURE 77 – Chaîne de type boucle avec plusieurs instances

Prenons un exemple pour illustrer les utilisations de ce type de chaîne et leurs différences. L'application de contrôle de la température de la serre du parc botanique a besoin de connaitre la température moyenne de la serre (chaîne C1) ainsi que de détecter des zones ayant différentes températures (chaîne C2).

Pour la chaîne C1, l'application définit un déclenchement régulier, par exemple toutes les 10 minutes. A ce rythme C1 va acquérir toutes les informations des capteurs de température de la serre pour calculer la moyenne et obtenir des résultats qualificatifs comme par exemple : "trop basse", "basse", "normale", "haute", "trop haute". Dans ce cas, pour chaque détection, une seule instance de C1 sera exécutée.

La chaîne C2 va également surveiller tous les capteurs de la serre mais ne sera informée que s'il y a des changements de température dans une zone (la surface de la serre est divisée en zones notées, par exemple, "a1", "b3", etc.), C2 va comparer les températures des zones voisines. Comme des changements peuvent être déclenchés en même temps, il est nécessaire que plusieurs instances de C2 puissent s'exécuter en parallèle. Le résultat de C2 peut être par exemple : "normal" - a1, c1, c5, a6 ; "trop basse" - d4, e6, d11, c6 ; etc.

La différence entre ces deux types de chaîne est que la deuxième permet de faire des raisonnements en parallèle pour plusieurs notifications de contexte. Comme elle exécute plusieurs instances en parallèle, les résultats de chaque instance vont être enregistrés sur le même "contexte de haut niveau" (contexte sémantique). Apparaît alors un problème de synchronisation des résultats. Par exemple, l'instance (t1) peut commencer en premier mais finir son traitement après l'instance (t2) (qui a commencé plus tard). Remettre les résultats en ordre est très important pour avoir un résultat sémantiquement correct ("haut niveau contexte"). C'est pourquoi nous avons doté les informations de contexte d'une date (timestamp) dans notre méta modèle (cf. section.6.1).

Chaîne avec suspension

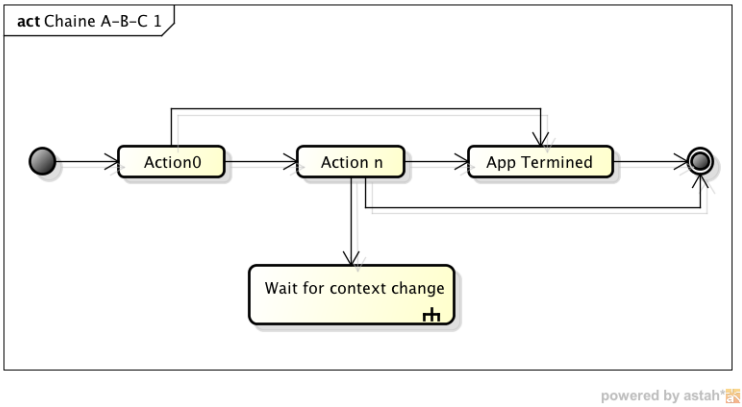

FIGURE 78 – Chaîne linéaire avec suspension

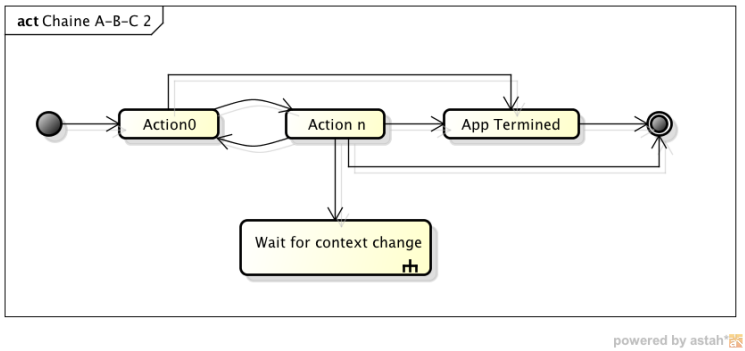

FIGURE 79 – Chaîne boucle avec seule instance et suspension

Les chaînes avec suspension sont des chaînes qui attendent des notifications de contexte ou des changements d'état de certains types de contexte. La suspension peut apparaître dans chacun des types de chaînes que nous venons de présenter, c'est pourquoi nous retrouvons les trois types de chaînes suivants :

— Chaîne linéaire avec suspension (cf. figure.78)
— Chaîne boucle avec une seule instance et avec suspension (cf. figure.79)
— Chaîne boucle avec plusieurs instances et avec suspension (cf. figure.80)

11.3.4 *Mise à jour du modèle de contexte de haut niveau*

Dans notre middleware de contexte "Kali2Much" (cf. Chapitre.10), il existe un dépôt de modèles de contexte qui s'appelle "Device Ontology Container" (détail voir Chapitre.9, page.68). Ce dépôt fournit une API pour retrouver des informations et pour mettre à jour les modèles. Le service "ContextModelUpdater" gère les mises à jour et le respect de leur ordre . Il prend comme entrées des "ContextMetaDataItem" et la référence du modèle et utilise le timestamp pour le respect de l'ordre. Si la dernière mise à jour d'une donnée devait être effectuée après la demande actuelle (le timestamp de la dernière mise à jour est supérieur à celui de la demande actuelle), la mise à jour demandée va être ignorée par le service. Le développeur de la chaîne peut invoquer ce service à la fin de ses chaînes de raisonnement pour effectuer les mises à jour.

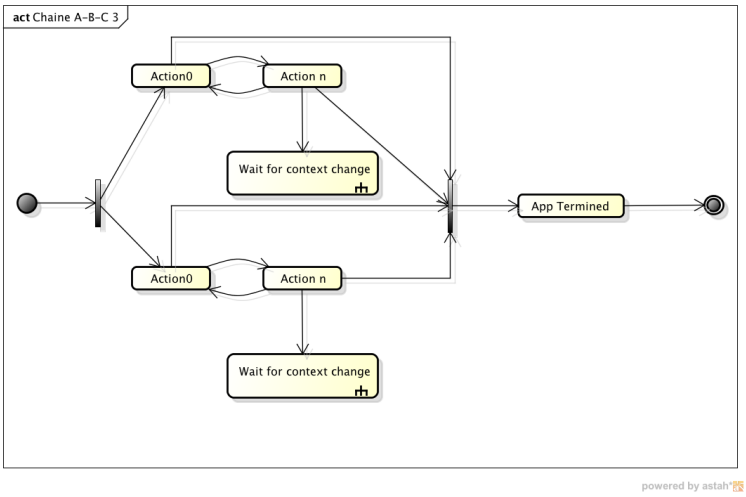

FIGURE 80 – Chaîne boucle avec plusieurs instances et suspension

Dans la section suivante, nous présentons l'implémentation du service de raisonnement "Reasoning Chain Engine" (RCE).

11.4 IMPLÉMENTATION

Nous allons commencer par étudier les moteurs existants de BPMN en lien avec nos besoins c'est à dire à leur intégration dans la plateforme Kalimucho (cf. Chapitre.1 section.1.1.1), puis nous mettons en oeuvre un scénario de cas d'utilisations pour illustrer comment fonctionne le service "RCE".

11.4.1 *Les moteurs BPMN existants*

Il existe plusieurs moteurs d'exécution BPMN. BonitaSoft (Bonita Open Solution), jBPM (jBOSS) et Activiti (Alfresco) sont les logiciels OpenSource les plus courants et qui respectent la spécification BPMN 2.0.

BonitaSoft propose une solution basée sur un outil graphique permettant de définir les processus et fait la génération de code. Néanmoins, son intégration avec Kalimucho est difficile, car il est conçu pour des utilisateurs finaux et n'offre pas d'API pour l'intégration. jBPM et Activiti sont plus orientés développement. Ils proposent une API Java pour configurer, communiquer et contrôler leur moteur d'exécution. Nous pouvons donc les retenir en vue d'une intégration dans Kalimucho.

jBPM est sous licence LGPL tandis qu'Activiti est sous licence Apache. Activiti est développé par Tom Baeyens et Joram Barrez qui sont les anciens fondateurs et développeurs principaux du projet jBPM. Les deux moteurs, très similaires, sont développés en Java. Nous avons préféré utiliser Activiti car il y a une plus grande communauté de développement et de support technique. Le projet est très actif.

La prochaine section présente l'intégration d'Activiti dans la plateforme Kalimucho.

11.4.2 *Intégration d'un moteur BPMN dans la plateforme Kalimucho*

Dans cette section nous présentons d'abord l'architecture générale de l'Activiti engine et ses modes d'exécution. Ensuite nous expliquons comment intégrer et exécuter un "standalone" Activiti engine dans la plateforme Kalimucho. Pour finir, nous expliquons l'intégration de composants Kalimucho dans Activiti engine.

11.4.2.1 *Architecture d'Activiti engine*

Le composant "Activiti Engine" fournit les interfaces du moteur qui met en œuvre la spécification BPMN 2.0. Le moteur comprend également une machine abstraite de processus virtuel qui traduit la logique BPMN 2.0 dans un modèle de machine d'état. Cette machine virtuelle est capable de supporter plusieurs langages de processus et fournit la couche de fondation du moteur Activiti (cf. figure.81). Le composant moteur est implémenté dans le fichier activ-moteur-version.jar.

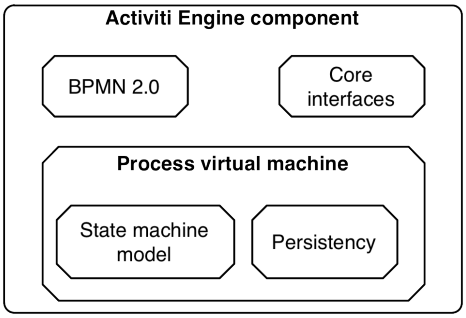

FIGURE 81 – Architecture d'Activiti Engine

Pour exécuter l'Activiti Engine, il existe trois façons de procéder. Dans la première, il peut être exécuté en mode "standalone" avec une base de données "in-memory" (cf. figure.82). Dans ce mode, tous les composants du moteur sont exécutés dans la même JVM (Java Virtual Machine) et font appel à une base de données en mémoire (Base de données Apache H2). Dans la seconde façon, le moteur s'exécute dans une JVM avec une base de données standalone (H2 ou autres). Enfin, dans la troisième, les applications communiquent avec le moteur via une interface de type REST. Le moteur s'exécute dans un serveur d'application, par exemple Apache Tomcat avec une base de données liée à ce serveur d'application.

La section suivante montre comment exécuter l'Activiti Engine dans un composant Kalimucho et présente le mode d'exécution convenant le mieux à notre intégration.

11.4.2.2 *Exécuter Activiti dans la plateforme Kalimucho*

Dans notre cas d'utilisation, nous voulons intégrer le moteur dans un composant Kalimucho pour que notre plateforme d'adaptation puisse le manipuler selon le contexte. Donc le troisième mode ne nous convient pas. Entre la première et la deuxième solution, la différence concerne seulement le type de base de données. Le deuxième mode offre une manière plus souple pour se connecter avec différents types de SGBD. Il est prévu pour une intégration à un système existant ayant une base de données installée.

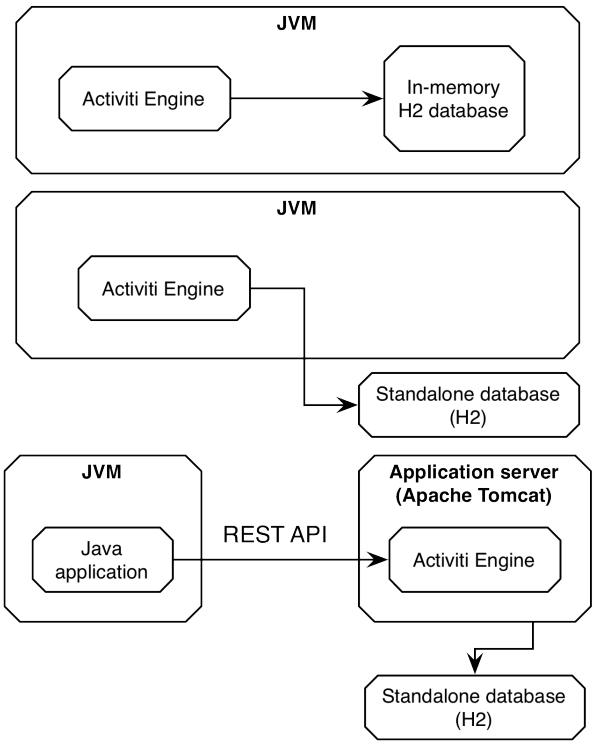

FIGURE 82 – Les trois modes d'exécution de l'Activiti Engine

Dans notre prototype, nous n'utilisons pas de base de données, donc nous sélectionnons le premier mode d'exécution pour l'intégration.

Le modèle de composants utilisé dans Kalimucho (modèle OSAGAIA) a été présenté dans [27]. Les composants sont inclus dans des conteneurs et sont supervisés par la plateforme forme d'exécution (Kalimucho). Elle reçoit de chaque composant de l'application des informations d'état et peut envoyer des commandes à chacun. En outre elle prend en charge la réorganisation dynamique de l'application par création/suppression de composants et redéfinition des interconnexions entre eux. Le composant encapsule la logique métier et le conteneur fournit les propriétés non-fonctionnelles. Ce modèle fait appel à un couplage lâche entre la logique métier et les propriétés non-fonctionnelles comme, par exemple, le contrôle et la qualité de service du réseau.

Nous créerons un composant Kalimucho appelé "BPMNRCEngine". C'est une classe héritant de la classe "BCModel" de Kalimucho. Dans la méthode "run_BC()" nous appelons l'API de l'Activiti Engine pour initialiser le moteur et le mettre en route. Le composant "BPMNRCEngine" n'a qu'une entrée permettant de passer le fichier BPMN au moteur. Chaque chaîne de raisonnement va s'exécuter dans son propre BPMNRCEngine et chaque "BPMNRCEngine" est un composant Kalimucho. L'avantage est que la plateforme peut contrôler et adapter les moteurs pendant leur exécution.

Comme le moteur doit pouvoir créer et lancer des composants Kalimucho, nous utilisons l'API Java Task de l'Activiti Engine pour demander à la plateforme Kalimucho d'initialiser ces composants et de les exécuter.

11.5 UTILISATION PAR LA PLATEFORME KALIMUCHO-A

La plateforme Kalimucho-A utilise ce service pour alimenter l'ontologie par les informations contextuelles dynamique (DECK, cf. page.69) de chaque hôte et pour identifier les situations d'adaptation (cf. Chapitre.12,section.12.4). Dans cette section nous présentons le Device Context Manager et l'utilisation de chaîne de raisonnement pour alimenter le DECK. Les méthodes pour identifier les situations seront présentées dans la section 12.4.

11.5.1 *Device Context Manager (DCM)*

Le Device Context Manager (DCM) se charge, au démarrage, de déployer les *KaliSensors* de l'hôte et les chaînes de raisonnement utilisées par la plateforme. Il y a qu'un seul DCM par hôte. Chaque DCM a une liste de KaliSensors et une liste de chaînes de raisonnement à déployer fournie par l'hôte lui-même. Nous supposons que chaque hôte fournit les deux listes. La liste de chaînes de raisonnement contient deux partie : 1) les chaînes de raisonnement de hôte pour traiter les contextes de haut niveau, 2) les chaînes de raisonnement pour identifier les situations de l'hôte.

Les KaliSensors vont être déployés en premier par le DCM. Ensuite, il vérifie l'état de fonctionnement du DECK. Dés que le DECK est lis en fonction, le DCM déploie les chaînes de raisonnement de l'hôte pour commencer à collecter les informations contextuelles, à raisonner sur les contextes de haut niveau et à identifier les situations de l'hôte.

11.5.2 *Utilisations de chaîne de raisonnement pour alimenter le DECK*

Les chaînes de raisonnement de l'hôte raisonnent sur les contextes de haut niveau. Elles vont enrichir la partie description dynamique du domaine d'ontologie *ComputingResource* (cf. section.9.3.4) et une partie du domaine d'ontologie *User* (cf. section.9.3.2).

Dans le domaine *ComputingResource* (cf. figure.41), nous avons quatre sous domaines : *Hardware*, *Software* (cf. figure.46, *Network* (cf. figure.48), et *Power* (cf. figure.50). Nous présentons maintenant les détails des mises à jour de chacune de ces parties. Puis nous nous intéresserons au domaine User pour lequel les chaînes de raisonnement vont enrichir la partie *Mobility* et la partie *Interaction*.

11.5.2.1 *Le sous domaine Hardware*

Elle concerne l'usage du CPU, de la mémoire RAM, de l'espace de stockage et les états des périphériques intégrées (par exemple caméra activée ou non) pour la prise de décision d'adaptation. Sur chaque hôte sont mises en place des chaînes de raisonnement concernant les informations d'état du matériel. Nous donnons ici l'exemple de la chaîne de raisonnement pour la charge du CPU.

La représentation de la charge du CPU en tant qu'information contextuelle de haut niveau correspond à : "Free", "Normal", "Busy", et "VeryBusy". C'est donc ce type de résultat que produira l'interprétation par la chaîne de raisonnement de la charge du CPU et qui sera inscrite dans l'ontologie locale de l'hôte (DECK). En raison de l'hétérogénéité des systèmes exploitation, les mesures de l'usage du CPU sont différentes. Le

système Android fournit un fichier indiquant la charge du CPU de l'appareil. Nous avons donc créé un KaliSensor qui récupère cette information dans ce fichier (cf. figure.83). Nous lui avons associé un RegularContextCollector avec une fréquence d'une minute qui notifie la chaîne de raisonnement. La première tâche de cette chaîne interprète les informations en termes d'informations contextuelles de haut niveau (composant "CPU Usage Interpreter"). La tâche suivante (composant "Value Checker" compare la valeur actuellement présente dans le DECK pour, le cas échéant, faire la mise à jour de la donnée par l' "Ontology Updater".

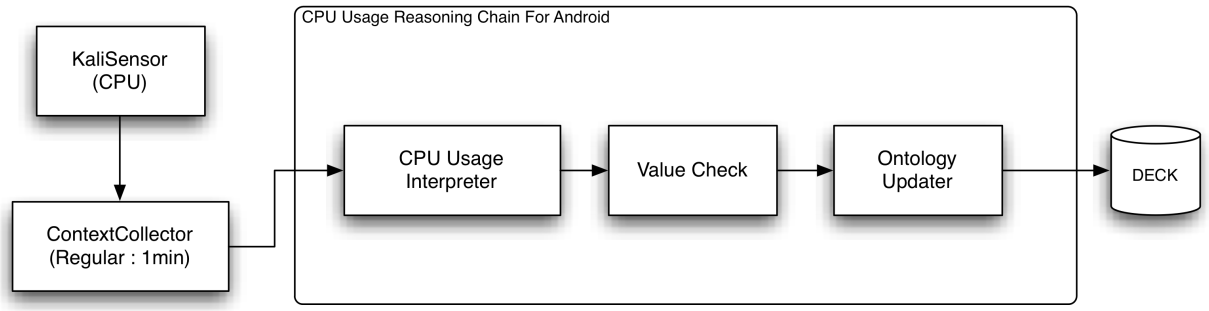

FIGURE 83 – La chaîne de raisonnement de l'usage CPU pour système Android

Pour le système Windows, la situation est plus complexe. En effet, aucun fichier de ce type n'est disponible. Toutefois, l'API Java, fournit des informations de temps d'utilisation du CPU pour chaque processus. Nous avons donc conçu un KaliSensor qui récupère ce temps pour tous les processus et lui avons associé un *RegularContextCollector* qui récupère toutes les minutes les mesures du KaliSensor et notifie la chaîne de raisonnement des données récupérées (cf. figure.84). Il est à noter que, dans ce cas comme dans le précédent, nous n'avons pas besoin de changer d'unité ou de présentation et n'utilisons donc pas de *Context Transformer*. La chaîne de raisonnement calcule, pour chaque processus, la différence entre la nouvelle valeur et l'ancienne pour déterminer la charge totale du CPU (composant "CPU Usage Calculator"). La suite de la chaîne de raisonnement est la même que sur Android.

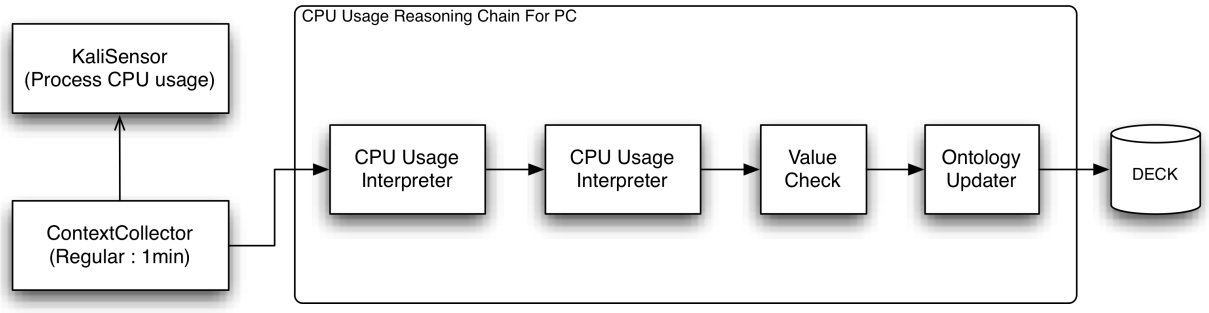

FIGURE 84 – La chaîne de raisonnement de l'usage CPU pour système Windows

Ce principe, décrit pour la charge CPU, est repris pour l'occupation mémoire, l'espace de stockage et l'état des périphériques.

11.5.2.2 *Le sous domaine Software*

La plateforme Kalimucho-A a besoin d'informations sur l'état de fonctionnement des composants Osagaia durant leur exécution. Un composant Osagaia connaît cinq états de fonctionnement : "Running", "Paused", "Stopped", "Execution Issue" et "Exception" correspondant respectivement à un état d'activité normal, un blocage du composant en attente d'une entrée ou une sortie, un arrêt du composant par la plateforme, une terminaison normale du composant et une erreur d'exécution (exception Java). Ces états sont directement captés par l'Unité de Contrôle du conteneur Osagaia qui encapsule le composant métier. Un KaliSensor a été conçu pour récupérer ce type d'informations. Chaque changement d'état d'un composant va être notifié à un Simple-ContextCollector et ensuite à la chaîne de raisonnement de l'état logiciel qui la traduit dans un vocabulaire sémantique et met à jour le DECK. L'état d' "Exception" provoquera une situation de type "Execution Error" qui sera identifié par une chaîne de raisonnement de situation. Nous présentons l'interprétation des états "Exception" et "Execution Issue" dans la section "Problème ou erreur d'exécution" du chapitre KaliSituation (page.125).

11.5.2.3 *Le sous domaine Network*

Le trafic réseau se fait au travers des connecteurs. Le modèle de conteneur de connecteurs *Korrontea* contient une *Unité de Contrôle* qui mesure le nombre de données émise ou reçues par le connecteur. Sur le modèle du *Kalisensor* pour les composants Osagaia, nous avons donc défini un *KaliSensor* pour récupérer ces informations auprès des unités de contrôle de *Korrontea* pour chacun des connecteurs non internes. Un *Context Collector* acquiert les données de ces *KaliSensors* chaque seconde et notifie la chaîne de raisonnement. Cette chaîne de raisonnement calcule les débits totaux en entrée et en sortie puis les convertit en données de haut niveau sémantique. La façon de mesurer est similaire à celle de la charge CPU.

11.5.2.4 *Le sous domaine Power*

Pour les énergies, nous identifions deux types d'alimentations : AC et Battery. L'état de fonctionnement des deux types d'alimentations correspond à trois possibilités : Batterie, Secteur et En charge (l'appareil est alimenté par secteur et la batterie est en charge). AC a deux états : "Active" et "Inactive" tandis que Battery a quatre états : "In charge", "Good", "Low", et "Dead".

Nous avons conçu deux KaliSensor pour les alimentations : un pour AC, l'autre pour Battery (cf. figure.85). Ces capteurs se chargent de récupérer l'état de la source alimentation via l'API de l'OS lorsque c'est possible. Chaque changement d'état sera notifié à un SimpleContextCollector puis à la chaîne de raisonnement. Le composant "Power State Interpretor" vérifie les états reçus et produit l'état d'alimentation de haut niveau sémantique.

11.5.2.5 *Le sous domaine Mobility*

Le sous domaine Mobility décrit l'état de mobilité de l'utilisateur. Il contient : un état (mobile, immobile), une vitesse de déplacement, une direction de déplacement (cf. page.57). Actuellement nous ne réalisons ce sous domaine que sur les hôtes équipés d'un GPS. Un KaliSensor utilise le GPS pour récupérer des positions qu'un SimpleContextCollector acquiert chaque minute. La chaîne de raisonnement va

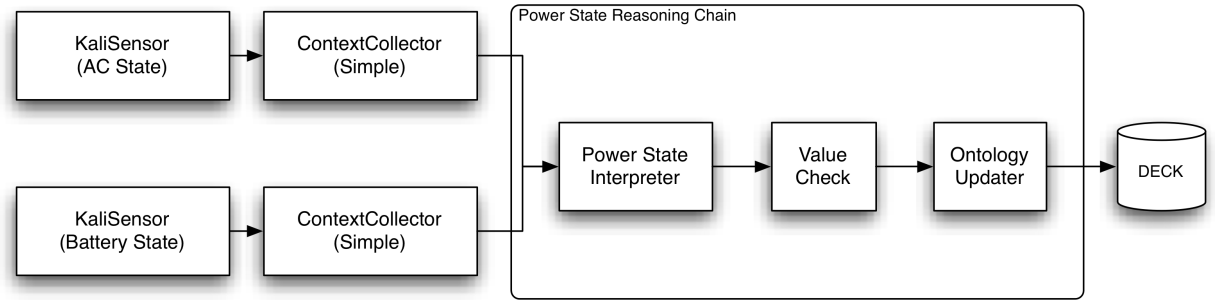

FIGURE 85 – La chaîne de raisonnement de l'état d'alimentation

recevoir ces informations et les comparer avec les anciennes valeurs, s'il y a des changements, l'état "Mobile" sera mis à jour. Le calcul de la vitesse et de la direction de déplacement se font par comparaison de positions successives.

11.5.2.6 *Le sous domaine Interaction*

Nous utilisons également plusieurs KaliSensor pour détecter l'interaction de l'utilisateur avec les hôtes. Pour un PC, nous avons des KaliSensors pour détecter le login, le réveil, l'utilisation du clavier et les mouvements de souris. Pour les dispositifs sous Android, les KaliSensors détectent le login, le réveil et les touchers de l'écran tactile. La chaîne de raisonnement est très simple, s'il y a un changement elle met à jour l'état d'interaction.

Après ce tour d'horizon des méthodes de capture du contexte à usage de la plateforme, nous allons présenter, dans la section suivante, l'utilisation de chaînes de raisonnement par l'application.

11.6 UTILISATION PAR L'APPLICATION

Considérons une application liée à la visite d'un parc dans laquelle on veut pouvoir détecter des situations du type :

— L'utilisateur se dirige vers un point d'intérêt du parc

— L'utilisateur s'éloigne d'un point d'intérêt du parc

Un point d'intérêt peut être un lieu de départ de visite guidée, un musée, un restaurant etc. Lors de la détection de la première situation, la plateforme peut déployer un service spécifique lié à chaque point d'intérêt par exemple la réservation de places pour une visite guidée, le choix du menu pour un restaurant etc. Lors de la détection de la deuxième situation, la plateforme pourra supprimer ce service. L'objectif est de mettre en place les moyens de détecter de telles situations.

Pour commencer l'application doit pouvoir capturer des informations de contexte en utilisant des KaliSensors, ContextProviders et des ContextCollectors – CC. Pour l'application de visite du parc on utilisera les capteurs physiques suivants :

— GPS

— Capteur d'orientation

Et on leur associera les types de ContextCollectors suivants :

```
CCZ: CC de détection de la présence dans une zone donnée (associé au GPS)
CCD: CC de détection d'un déplacement d'au moins x mètres (associé au GPS)
CCP: CC de lecture directe de la position (associé au GPS)
CCR: CC de détection d'un changement d'orientation d'au moins x degrés (associé au capteur d'
    orientation)
CCO: CC de lecture directe de l'orientation (associé au capteur d'orientation)
```

Lors de l'initialisation de l'application de visite du parc va récupérer par un service web auprès du serveur du parc la liste des zones contenant un point d'intérêt. Ensuite les opérations suivantes seront réalisées :

```
Créer un CCZ associé au GPS pour chacune de ces zones
Créer un CCD et un CPP associés au GPS
Créer un CCR et un CCO associés au capteur d'orientation
```

Le fonctionnement des ContextCollectors est le suivant :

```
Les CCZ envoient une valeur (ENTREE / SORTIE) quand l'utilisateur rentre ou sort de la zone définie
Le CCD envoie la position GPS s'il y a eu un déplacement de plus de x mètres
Le CCP envoie la position GPS quand on le lui demande
Le CCR envoie l'orientation par rapport au nord s'il y a eu un changement d'orientation de plus de x
    degrés
Le CCO envoie l'orientation par rapport au nord quand on le lui demande
```

L'application de visite du parc utilisera la chaîne de raisonnement suivante :

— Une étape détecte l'arrivée dans une zone contenant un point d'intérêt

— Une étape détecte l'approche d'un point d'intérêt et créée une situation (1^{er} événement de haut niveau du scénario)

— Une étape détecte l'éloignement d'un point d'intérêt et créée une situation ($2^{ème}$ événement de haut niveau du scénario)

Détail des étapes de la chaîne de raisonnement :

Etape 1 :

```
1.      Activer les CCZ
2.      Attendre que l'un des CCZ envoie la valeur ENTREE (l'utilisateur est entré dans la zone Zi)
3.      Passer à l'étape 2
```

Etape 2 :

```
1.      Désactiver les CCZ sauf celui qui a détecté l'entrée dans la zone
2.      Activer CCD, CCP, CCR et CCO
3.      Attendre une valeur venant de CCZ ou de CCD ou de CCR
4.      Si c'est CCZ qui a envoyé la valeur SORTIE désactiver CCD, CCP, CCR et CCO puis revenir à l'é
    tape 1 (l'utilisateur a quitté la zone sans s'être dirigé vers le point d'intérêt)
5.      Si c'est CCD ou CCR qui ont envoyé une valeur (l'utilisateur se déplace ou change d'
    orientation dans la zone)
6.      Récupérer la position et l'orientation avec CCP et CCO
```

```
7.      Envoyer ces valeurs à un web service du serveur du parc. Ce WS connaît le plan de parc et les
    points d'intérêt, il calcule si les valeurs transmises indiquent une position et une direction
    dirigées vers le point d'intérêt de la zone. Si oui il renvoie l'identifiant de ce point d'intérêt
     sinon il renvoie une valeur indiquant qu'aucun point d'intérêt n'est désigné.
8.      Si la réponse du web service est un identifiant créer une situation de type: "l'utilisateur se
     dirige vers le point d'intérêt P" puis passer à l'étape 3. Sinon revenir au 3 de cette étape.
```

Etape 3 :

```
1.      Désactiver CCD, CCP, CCR et CCO
2.      Attendre que le CCZ envoie la valeur SORTIE
3.      Créer une situation de type: "l'utilisateur quitte le point d'intérêt P"
4.      Revenir à l'étape 1
```

Cette chaîne créée bien les deux situations (arrivée et départ d'un point d'intérêt) qui pourront être utilisées par la plateforme pour déployer/supprimer les services relatifs aux points d'intérêt du parc.

11.7 CONCLUSION

Dans ce chapitre nous avons présenté notre service de raisonnement de contexte. Nous utilisons des chaîne de raisonnement pour réaliser les interprétations de contextes. Les raisonnements logiques fournis par OWL permettent de vérifier la cohérence des concepts et de raisonner (classer) sur les données mais pas de faire des raisonnements plus complexes qui demandent de calculer ou de communiquer avec d'autres services (par exemple un WebService pour récupérer la carte d'un lieu que l'on souhaite comparer à la position GPS de l'utilisateur). Notre architecture permet aux applications d'étendre les ontologies et de mettre en place des chaînes de raisonnement pour traiter les informations de contexte au niveau sémantique adéquat.

L'implémentation est faite par le standard de l'orchestration de services qu'est BPMN dont nous avons intégré un moteur d'exécution BPMN. Ensuite nous avons expliqué comment utiliser un moteur d'exécution de BPMN dans la plateforme de Kalimucho-A. Les tâches déclenchées à partir des indications fournies par les fichiers BPMN sont réalisées par des composants Osagaia ce qui permet à la plateforme Kalimucho d'en assurer le contrôle total. Ainsi c'est elle qui les créer, les interconnecte, les lance, les arrête et les supprime. Par ailleurs, faisant partie des composants métiers gérés par la plateforme, ces Raisonneurs peuvent être inclus dans les processus d'adaptation. Ils peuvent, par exemple, être migrés si les ressources de l'hôte qui les accueillent se révèlent insuffisantes à un moment.

Dans la chapitre suivante "KaliSituation", nous présentons notre modèle de situations sémantiques "KaliMOSA". Nous avons également implémenté ce modèle autour d'une ontologie et utilisons des chaînes de raisonnement pour identifier les situations d'adaptation. La prise de décision de l'adaptation est ensuite basée sur ces situations d'adaptation.

12

KALI-SITUATION : IDENTIFIER LES BESOINS D'ADAPTATION

Ce chapitre s'intéresse à la détection par le système des moments où doivent se faire des adaptations. Nous avons vu que, grâce aux raisonnements faits par les chaînes de raisonnement, les informations contextuelles de bas niveau sont devenues des informations contextuelles de haut niveau. Ces informations de haut niveau sont stockées dans notre ontologie de contexte (KaliCOnto) précédemment vue. C'est par des raisonnements sur cette ontologie que les situations d'adaptation sont identifiées. Chacune déclenchera une notification au décisionnaire qui prendra en charge la décision d'adaptation.

12.1 PROBLÉMATIQUE

Les informations de haut niveau vont nous permettre de détecter les adaptations nécessaires et de les mettre en œuvre, ce qui revient à répondre aux deux questions suivantes :

1. Quand avons-nous besoin de réaliser une adaptation ?
2. Comment le détecter ?

Chaque activité de l'utilisateur suppose l'accès aux services applicatifs composant l'application. L'obtention de ces services n'est pas traitée ici. Le lecteur intéressé pourra se référer à la thèse de Ghada Ben Nejma ou aux articles suivants[20, 19] . Chaque service correspond à une configuration, c'est à dire à un déploiement d'un ensemble de composants.

La figure 86 présente la façon de transformer les besoins de l'utilisateur à un moment donné en une configuration possible d'application. Cette configuration est contrainte par les composants logiciels et les ressources matérielles actuellement disponibles. Dès que les besoins de l'utilisateur ou les ressources changent, une reconfiguration peut s'avérer nécessaire.

Le déploiement des services applicatifs est lié à certaines contraintes. La première est bien évidemment de connaître la configuration à déployer, c'est-à-dire l'assemblage de composants de manière à connaître les dépendances entre eux puis d'obtenir les archives (fichiers Jar) contenant le code de chacun d'entre eux dans les entrepôts de composants [42].

Le déploiement d'un composant est également lié à des contraintes d'exécution dues aux dépendances matérielles : quantités de CPU/RAM/bande passante nécessaires, périphérique cible spécifique, présence de certains capteurs comme le GPS, etc.

Enfin, le déploiement d'une configuration (assemblage de composants) ne peut s'opérer que sur un ou plusieurs périphériques disposant des ressources nécessaires, conformément aux contraintes exprimées

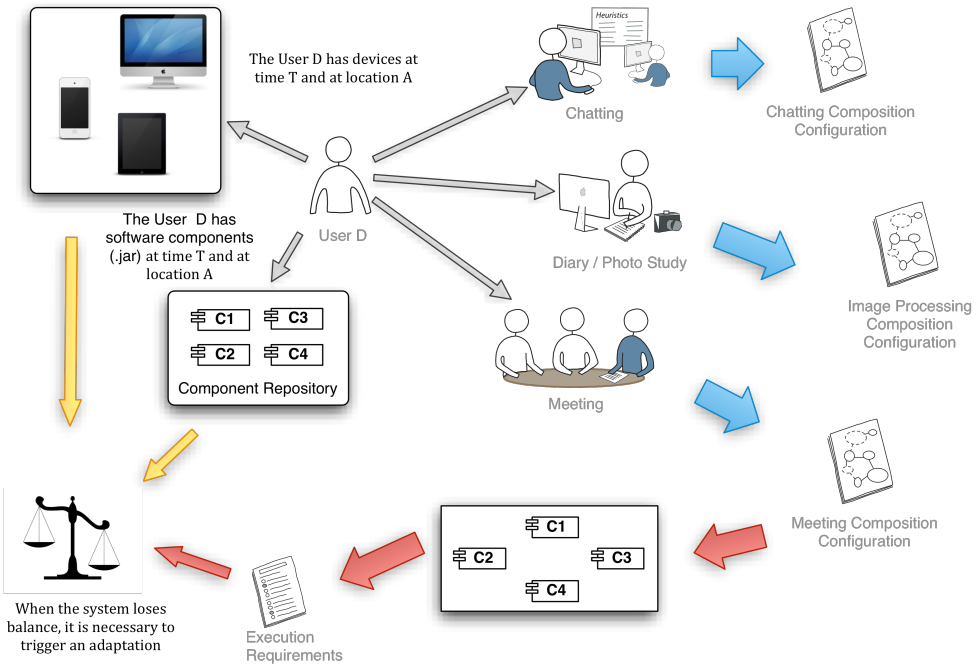

FIGURE 86 – La transformation des besoins de l'utilisateur en configurations

ci-dessus. Le cas où il n'est pas possible de réaliser le déploiement constitue également une situation nécessitant une adaptation.

Une fois le déploiement réalisé, l'étape suivante est l'exécution (Runtime). Le paradigme composant réduit énormément les problèmes de dépendances, car par nature, les dépendances (fichiers de ressources) sont intégrés dans l'archive de ce dernier. Néanmoins, lorsque l'on souhaite aller vers de l'adaptation encore plus fine, il est envisageable de travailler sur l'adaptation du composant lui-même lorsqu'il tente d'accéder à des ressources non accessibles. Néanmoins ceci relève de l'adaptation du composant et sort du cadre de nos travaux. Le lecteur trouvera plus d'informations sur [14, 18].

Pour terminer, l'exécution d'un composant ne peut se poursuivre qu'à la condition que les contraintes d'exécution (RAM/localisation, etc.) soient maintenues. Dans la négative, il sera nécessaire de reconfigurer. Donc la disponibilité des ressources est l'un des critères d'identification du moment d'adaptation. Nous appelons les situations (cf. page.24) d'adaptations liées à ce critère "Situations d'adaptation générales". Nous avons identifié d'autre critères d'adaptation comme : la logique d'usage d'une application (situation spécifiée par application) et la logique de fonctionnement de la plateforme (situation du type arrivée d'un utilisateur). Ce genre de situations est, bien entendu, plus complexe à détecter que les situations générales. Ces situations sont liées à l'activité de l'utilisateur, au comportement d'un composant spécial ou à des enchainements particuliers de situations. Les situations d'application doivent pouvoir être définies par développeur de l'application tandis que nous avons défini les situations types pour la plateforme. Toutefois cette liste de situations peut évoluer dans le futur, c'est pourquoi notre plateforme doit fournir des mécanismes pour étendre les situations d'adaptation et en permettre l'identification tant pour la plateforme elle-même que pour les applications.

Dans la figure 87 nous rappelons l'architecture de la plateforme Kalimucho-A et la positionnement du modèle de situation KaliMOSA dans la plateforme. Le modèle KaliMOSA sera le modèle de situation qui

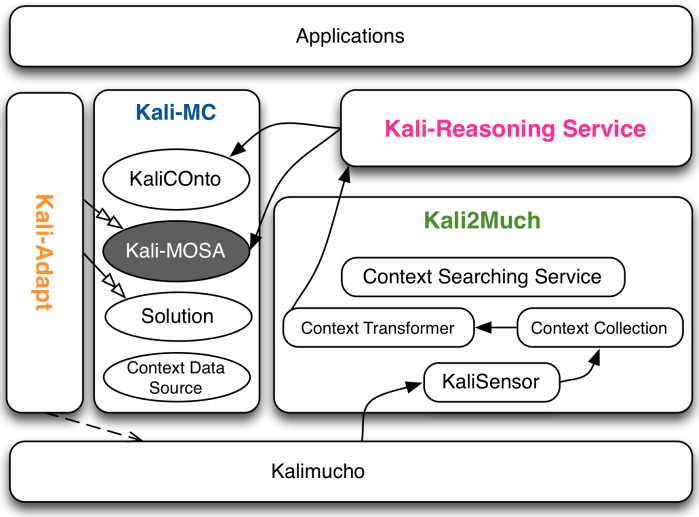

FIGURE 87 – Positionnement de "KaliMOSA" dans la plateforme

Nom de situation	**Description**
User Lost	Utilisateur perdu
User Isolated	Utilisateur isolé
Adaptation Service Lost	Perte du DOCK et du Decision-Maker
Adaptation Service Duplicated	Dupliquation du DOCK et du Decision-Maker dans le même domaine d'adaptation
PF Service Dysfunction	Dysfonctionnement d'un service de la plateforme

TABLE 9 – Liste des situations de la plateforme

va être présenté dans la section 12.3. L'objectif de ce chapitre est de présenter les entités mises en œuvre pour répondre aux deux questions qui sont quand adapter l'application et comment le savoir. Nous présentons aussi les situations identifiées pour la plateforme et le mécanisme d'extension permettant l'identification de nouvelles situations. La table 10 indique la liste des situations d'adaptation ("GeneralASituation") que nous avons idenfiées. La table 9 contient la liste des situations de la plateforme ("PlatformASituation"). Les applications peuvent, ils pouvent étendre la partie "AppASituation" pour leur usage spécifique.

Nom de situation	Description	Nom de situation	Description
Deploy Software	Déploiement de logiciel (application, service, composant) demandé par un utilisateur ou un logiciel.	New Device	Nouvel Hôte trouvé
Stop Software	Arrêt de logiciel (application, service, composant) demandé par un utilisateur ou un logiciel.	Device Lost	Hôte disparu
Software Dysfunction	Occurence d'une exception ou non satisfaction des exigences durant l'exécution du logiciel	New Peripherique	Nouveau périphérique trouvé
Reduce QoS	Nécessité de baisser la QdS	Peripherique Lost	Périphérique disparu
Raise QoS	Possibilité d'augmenter la QdS	New Network	Nouveau réseau trouvé
Low Energy Power	L'hôte est sur batterie et la batterie est faible	Low Computing Power	CPU est très occupé ou manque de RAM
Network Lost	Réseau disparu	Computing Power Raise	Charge de CPU et charge de RAM reviennent normaux
Low Connection	Débit réseau faible	User Mobility Change	Utilisateur mobile ou changement de situation de mobilité
Energy Power Stable	Alimentation stable	User Interaction Change	Changement dans les interactions de l'utilisateur avec l'hôte
Energy Power Instable	Alimentation instable, par exemple, connection AC instable, changement de mode d'alimentation trop fréquent		

TABLE 10 – Liste des situations d'adaptation

12.2 DÉTECTION DES SITUATIONS GÉNÉRALES

Les utilisateurs souhaitent pouvoir trouver sur leur périphérique les applications correspondant à leurs besoins du moment. Ce sont donc les activités quotidiennes des utilisateurs qui définissent ces besoins. Dans l'optique de pouvoir offrir des applications qui suivent l'utilisateur et évoluent avec lui dans ses activités quotidiennes (principe des "long life" ou "eternal" applications), il est souhaitable de pouvoir répondre de façon automatique à ces changements d'activité. A l'heure actuelle les Smartphones ou les tablettes permettent une grande variété de types d'utilisation mais à la condition que l'utilisateur trouve et télécharge autant d'applications que d'activités souhaitées. Notre objectif est que la plateforme sélectionne des composants logiciels disponibles et les assemble en services constitutifs d'applications en réponse aux besoins des utilisateurs. Par ailleurs, la notion de domaine de l'utilisateur qui comprend la totalité des ressources auxquelles il peut accéder permet de faire en sorte que l'utilisateur dispose d'une machine virtuelle sur laquelle s'exécute une application qui se modifie en temps réel en fonction du contexte et de ses besoins.

Les services d'une application peuvent avoir plusieurs configurations d'implémentation possibles correspondant à des qualités de service différentes. Chaque configuration possède ses propres exigences en termes de ressources logicielles et matérielles. La présence de ces ressources est fondamentale pour l'exécution de ces services. De sorte que, de même que les changements d'activité de l'utilisateur, les modifications de présence de ces ressources peuvent provoquer des reconfigurations de services. D'autre part, l'environnement peut, lui aussi, influer sur les disponibilités des ressources, par exemple, dans un bâtiment l'utilisateur peut perdre la connexion internet.

Enfin, lorsqu'un service s'exécute il peut demander des ressources supplémentaires ou en libérer. L'application induite par l'activité de l'utilisateur constitue, elle-même une source de variation des besoins en ressources. La figure 88 présente les relations entre ces entités.

Nous avons ainsi identifié trois origines possibles de reconfiguration :

1. L'utilisateur
2. L'application
3. La présence des ressources matérielles et logicielles

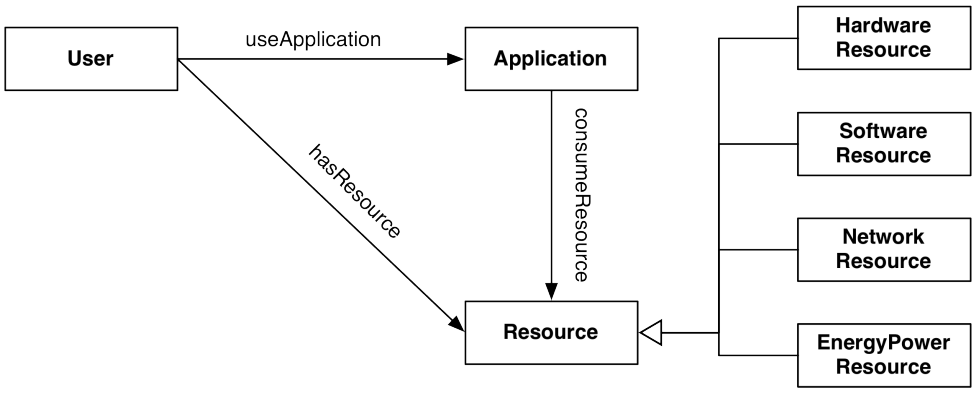

FIGURE 88 – Les relations entre les trois origines de reconfiguration

12.2.1 *L'utilisateur*

Au travers de son interface, l'utilisateur peut, bien entendu, demander de nouveaux services ou en arrêter certains. Par exemple, une application de chat peut proposer une communication écrite, vocale ou vidéo. Chacun de ces changements provoque l'installation ou la suppression d'un service complet c'est à dire d'un assemblage de composants et de liens entre ces composants.

Par ailleurs, la mobilité est, elle aussi, source de modifications du contexte d'exécution. Ainsi le déplacement de l'utilisateur peut se traduire par une baisse ou une hausse du débit de sa connexion réseau. Contrairement aux demandes explicites de l'utilisateur en termes de services à ajouter ou enlever, la mobilité se traduit par une modification de disponibilité des ressources. Ce cas rejoint donc celui examiné plus loin (cf. section.12.2.3).

12.2.2 *L'application*

Au cours de son exécution l'application peut rencontrer des problèmes d'exécution concernant les données manipulées. Certains composants métiers peuvent ne pas parvenir à traiter les données qu'ils reçoivent au rythme voulu (problème de traitement). Il peut arriver également que ces données ne parviennent pas à circuler suffisamment rapidement d'un composant à un autre (problème de transmission).

Par ailleurs, au cours de son exécution, l'application peut être amenée à demander l'installation de services supplémentaires ou la suppression de services devenus inutiles. Si la plateforme se charge d'effectuer les reconfigurations rendues nécessaires par les problèmes de fonctionnement évoqués précédemment, elle n'intervient pas dans la logique métier des services installés. En revanche elle peut servir de support aux demandes de reconfigurations produites par l'application.

Enfin l'exécution de l'application peut provoquer des erreurs rendant impossible son fonctionnement. Dans un tel cas la plateforme peut, si elle a connaissance de ces erreurs, tenter d'assurer la continuité du service par le remplacement du service complet ou du composant défectueux.

12.2.3 *La présence des ressources matérielles et logicielles*

Concernant les ressources matérielles nous distinguerons deux cas. Une ressource :

— peut être ou non disponible

— peut voir sa qualité diminuer ou augmenter

Nous ne parlons pas ici de ressources matérielles non disponibles par construction du périphérique mais bien de ressources devenant indisponibles. En effet, si un périphérique ne dispose pas de GPS par exemple cette information est connue en amont par la plateforme qui ne déploiera pas sur cette machine de configuration en demandant la présence. En revanche, si l'utilisateur est en un lieu ne permettant le fonctionnement de son GPS, il faut pouvoir réagir en proposant une reconfiguration dans laquelle ce capteur ne sera pas nécessaire. On pourra, par exemple, envisager une reconfiguration déployant le composant nécessitant un accès au GPS sur un périphérique voisin appartenant au domaine de l'utilisateur (cf. page.5) s'il en existe un. De la même façon, lorsqu'une ressource (re-)devient disponible, il est possible

de proposer un service de meilleure qualité. Dans le cas précédent, le fait d'utiliser le GPS local plutôt qu'un GPS voisin offre une meilleure précision de positionnement.

D'autre part, à certaines ressources matérielles (batterie/mémoire/CPU/réseau) peuvent être associées des mesures de présence qui ne sont pas binaires. On prendra en compte la taille de mémoire libre, la charge de la batterie, la charge du CPU et la bande passante du réseau utilisée. Ces mesures permettent de guider les choix de déploiement en évitant d'installer un composant sur un périphérique dont la disponibilité de l'une des ressources est trop faible. Par exemple un composant de traitement peut être migré si la charge CPU du périphérique qui l'héberge baisse. Par la suite nous utiliserons le terme de "qualité de ressource" pour désigner ces mesures de disponibilité tandis que le terme de "présence de ressource" sera réservé au cas de présence ou d'absence d'une ressource.

Enfin, dans la mesure où ces valeurs changent en cours d'exécution, des reconfigurations peuvent être provoquées aussi bien par une baisse que par une hausse de disponibilité de ressource.

En ce qui concerne les ressources logicielles seul l'aspect de présence est pris en compte (valeur binaire). Ainsi, un composant métier peut, pour fonctionner, nécessiter l'accès à une ressource logicielle comme un service web ou une activité Android de scan de QrCode sur un Smartphone alors que cette ressource pourrait ne pas être accessible sur le périphérique sur lequel il s'exécute.

12.3 MODÈLE DE SITUATION D'ADAPTATION (KALI-MOSA)

Dans cette section nous présentons notre modèle de situation. Ce modèle de situation est conçu pour représenter les situations d'adaptation. Nous utilisons encore une fois une ontologie pour concevoir ce modèle parce que nous voulons une représentation sémantique facile à étendre et aussi un outil adapté aux raisonnements et partager la même structure de connaissance de contexte de KaliCOnto. Une situation d'adaptation est soit détectée par un défaut d'adéquation de ressource, soit identifiée par l'application, soit identifiée par la plateforme. Une situation d'adaptation est un résultat des états du contexte actuel. Elle peut être simplement causée par une seule condition de contexte, par exemple si l'occupation de la mémoire RAM est supérieure à 90%, il s'agit d'une situation de "Low Computing Power". Elle peut également dépendre d'autres événements, par exemple, une situation correspondant à un déploiement de service peut provenir du fait que l'application demande un nouveau service ou provenir de l'apparition d'un nouveau périphérique dans le domaine de l'utilisateur ou encore d'une erreur d'exécution d'un composant. Notre objectif est de concevoir ce modèle pour supporter l'identification des situations définies par l'application et de celles définies par la plateforme.

Une situation d'adaptation dans Kali-MOSA (cf. figure.89) correspond à l'une des trois grandes catégories suivantes : "GeneralASituation", "PlatformASituation" et "AppASituation". Les situations générales ("GeneralASituation") correspondent à un déséquilibre entre les besoins de l'utilisateur et les ressources disponibles dans l'environnement actuel. Les situations d'adaptation de la plateforme ("PlatformASituation") sont liées à la structure et la logique de fonctionnement de la plateforme elle-même. Les situations d'adaptation de l'application ("AppASituation") sont seulement détectées pour être transmises à l'application qui les traitera selon sa logique métier. Les catégories "PlatformASituation" et "AppASituation" sont extensibles. La plateforme et les applications pourront étendre ces situations selon leurs besoins.

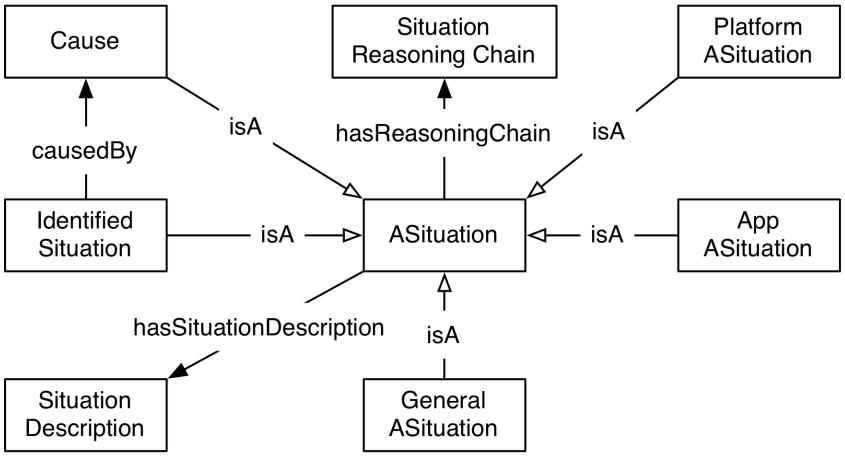

FIGURE 89 – Ontologie Kali-MOSA

A une situation sont liées une description et une cause. La description est soit un texte de description en langage naturel, soit un état d'éléments de contexte (Activité de l'utilisateur, Environnement, Présence de réseau, etc.), soit encore un ensemble de relations entre des situations. La description permet d'identifier une situation mais une situation peut avoir plusieurs façons d'être identifiée. Nous définissons une chaine de raisonnement pour chaque situation, quand la plateforme démarre, elle construit l'ensemble des chaines de raisonnement par une requête à Kali-MOSA. Puis elle déploie automatiquement ces chaines de raisonnement. Les situations d'adaptation peuvent aussi être identifiées par OWL-DL [1].

La cause associé à une situation est l'ensemble des contextes dans lequel cette situation est présente, c'est ainsi qu'une situation est identifiée. Une fois qu'une situation a été identifiée, elle devient une "IdentifiedASituation". Cette nouvelle instance de situation sera automatiquement classifiée par OWL-DL.

KaliMOSA est intégré sans le DOCK. Il peut accéder aux connaissances de contexte de KaliCOnto pour classifier les situations. Par exemple, la situation "*DeviceLost*" pour un hôte en interaction avec l'utilisateur se classifiée comme "*InteractionDeviceLost*", qui est un sous concept de la *DeviceLost*. Nous allons maintenant aborder les situations d'adaptation problématique.

Une situation d'adaptation peut être une simple situation problématique ou une situation composée c'est à dire contenant plusieurs situations problématiques de différents niveaux liées par une relation "cause". De plus une situation d'adaptation a une source (de type ContextEntity), une date d'identification et une localisation.

Une situation d'adaptation problématique [2] peut être produite par une chaine de raisonnement à partir de l'état des ressources et de l'utilisateur ou par une chaine de raisonnement à partir d'un raisonnement logique sur l'ontologie (nous présenterons les identifications des situations dans la section suivante). Les deux cas correspondent à un déséquilibre entre les besoins de l'utilisateur et les ressources disponibles dans l'environnement actuel. Nous distinguons les situations liées à l'utilisateur, celles liées au logiciel et celles liées au matériel.

1. OWL-DL : "OWL DL includes all OWL language constructs with restrictions. OWL DL is so named due to its correspondence with description logics." – http ://www.w3.org/TR/owl-guide/

2. Toutes les situations que la plateforme identifie sont des situations contextuelles. Les situations déclenchant une prise de décision sont les situations d'adaptation. Les situations problématiques sont des situations pouvant causer une adaptation qui sont identifiées par les chaînes de raisonnement de la plateforme.

Dans la catégorie des situations liées à l'utilisateur nous trouvons :

— Le changement d'environnement d'utilisateur

— La mobilité de l'utilisateur

Ces situations peuvent être détectées par des changements dans les ressources, par exemple, perte de connexion WIFI en raison de la mobilité de l'utilisateur ou entrée de l'utilisateur dans un nouvel environnement dans lequel des hôtes du domaine de cet utilisateur deviennent accessibles.

Les situations liées au matériel correspondent à des ajouts ou des retraits de ressources comme :

— Nouvel hôte "New Device"

— Nouveau périphérique "New Peripherique"

— Nouveau réseau "New Network"

— Disparition d'un Hôte "Device Lost"

— Déconnection d'un périphérique "Peripherique Lost"

— Disparition d'un réseau "Network Lost"

— Débit réseau faible "Low Connection"

— Alimentation stable "Energy Power Stable"

— Alimentation instable "Energy Power Instable"

— Alimentation faible "Low Energy Power"

La découverte de nouvelles ressources peut permettre une augmentation de la QdS, une reconfiguration en cas de dysfonctionnement ou un nouveau déploiement logiciel. La disparition de ressources peut provoquer des dysfonctionnements logiciels ou une baisse de QdS si les ressources disparues sont utilisées par certains logiciels en cours d'exécution.

Concernant le logiciel en cours d'exécution nous trouvons :

— La baisse de qualité logicielle

— Les exceptions d'exécution

La situation "Baisse de qualité logicielle" est identifiée par l'accumulation de données dans les connecteurs et est détectée par la plateforme Kalimucho qui supervise le cycle de vie des connecteurs (cf. section.12.4.2). Elle signifie qu'un composant logiciel fonctionne mal mais qu'il fonctionne encore. La situation "Exception d'exécution" est détectée par la plateforme Kalimucho qui supervise le cycle de vie des composants. Elle correspond à un dysfonctionnement de service ou de composant.

L'état d'énergie de l'hôte correspond à deux situations : "Energy Power stable" et "Energie Power instable". L'état "Energie Power stable" signifie que l'hôte dispose d'une source d'énergie stable comme, par exemple le secteur ou lorsque la batterie est chargée. L'état "Energiy Power instable" signifie que la source d'énergie de l'hôte change trop fréquemment. L'état "Low Energy Power" signigie l'hôte est sur batterie alors que le niveau de la batterie est faible.

La QdS du matériel peut augmenter ou diminuer, ces changements dynamiques peuvent provoquer des situations d'adaptation si les exigences des logiciels exécutés actuellement ne peuvent plus être respectées.

Parmi toutes ces situations problématiques certaines peuvent correspondre à une nécessité de reconfiguration tandis que d'autres peuvent simplement permettre une reconfiguration. Ainsi la perte d'une ressource indispensable à un service doit impérativement remettre en question l'architecture de l'applica-

tion tandis que si l'arrivée d'une nouvelle ressource peut permettre une amélioration de la QoS mais elle peut également ne provoquer aucune modification de l'application en cours. Il est important pour nous de savoir distinguer ces deux cas.

Il y a deux catégories de situations provoquant à coup sûr des adaptations : le premier est le déploiement ou l'arrêt de services à la demande de l'utilisateur ou de l'application. Le deuxième est l'état de dysfonctionnement logiciel provoqué soit par le logiciel lui-même soit par un changement d'environnement de l'utilisateur.

Déploiement ou arrêt de services à la demande de l'utilisateur ou de l'application :

— Déployer du logiciel (une application, un service, ou un composant) "Deploy Software"

— Arrêter du logiciel (une application, un service, ou un composant) "Stop Software"

Lancer une application est toujours du fait de l'utilisateur. Chaque lancement d'une application réussi va provoquer un ajout d'activité. Déployer un service est provoqué par une demande de l'application liée à sa logique métier incluant les vœux de l'utilisateur.

L'arrêt d'une application ou d'un service correspond à l'arrêt d'un ou plusieurs services en cours d'exécution ce qui aura pour conséquence de libérer des ressources. Un raisonnement sur les ressources disponibles va être mis en place. Si le résultat est positif, la situation "Raise QoS" sera identifiée, il s'agit là d'une situation d'adaptation possible mais pas impérative. Il faudra avoir recours à des raisonnements pour déterminer si l'adaptation doit être faite.

Les situations problématiques en rapport avec l'état logiciel correspondent aux trois cas suivants :

— dysfonctionnement logiciel "Software Dysfunction"

— Nécessité de baisser la QdS "Reduce QoS"

— Possibilité d'augmenter la QdS "Raise QoS"

Un dysfonctionnement logiciel est identifié par l'insatisfaction des exigences statiques du logiciel, par exemple, un composant ne fonctionne plus, un hôte où un logiciel tombe en panne ou disparait, il n'y a plus d'accès à Internet alors que cet accès est une condition nécessaire.

La nécessité de baisse de QdS est identifiée par l'insatisfaction des exigences de QdS dynamique. Par exemple, l'accès à Internet est une condition nécessaire à un haut niveau de QdS.

La possibilité d'augmenter la QdS est identifiée par le fait que les ressources actuelles peuvent supporter une augmentation de QdS. Par exemple, dans le cas précèdent, l'accès à Internet est à nouveau disponible et le service peut augmenter sa QdS.

Dans la section suivante, nous présentons les situations d'adaptation générales et leur identification.

12.4 COMMENT POUVONS-NOUS DÉTECTER LES SITUATIONS D'ADAPTATION PROBLÉMATIQUES ?

De ce qui précède nous pouvons extraire les différentes origines de reconfiguration telles que présentées dans la figure 90.

Il s'agit donc maintenant d'identifier comment la plateforme peut détecter les quatre origines de reconfiguration c'est à dire :

— Demande d'ajout ou de suppression de service

— Problème ou erreur d'exécution

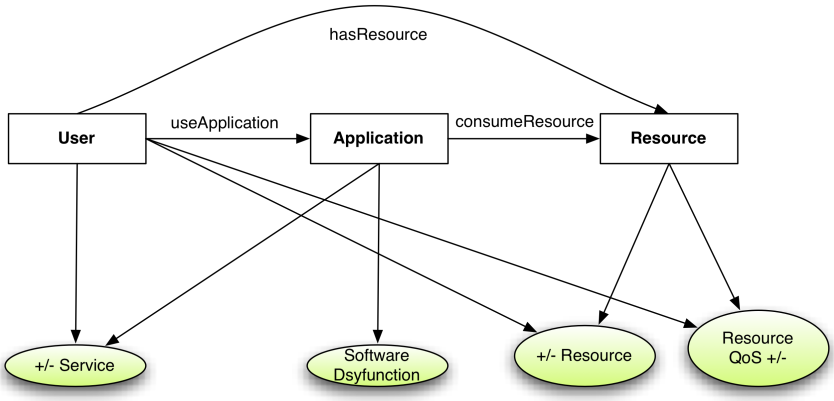

FIGURE 90 – Origines de reconfigurations possibles

— Présence de ressource

— Qualité de ressource

Chaque origine correspond à une information de contexte de haut niveau. Elles sont définies dans l'ontologie de contexte KaliCOnto à la page.62.

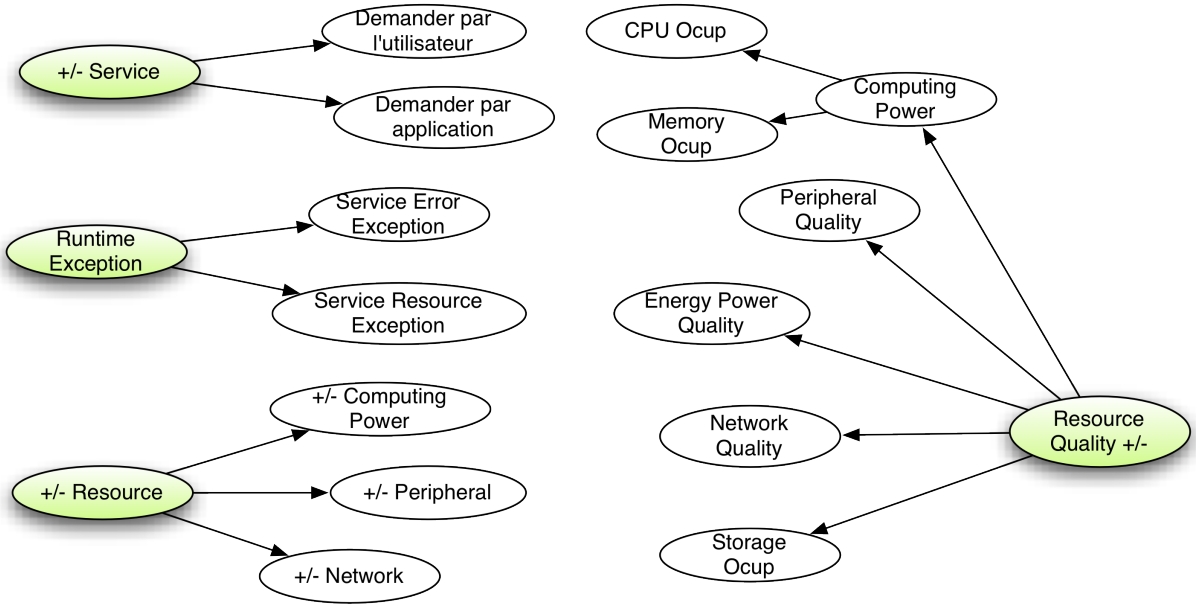

FIGURE 91 – Origine des adaptations

12.4.1 *Demande d'ajout/suppression de service*

Nous avons vu que l'ajout ou la suppression de service peut être provoquée soit par l'utilisateur soit par l'application. Il convient maintenant d'envisager deux cas :

— L'utilisateur ou l'application fait une demande explicite d'ajout ou de suppression de service en cours d'utilisation d'une application.

— L'ajout d'un ou de plusieurs services est causé par un premier déploiement d'application fait soit à la demande de l'utilisateur par une action sur une interface soit par détection de sa présence en un lieu donné. Ce dernier cas correspond, par exemple, au déploiement automatique d'une application spécifique lorsque l'utilisateur rentre dans un musée.

L'utilisateur fait sa demande au travers de l'interface offerte par l'application installée. Par exemple un bouton sur son interface lui permet de mettre en place un nouveau service ou d'en arrêter un. L'application, quant à elle, peut demander la mise en place d'un nouveau service, par exemple, lorsque les données qu'elle manipule sont sensibles elle ajoute un service de cryptage.

La détection de ces événements se fera par la mise en place d'un service dédié appelé ContextCollectionService (service fourni par un capteur logique de type KaliSensor ainsi qu'un SimpleNotifyContextCollector. Il reçoit en entrée un événement (EventAppAddService/EventAppRemoveService) envoyé par l'application et produit un événement en sortie.

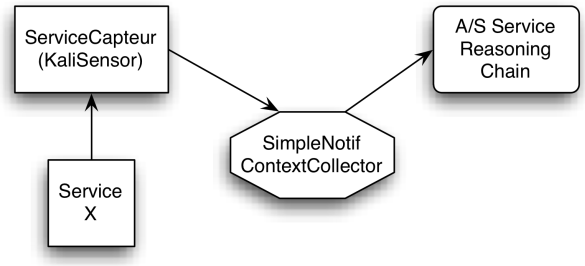

FIGURE 92 – Schéma de détection de demande d'ajout/suppression de service

Le problème est de détecter soit une action spécifique de l'utilisateur sur l'interface soit sa présence en un lieu.

Le déclenchement d'un premier déploiement par action sur l'interface est réalisé de la façon suivante : un composant métier de Kalimucho est installé et propose une interface. Lorsque l'utilisateur agit sur cette interface, le composant métier envoie un événement à un ContextCollectionService (capteur logique KaliSensor associé à un SimpleNotifyContextCollector).

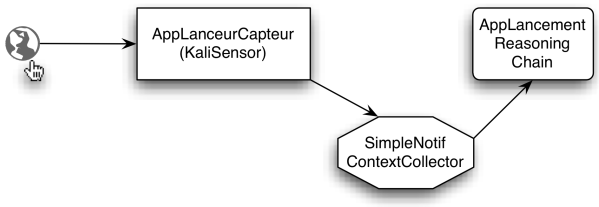

FIGURE 93 – Déploiement par une action d'interface

Le déclenchement d'un premier déploiement par présence de l'utilisateur en un lieu donné utilise le fait que la plateforme Kalimucho envoie en broadcast un message signalant sa présence dès qu'elle détecte un nouveau réseau accessible. Ce message est reçu par les plateformes placées sur les machines de ce réseau. Si sur l'une de ces plateformes a installé un ContextCollectionService dont l'entrée est ce type de message, il produit en sortie un événement de présence. Sur cette machine cet événement suit le

cheminement normal (ContextCollectionService, chaîne de raisonnement etc.) qui provoquera la détection d'une situation susceptible de provoquer le déploiement d'un ou de plusieurs services.

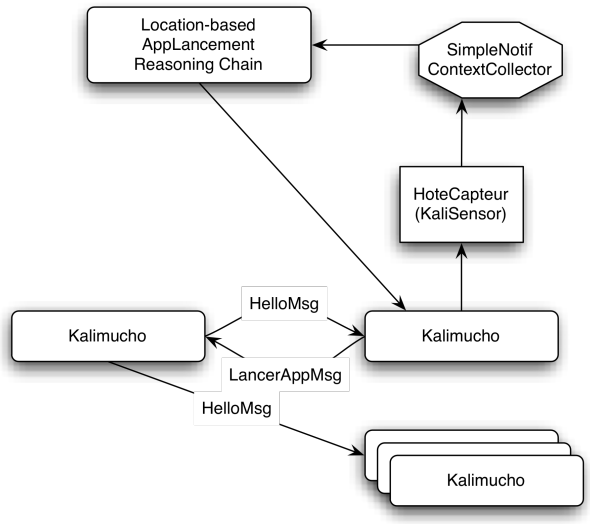

FIGURE 94 – Lancement d'application basé sur une localisation spécifique

12.4.2 *Problème ou erreur d'exécution*

Nous envisagerons ici trois types de problèmes d'exécution :

— un composant rencontre une exception lors de son exécution

— un composant fonctionne mais ne remplit pas convenablement son rôle car il ne parvient pas à traiter les données qui lui sont envoyées

— un connecteur ne remplit pas convenablement son rôle car il ne parvient pas à transmettre les données qu'il reçoit

Si le premier cas suppose une reconfiguration immédiate sous peine que le service applicatif ne soit plus assuré, les deux autres signalent simplement une baisse de qualité de service qui peut donner lieu à une reconfiguration ou n'être que transitoire.

Lorsqu'un connecteur ne remplit pas son rôle il convient de différencier le cas où il s'agit d'un problème de réseau (qui sera traité plus loin) de celui où il s'agit d'un problème propre au composant métier inclus dans le connecteur (cf. page.1.1.1.2).

Pour discriminer les différentes situations nous allons utiliser le tableau suivant :

12.4.2.1 *Erreur d'exécution*

Une erreur d'exécution d'un service correspond au fait que le service est complètement arrêté en raison d'une exception qui n'a pas été prise en compte par le service lui-même.

Nous rappelons ici le cycle de vie (figure.95) du modèle Osagaia (voir page.40). Il contient six états : init, run, pause, migration, destroy, et exception.

Type de connecteur	Accumulation dans le buffer d'entrée	Accumulation dans le buffer de sortie
Interne	Inadéquation du CM du connecteur	Inadéquation des deux composants liés par ce connecteur
Sortie sur réseau	Inadéquation du CM du connecteur	Problème de transmission par le réseau
Entrée sur réseau	Inadéquation du CM du connecteur	Inadéquation des deux composants liés par ce connecteur

TABLE 11 – Détection erreur d'exécution

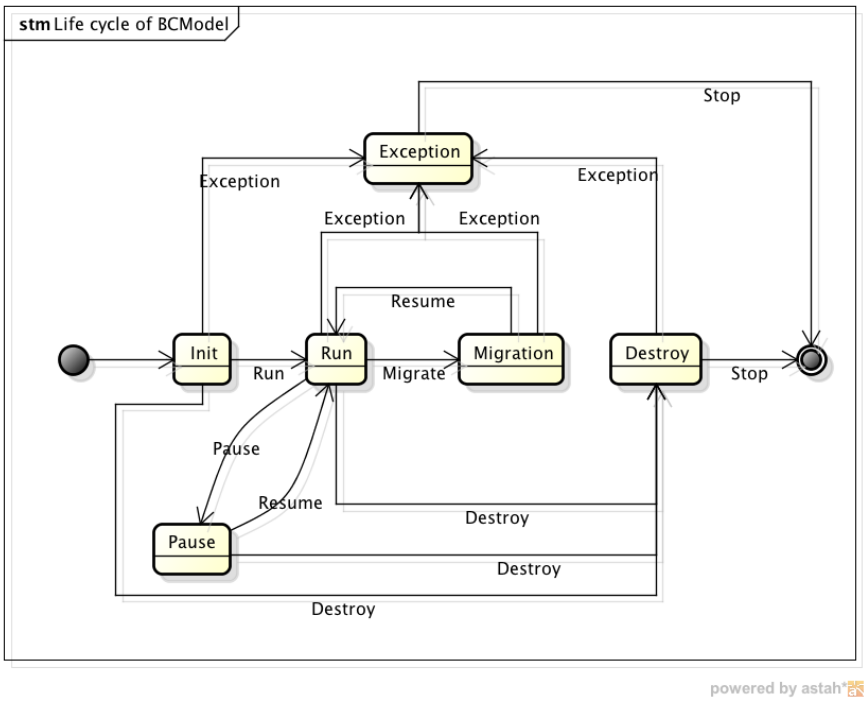

FIGURE 95 – Le cycle de vie d'Osagaia

Les exceptions sont récupérées par la plateforme Kalimucho qui transmet ces évènements à des KaliMonitors qui sont des ContextCollectionServices. Un KaliMonitor est associé à chaque service et produit un événement qui est enregistré dans l'ontologie de contexte KaliCOnto pour être traité.

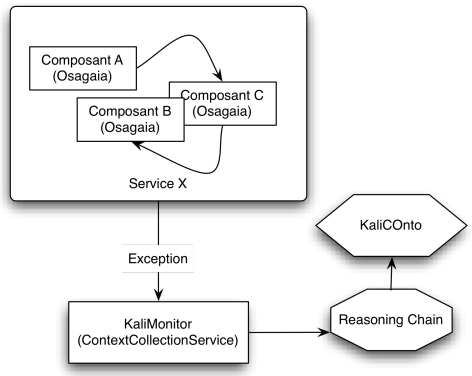

FIGURE 96 – Erreur d'exécution

12.4.2.2 *Problème d'exécution*

Lorsqu'un composant ne parvient pas à traiter les données qu'il reçoit via ses connecteurs d'entrée, celles-ci s'accumulent dans les buffers de ces connecteurs. On a vu que ceci peut correspondre à une inadéquation des deux composants liés par ce connecteur. C'est à dire que soit l'un produit trop de données soit l'autre ne parvient pas à traiter les données produites. Ceci peut également correspondre à une inadéquation du composant métier encapsulé dans le connecteur (le cas de problème réseau sera traité par ailleurs). Les ajouts ou suppression de données dans l'un des buffers d'un connecteur sont connus de l'UC de ce connecteur qui les transmet à un KaliSensor. Nous utiliserons donc un KaliSensor associé à un ConditionalNotifyContextCollector en mode de détection de seuil pour chacun des connecteurs présents.

On a également vu que la différence entre un problème de qualité de service et un problème de réseau se fait en regardant si le connecteur émet sur le réseau et que l'accumulation de données s'est produite dans son buffer de sortie. Le ConditionalNotifyContextCollector sera programmé de façon à ne produire d'événement de sortie que lorsque l'accumulation montre un problème de qualité de service. L'événement produit contient le nom du connecteur et l'indication du buffer ayant franchi le seuil (buffer d'entrée ou de sortie).

12.4.3 *Présence de ressource*

Les seules ressources gérées sont celles auxquelles l'utilisateur peut accéder directement ou via le réseau (domaine de l'utilisateur). Sur chaque machine où est installée la plateforme Kalimucho (machines que nous appellerons KaliHôtes) elle assure la surveillance des ressources. Notre travail consiste donc seulement à capturer les informations issues de la plateforme et à les injecter dans notre ontologie de contexte. Nous associons donc à chaque ressource un capteur qui reçoit les informations produites par la plateforme Kalimucho. Nous distinguerons les deux types de présence de ressources suivants : matérielle et réseau La présence des ressources matérielles correspond à celle des KaliHôtes et des périphériques.

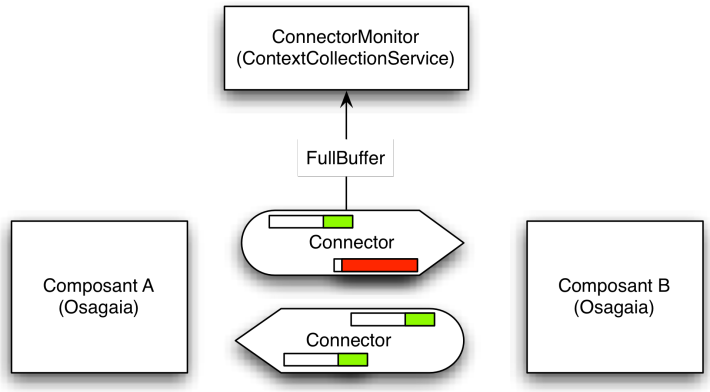

FIGURE 97 – Schéma de la détection d'un problème d'exécution

Les périphériques sont des dispositifs supplémentaires connectés aux KaliHôtes (capteur sans fils, GPS externe, souris Bluetooth, etc.) Ces périphériques correspondent aux ressources matérielles d'un KaliHôte et relèvent de l'un des trois types suivants :

— réseau (interface wifi, Bluetooth, etc.)
— stockage (disque dur, carte SD, etc.)
— interaction (écran tactile, capteurs d'environnement, etc.)

La présence d'une ressource réseau correspond à la présence des réseaux locaux et des accès internet auxquels peut se connecter l'utilisateur via Le KaliHôte (PC, Smartphone, Tablet, etc.). Par exemple, un smartphone est connecté à l'internet en 3G ou connecté au réseau local en WIFI et peut aussi être connecté à plusieurs téléphones en Bluetooth. Ce Smartphone possède donc plusieurs ressources réseau.

La détection de ces disponibilités est faite par des ContextCollectionServices selon les méthodes suivantes :

12.4.3.1 *Détection d'un hôte*

Au démarrage ou lorsqu'elle détecte un nouveau réseau, chaque plateforme envoie en broadcast, sur tous les réseaux disponibles, un message de type "Hello". Ce message est reçu par les plateformes situées sur les autres hôtes et provoque l'ajout de ce nouvel hôte dans leur DNS local. En effet, les plateformes gèrent un DNS local dans lequel elles inscrivent toutes les plateformes qu'elles connaissent soit par le biais d'un message de "Hello" reçu soit parce qu'elles ont reçu un message de leur part. De plus, ces DNS sont diffusés en broadcast par les plateformes à intervalles réguliers. Ceci permet à chaque plateforme réceptrice de compléter les informations contenues dans son propre DNS. Chaque ajout d'un hôte dans le DNS provoque l'envoi d'un événement au KaliSensor chargé de détecter l'apparition ou la disparition d'un hôte [42].

La détection de la disparition d'un hôte est moins aisée à réaliser et peut ne pas être possible. Lors de l'arrêt d'une plateforme sur un hôte elle envoie en broadcast un message "Good bye" sur tous les réseaux disponibles. De même, afin d'éviter la fermeture de connexion par les routeurs, les plateformes connectées en GSM via un proxy maintiennent la connexion ouverte par l'envoi de messages à intervalles réguliers. Ce proxy peut donc détecter leur disparition s'il ne reçoit plus ces messages au bout d'un délai. En revanche la disparition d'une plateforme après une perte de connectivité ne peut pas être connue des autres. Toutefois

la durée de vie des hôtes inscrits dans les DNS est limitée et lorsqu'elle se termine l'hôte est considéré comme ayant disparu. Dans tous les cas où une disparition est détectée, les plateformes mettent à jour leur DNS en en retirant l'hôte qui a disparu et diffusent instantanément ce DNS sans attendre le délai normal. Chaque retrait d'un hôte dans le DNS provoque l'envoi d'un événement au KaliSensor chargé de détecter l'apparition ou la disparition d'un hôte.

12.4.3.2 *Détection des périphériques associés à un hôte*

Lorsque le système d'exploitation de l'hôte le permet (UPnP, broadcast Listener sur Android), un KaliSensor reçoit un événement de détection de l'apparition ou de la disparition d'un périphérique connecté à cet hôte.

12.4.3.3 *Détection de la présence réseau*

Lorsque le système d'exploitation de l'hôte le permet (broadcast Listener sur Android), un KaliSensor reçoit un événement de détection de l'apparition ou de la disparition d'une connexion réseau.

Toutefois si certaines ressources peuvent se caractériser par une simple présence ou absence, il faut également tenir compte du fait que certaines se caractérisent par une qualité de service. Ainsi, si la présence d'un CPU sur un hôte ne fait aucun doute, sa charge peut conduire à devoir envisager des reconfigurations. Il convient donc d'étudier les ressources selon cet aspect de qualité.

12.4.4 *Qualité de ressource*

Nous distinguerons, les cinq types de ressources suivants en ce qui concerne la mesure de la qualité :

1. Puissance
   a) CPU
   b) Mémoire
2. Etat des périphériques
3. Stockage
4. Réseau
5. Energie

Il existe donc cinq types de mesures de ressources. La première mesure l'énergie matérielle, le seconde sa puissance, la troisième informe sur l'état des périphériques, la quatrième donne des informations sur l'espace de stockage et la cinquième sur le réseau.

Energie matérielle : la mesure d'énergie d'un appareil est effectuée par l'hôte lui-même. Il mesure son propre état d'énergie ainsi que celui des périphériques connectés (un capteur sans fils a peu de puissance de calcul, il est considéré comme un périphérique, mais il peut mesurer son état d'énergie et l'envoyer à l'hôte auquel il est connecté). La mesure d'énergie est effectuée par le *Context Collection Service* appelé *KaliNergie* constitué d'un capteur logique de type *KaliSensor* (cf. page.76) et de plusieurs *ContextCollectors* (cf. page.81) associés aux types de mesures possibles. Le *KaliNergie* est chargé de mesurer : le type d'énergie (secteur, batterie), l'état de l'énergie (en charge, chargé, en consommation) et la capacité restante en

pourcentage. Dans le cas où l'hôte est un appareil mobile, le *KaliNergie* utilise un *SimpleNotifyContextCollector* pour chaque modification de la source d'énergie (branchement/débranchement d'un chargeur). Ce *ContextCollector* produit des informations sur l'état et le type d'énergie. Le *KaliNergie* utilise également un *ConditionalNotifyContextCollector* pour signaler les variations de charge de la batterie et produit des notifications contenant le pourcentage de batterie restant.

Puissance de calcul : Pour mesurer la puissance de calcul il faut connaître la puissance du CPU, le cas échéant, la quantité de RAM initiale et disponible en cours d'exécution. La puissance du CPU et la taille de la mémoire sont mesurées lors de l'ajout d'un hôte comme nous l'avons expliqué à la section précédente. Nous définissons le KaliCPUMonitor et le KaliMemoMonitor qui sont tous les deux de type ContextCollectionService afin de fournir les mesures d'occupation de ces ressources matérielles. L'occupation du CPU et de la mémoire sont mesurées à intervalles réguliers et une moyenne est réalisée car la valeur instantanée n'a pas de sens. L'occupation du CPU est exprimée en pourcentage. Celle de la RAM est exprimée en pourcentage de RAM disponible et accompagnée de la quantité de mémoire disponible.

Etat des périphériques : Les périphériques sont des appareils sans puissance de calcul ou ayant trop peu de puissance et étant obligés de se connecter à l'hôte de l'utilisateur (par exemple, une souris Bluetooth ou une caméra USB). Un KaliPeriphMonitor (de type ContextCollectionService) surveille les états de connexion (connexion bonne, normale, faible, et mauvaise - selon le niveau de signal mesuré dans le cas de connexion sans fil), le niveau de batterie (s'il en a une) et les états de fonctionnement (en cours d'exécution, en veille, etc.) de ce périphérique.

Espace de stockage : L'espace de stockage qui nous intéresse est celui disponible sur le périphérique et non, par exemple, celui éventuellement disponible en réseau ou sur le Cloud. Nous utiliserons, pour mesurer l'espace de stockage disponible, un KaliLocalStorage (de type ContextCollectionService). Il s'agit d'un capteur de type KaliSensor associé à un collecteur en mode requête (QueryContextCollector).

Etat du réseau : Le rôle de la plateforme consiste à reconfigurer l'application en cas de problème de qualité sur le réseau (la présence du réseau a été traitée précédemment). Il ne s'agit pas ici de contrôler le fonctionnement du réseau au-delà de ce dont a besoin l'application. Par conséquent seul le trafic dans les connecteurs distribués est significatif pour la plateforme.

Un connecteur recevant des données par le réseau ne peut fournir aucune information pertinente sur la qualité du réseau dans la mesure où il connaît la quantité d'informations qu'il reçoit mais pas celle qu'il devrait recevoir. En revanche un connecteur envoyant des données sur le réseau peut détecter le fait qu'il ne parvient pas à transmettre ses données à un rythme suffisant. Ce type de problème est détectable par l'observation du contenu du buffer de sortie du connecteur (cf. tableau.11). Nous utiliserons donc, comme nous l'avons fait pour le cas des problèmes d'exécution un KaliSensor associé à un ConditionalNotifyContextCollector en mode de détection de seuil pour chacun des connecteurs présents.

Après ce tour d'horizon des moyens de détection des diverses origines de situations susceptibles de provoquer des reconfigurations, nous allons nous intéresser au modèle de contexte utilisé pour stocker les informations produites par les divers ContextCollectionServices. Ce modèle doit permettre ensuite aux chaînes de raisonnement de produire des informations de haut niveau sémantique qui pourront être interprétées en tant que situations de reconfiguration.

Dans cette partie nous avons détaillé les moments auxquels peuvent être faites des adaptations. Ils sont liés à la demande de l'utilisateur, à l'état de l'application, à la présence et à la qualité des ressources. Dans la prochaine section nous présenterons notre modèle de contexte pour la plateforme d'adaptation mais avant nous allons étudier un cas d'utilisation simple.

12.4.5 *Cas d'utilisation d'une détection de situation d'adaptation*

Nous utilisons un exemple pour expliquer comment fonctionne la plateforme depuis la collection du contexte de bas niveau jusqu'à la notification de la situation d'adaptation au service d'adaptation (Kali-Adapt).

Nous considérons tout d'abord une situation très simple de type "Low Energy Power" sur un hôte (Device A). Puis nous envisagerons un second scénario pour montrer comment la plateforme réagit face à des situations plus complexes.

Après l'événement "Low Energy Power" sur le "Device A", celui-ci s'éteindra faute de batterie. Avant que cela ne se produise, la plateforme tentera de migrer le composant (C1) qui appartient au service (S1) de l'application (App1) (cf. figure.98).

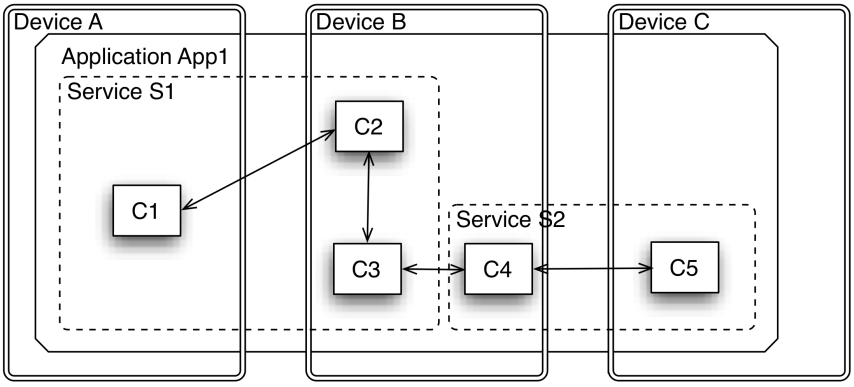

FIGURE 98 – La déploiement de l'application (App1)

Dans la figure 99, nous présentons le premier scénario qui utilise une chaîne de raisonnement sur l'état d'alimentation (cf. figure.85 dans le chapitre KaliReason.) Cette chaîne s'exécute sur le "Device A" et met à jour le DECK local. Par ailleurs, sur l'hôte qui héberge le DOCK (Device B), il y a des chaînes de raisonnement pour chaque situation contextuelle identifiée et en particulier celle de la perte d'energie d'un hôte du domaine. La chaîne de raisonnement sur l'énergie ("Energy Power Reasoning Chain") est abonnée aux notifications de changement de l'état d'alimentation (via les DECKs) de tous les hôtes du domaine d'adaptation. Quand cette chaîne reçoit une notification, elle va l'analyser et détecter qu'elle a pour source le "Device A" et pour cause "Low Energy Power ". Ceci aboutira à l'ajout d'une situation problématique dans le DOCK. Le service d'adaptation (Kali-Adapt) en sera notifié et proposera une reconfiguration qui pourrait être, dans ce cas, de migrer le composant C1 vers un autre périphérique du domaine.

Le deuxième scénario (cf. figure.100) correspond au moment où, faute d'énergie, le Device A s'éteint. Nous supposons de plus que la résolution du problème correspondant au scenario 1 n'a pas abouti par exemple parce que la migration de C1 n'a pas pu se faire.

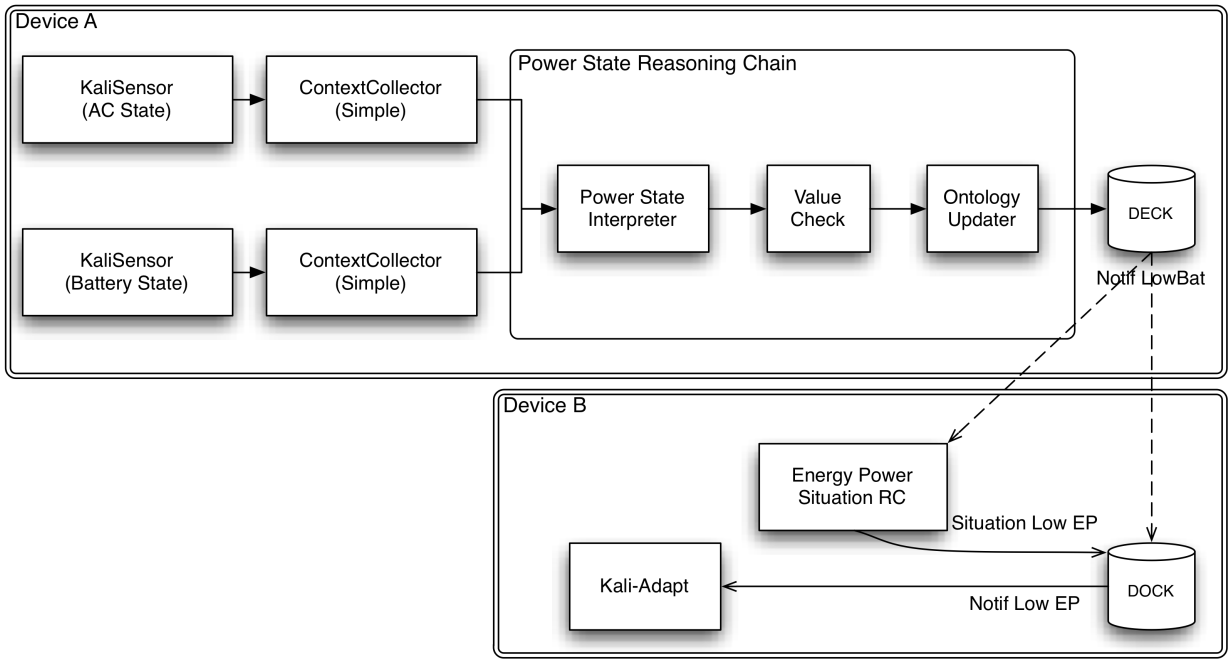

FIGURE 99 – Composition des services du scénario 1

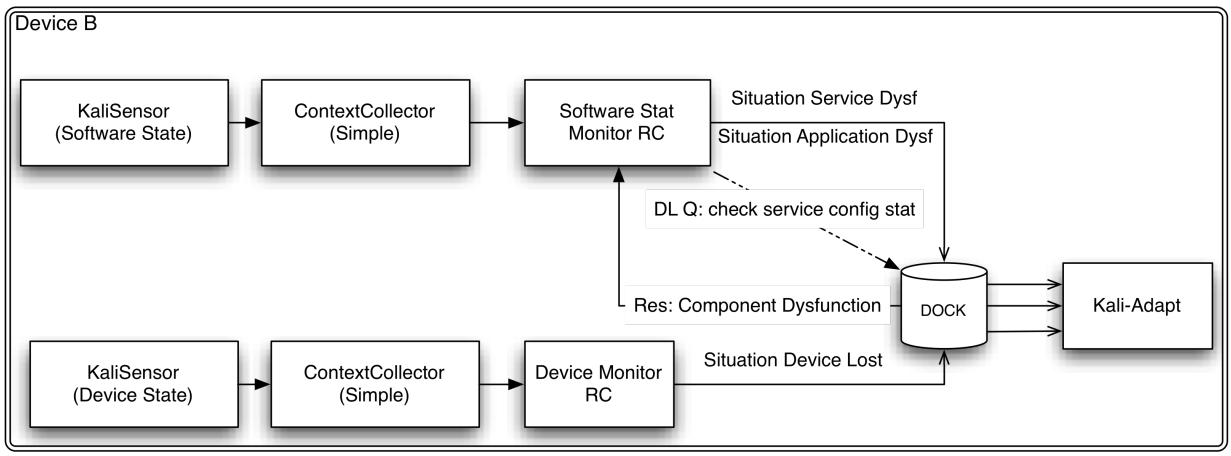

FIGURE 100 – Composition des services du scénario 2

La chaîne de raisonnement "Device Monitor" qui observe l'ajout et la disparition des hôtes du domaine va détecter la disparition du "Device A" et ajouter cette situation au DOCK. En même temps, la chaîne de raisonnement "Application Monitor" qui observe les demandes de déploiement, d'arrêt et de dysfonctionnement de logiciel détecte un dysfonctionnement du service (S1) (cf. section.12.4.2 : Problème ou erreur d'exécution). Pour en trouver la cause, elle consulte le DOCK, par une requête DL Query, pour vérifier l'état de la configuration du service S1. Un raisonnement OWL-DL sur le DOCK permet d'identifier une situation de perte du composant C1 du service S1 dont la cause est la disparition du Device A. La chaîne "Application Monitor" va créer une situation "Service Dysfunction" ayant pour source S1 et pour cause "Component Lost" de C1 et une situation "Application Dysfunction" avec comme source App1 et comme cause "Service Dysfunction" de S1. La figure 101 présente le graphe correspondant à ces situations. faca à une telle sitution le service de reconfiguration pourra par exemple proposer de remplacer le service S1 par un autre ou de déployer un composant de remplacement de C1.

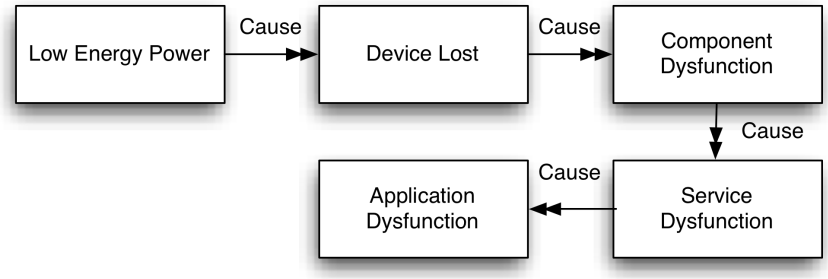

FIGURE 101 – Graphe des situations

12.5 LES SITUATIONS D'ADAPTATION POUR LA PLATEFORME

Les situations d'adaptation pour la plateforme sont liées à la structure et la logique de fonctionnement de la plateforme. Notre plateforme contient trois grandes parties (cf. figure.87) : le middleware de contexte (Kali2Much), le conteneur de modèle de contexte (KaliCOnto) qui gère le DOCK et le DECK (cf. page.69) et le service de prise de décision (Kali-Adapt) que nous présenterons dans le chapitre suivant. Pendant le runtime, la plateforme peut se trouver dans des situations nécessitant des adaptions. Les services de la plateforme sont implémentés par le même modèle de composants que les applications. Ainsi toutes les identifications de situations générales peuvent également être appliquées aux services de la plateforme. Mais les situations d'adaptation générales liées aux services de la plateforme ont un plus haut niveau de priorité que celles des applications. Kali-Adapt va donc les traiter en premier. En cours d'exécution la plateforme peut rencontrer des situations différentes de celles d'adaptation générales. Elles sont créées par l'interaction entre l'utilisateur et la plateforme ou par des dysfonctionnements de la plateforme. Par exemple quand l'utilisateur perd la connexion réseau sur son mobile. Dans cette situation la plateforme peut perdre le DOCK, ou le service Kali-Adapt qu'elle devra donc reconstruire par des opérations d'adaptation. Dans cette section nous présentons les six situations d'adaptation de la plateforme que nous avons identifiées : Utilisateur perdu, Utilisateur isolé, service d'adaptation perdu, duplication du service d'adaptation et dysfonctionnement de service plateforme (cf. table.9).

Nous allons envisager un cas d'utilisation pour expliquer les situations d'adaptation de la plateforme liées à la mobilité de l'utilisateur. Dans un premier temps l'utilisateur utilise son PC portable chez lui (cf. figure.102). Puis il sort de chez lui avec son téléphone et sa tablette mais il perd la connexion avec sa maison (cf. figure.103). Cet exemple montre que l'on identifiera deux situations différentes pour le même événement (l'utilisateur sort de chez lui). Ces deux situations correspondent à deux points de vue de la plateforme. En effet, quand l'utilisateur quitte sa maison et perd la connexion, la partie de la plateforme située chez lui détecte une perte de l'utilisateur. Tandis que la partie de la plateforme sur ses mobiles détecte une perte du service d'adaptation (DOCK et Kali-Adapt).

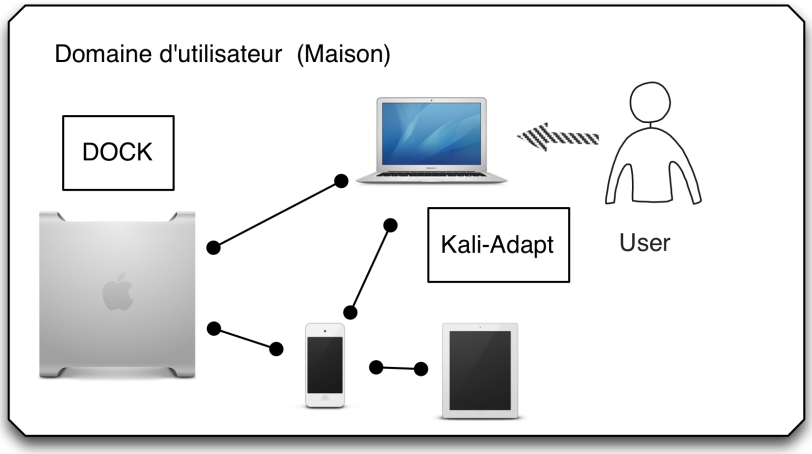

FIGURE 102 – L'utilisateur interagit avec la plateforme chez lui

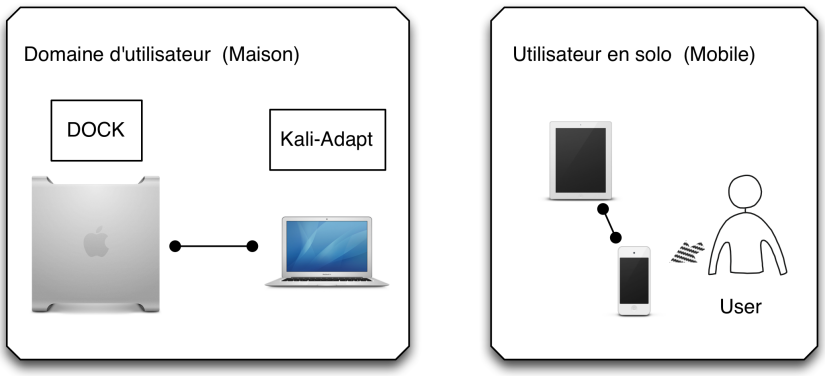

FIGURE 103 – L'utilisateur quitte sa maison, ses hôtes mobiles ont perdu la connexion

Mais les situations d'adaptation de la plateforme ne sont pas seulement causées par la mobilité de l'utilisateur. Elles peuvent provenir de la perte d'énergie, des défauts de fonctionnement d'un hôte, etc. Nous allons maintenant examiner les situations d'adaptation de la plateforme en détail.

12.5.0.1 *Utilisateur perdu*

Il s'agit d'une situation d'adaptation définie par la perte de toute connexion au réseau par un utilisateur alors qu'une application est en cours.

Ce cas correspond à celui détecté par la plateforme fonctionnant chez l'utilisateur dans la figure 103 lorsque ce dernier quitte son domicile.

12.5.0.2 *Utilisateur isolé*

Utilisateur isolé est une situation d'adaptation définie par : un utilisateur a accès à un KaliHôte mais il a perdu la communication avec son domaine et donc avec la plateforme qui contient les services principaux d'adaptation. Cette situation correspond à au moins l'une des deux situations identifiées comme la perte de service d'adaptation (DOCK et Kali-Adapt). Elle est identifiée par la plateforme résidant sur l'hôte de l'utilisateur.

12.5.0.3 *Service d'adaptation perdu*

Le service d'adaptation est la partie plus importante de la plateforme d'adaptation puisqu'il contient toutes les connaissances du domaine. La plateforme peut perdre ce service pendant son exécution soit en raison d'un dysfonctionnement de l'hôte où s'exécute ce service soit en raison de la perte de connexion avec cet hôte.

Le dysfonctionnement de ce service ou la perte de connexion sont des situations d'adaptation générales.

12.5.0.4 *Service d'adaptation dupliqué*

La duplication du service d'adaptation peut se produire lorsque, après une disparation de ce service lors d'une perte de connexion, l'utilisateur retrouve l'ancien service après une reconnexion. Dans ce cas il y a deux services d'adaptation dans un même domaine d'utilisateur.

La présence de plusieurs services d'adaptation en exécution dans le même domaine d'utilisateur, est une situation qui doit être identifiée. Dans la section 13.2.5, nous présenterons les détails de la solution pour cette situation.

La situation "PF Service Dysfunction" va être traité comme une situation de "Software Dysfunction" mais avec une priorité élevé.

12.6 CONCLUSION

Dans ce chapitre nous nous sommes tout d'abord intéressés au moment auquel la plateforme est susceptible de faire des adaptations pour les applications ou pour elle-même. Par la suite nous avons expliqué pourquoi ces situations sont importantes et comment les identifier. Puis nous avons présenté notre modèle de situations d'adaptation Kali-MOSA organisé en ontologie. Pour finir nous avons présenté les différents types de situation d'adaptation : situation d'adaptation générale, situation d'adaptation de la plateforme et situation d'adaptation de l'application.

Dans le chapitre suivant, nous présenterons le mécanisme de la prise de décision et le modèle de solution qui se base sur les situations d'identifiées dans ce chapitre pour trouver des solutions d'adaptation. Puis nous montrerons comment sont exécutées les opérations d'adaptation.

13

SERVICE D'ADAPTATION (KALI-ADAPT)

Dans ce chapitre nous présentons la partie concernant la prise de décisions d'adaptation. Selon les différents types de plateformes d'adaptation, la définition de cette prise de décision change. Par exemple, pour les plateformes basées sur des règles ECA (Évènement, Condition, Action) [75] , une prise de décision signifie que quand la plateforme capte des évènements de changement de contexte, si ces évènements remplissent les conditions énoncées dans les règles prédéfinies, une adaptation va être effectuée par les actions définies dans ces mêmes règles. Pour les plateformes à heuristique comme MUSIC [51] , CAMPUS [143] , etc. une prise de décision est équivalente à une paramétrisation de composant logiciel en fonction des ressources disponibles. Notre plateforme est basée sur des situations sémantiques, ceci nous permet de traiter le contexte avec une vision de plus haut niveau. La prise de décision d'adaptation consiste à trouver une nouvelle architecture de l'application applicable dans le contexte actuel pour répondre à la situation d'adaptation identifiée.

Nous rappelons que le "Decision-Maker" travaille dans le domaine d'adaptation de l'utilisateur. Chaque domaine d'adaptation a un seul "Decision-Maker". Dans ce chapitre nous présentons d'abord les problématiques de la prise de décision d'adaptation, puis notre proposition.

13.1 PROBLÉMATIQUE

La question qui se pose est : "Comment adapter ?". Pour répondre à cette question il faut, tout d'abord savoir "Quand" avons-nous besoin d'une adaptation, et "Quoi" adapter pour offrir un service correct à l'utilisateur. Dans le chapitre précédent nous avons répondu à la question "Quand", dans ce chapitre nous allons répondre à la question "Quoi". Il faut donc savoir, dans une situation d'adaptation donnée, ce qu'il y a à adapter pour garantir un fonctionnement du service qui réponde aux besoins de l'utilisateur dans le contexte actuel. Nous pouvons modifier les assemblages de composants logiciels de l'application et ceux de la plateforme elle-même. Donc une adaptation peut se résumer à une suite de modifications d'architecture (ajout/suppression/migration de composants et connexions/déconnexion de composants) dans un environnement d'exécution donné. Enfin, une adaptation peut s'exprimer comme une solution proposée en réponse à une situation identifiée. Bien entendu, il convient de déterminer quelle partie du logiciel et quelle partie de l'environnement d'exécution doivent être modifiées. Pour cela il faut déterminer ce qui influence (ou qui est la cause de) la situation d'adaptation. Rappelons qu'une situation d'adaptation contextuelle provient d'un déséquilibre entre les besoins de l'utilisateur et les ressources actuelles de l'environnement d'exécution.

La seconde question est "qu'est-ce qu'une prise de décision d'adaptation ?". Pour notre plateforme, selon l'explication précédente, nous savons qu'une prise de décision d'adaptation consiste à trouver une solution qui fonctionne dans le contexte actuel en réponse à une situation d'adaptation donnée. Donc, pour identifier une solution, nous devons connaître les relations entre les solutions et les situations d'adaptation. Mais nous devons également tenir compte de la compatibilité de la solution trouvée avec le contexte actuel.

La troisième problématique est la mobilité. À cause de la mobilité de l'utilisateur, le "Decision-Maker" peut être séparé de l'utilisateur. Par exemple quand le "Decision-Maker" s'exécute sur un PC fixe chez lui tandis que l'utilisateur est parti avec son Smartphone et que la connexion entre ces machines est interrompue. Il s'agit d'une situation concernant la plateforme dont nous avons parlé dans le chapitre précédent. Dans ce chapitre nous présentons notre proposition de solution pour cette situation.

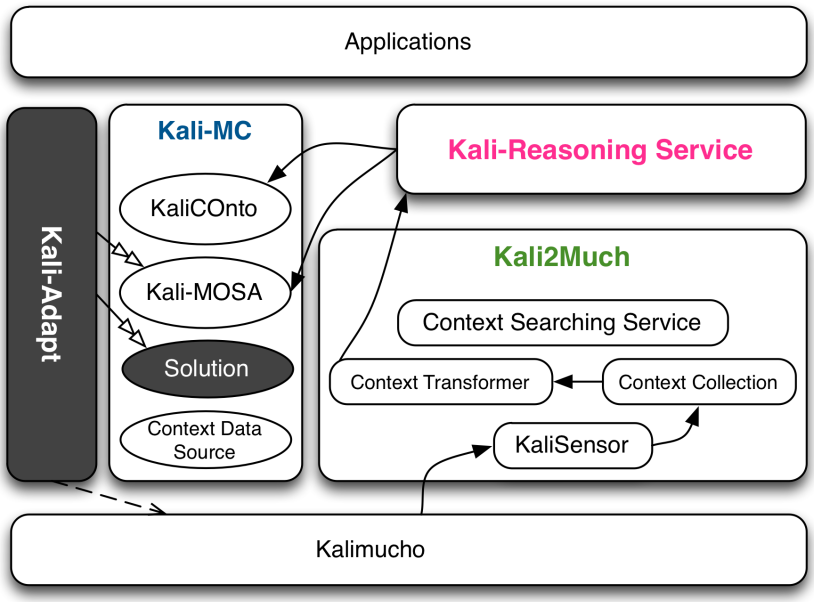

FIGURE 104 – Positionnement de "Kali-Adapt" dans la plateforme

Dans le figure 104 nous rappelons l'architecture de la plateforme Kalimucho-A, et le positionnement du Kali-Adapt dans la plateforme. Nous avons choisi d'utiliser une base de connaissances pour la prise de décisions d'adaptation. Il s'agit de solutions associées à la base de connaissances du contexte. Cette dernière permet l'identification de la solution et la vérification de sa compatibilité avec le contexte courant. La section suivante présente notre proposition en détail.

13.2 PROPOSITION

Nous proposons une architecture utilisant les situations sémantiques produites par les chaînes de raisonnement pour réaliser une prise de décision d'adaptation en fonction du contexte courant (cf. figure.105). Dans cette section nous présentons les détails de cette architecture et les rôles de chaque composant.

Lorsqu'une situation d'adaptation est identifiée, une notification de type "nouvelle situation d'adaptation identifiée" est envoyée au "Decision-Maker" (cf. Annex.A.5.2). Le "Decision-Maker" est notifié de l'ajout, dans la base de connaissances, d'une nouvelle "ASituation" d'alarme. (ASituation est le concept as-

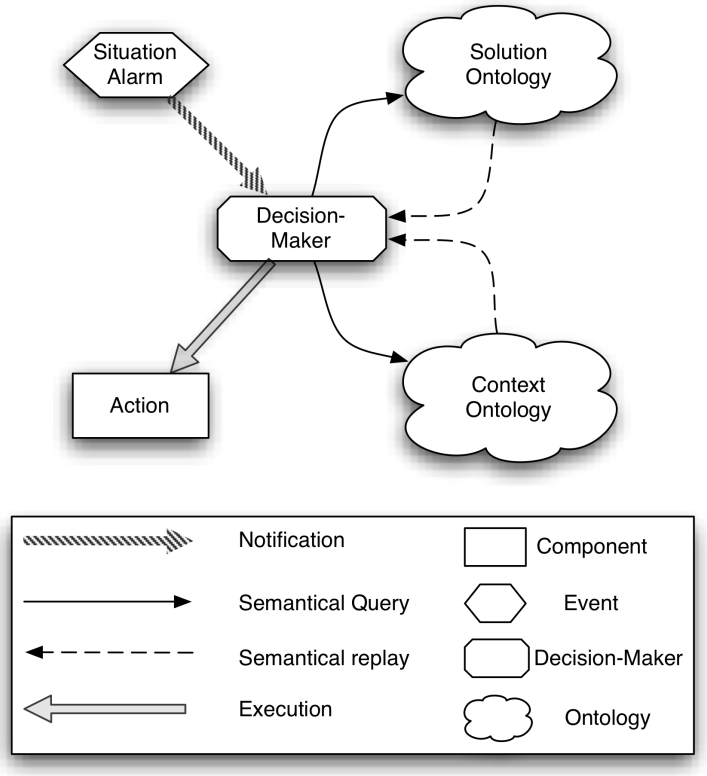

FIGURE 105 – Architecture pour la prise de décisions d'adaptation

socié aux situations d'adaptation, cf. figure.89). Il est alors chargé de la prise de décision d'adaptation pour laquelle il utilise les deux ontologies (situation et solutions) pour raisonner et trouver une solution à la notification reçue. L'ontologie de solutions contient des connaissances de solutions en rapport à des situations et à leurs causes. L'ontologie de situation contient les informations de situation y compris les "ASituation". Le DOCK (KaliCOnto plus KaliMOSA, cf. page.120) contient toutes les informations de contexte y compris les "Actions" et les "ASituations".

Lorsque le "Decision-Maker" a choisi une solution, il va déployer des "Actions" pour réaliser l'adaptation. Une solution d'adaptation contient la logique de traitement spécifique à la situation associée. Une solution est décrite de manière abstraite par des actions sémantiques. Chaque action peut avoir recours à des actions concrètes destinées à différents environnements d'exécution.

Nous présentons maintenant le processus de prise de décision d'adaptation (cf. figure.106).

Quand le "Decision-Maker" a reçu une notification ("ASituation Alarme"), il utilise le nom de la situation pour trouver les données de la situation dans le DOCK. Ensuite, il lance une requête pour trouver des solutions possibles. Les réponses à cette requête seront des "Tâches de solutions" (une tâche est une solution concrète dont les détails seront donnés dans la section suivante 13.2.1). Chaque "Tâche de solution" correspond à des "Actions" et des "Exigences d'action". Le "Decision-Maker" vérifie les "Exigences d'action" de chaque tâche dans la base de connaissances de contexte. Cette étape lui permet de trouver une solution applicable dans le contexte courant. Après un choix de solution, le "Decision-Maker" doit trouver les "Actions" pour chaque étape de la solution. Ces informations sont également disponibles dans la base

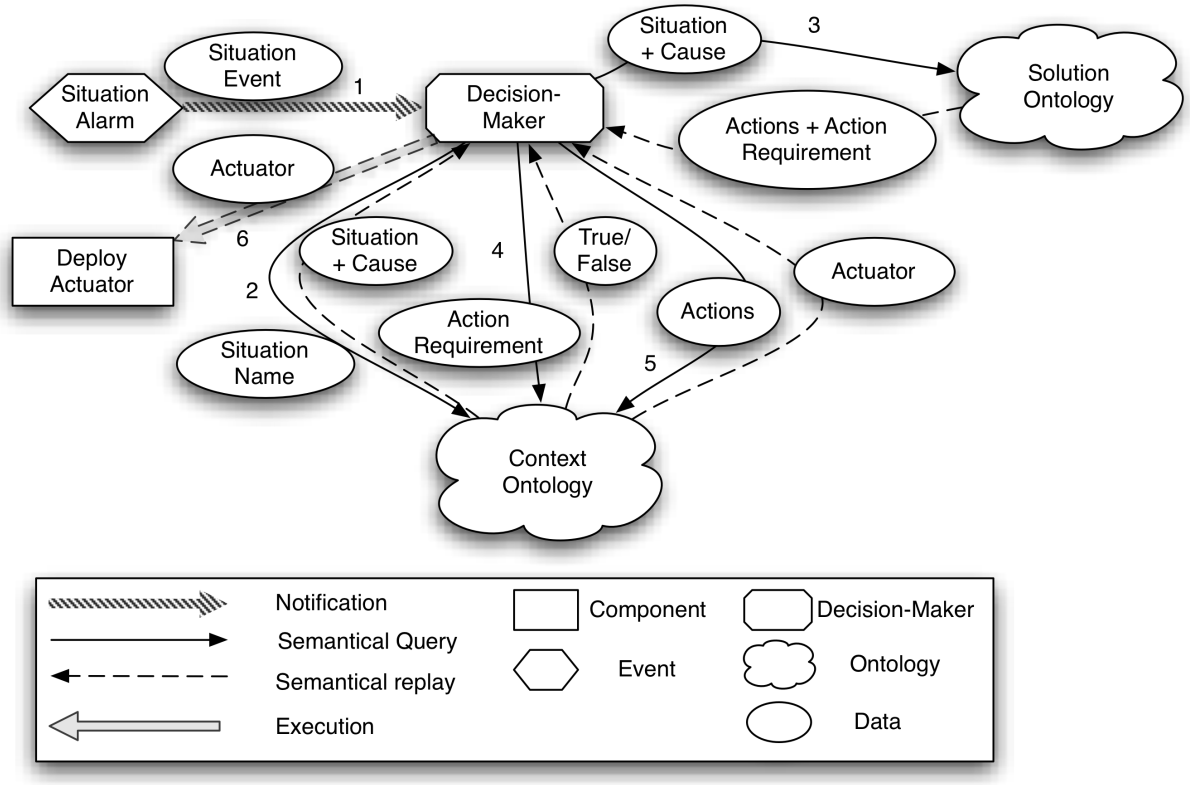

FIGURE 106 – Data flow de la prise de décision

de connaissances de contexte. Pour finir, il demande au service de déploiement de la plateforme ("Déploiement Action") d'exécuter les "Actions" liés à la solution choisie pour réaliser l'opération d'adaptation.

La section suivante détaille ce fonctionnement.

13.2.1 *Le Decision-Maker*

Le rôle du "Decision-Maker" est de trouver une solution rapidement et de résoudre la situation d'adaptation efficacement. Il est basé sur deux ontologies, l'une qui contient les connaissances du contexte actuel et les connaissance de situation (DOCK), l'autre qui contient les connaissances de solutions. Le "Decision-Maker" va filtrer, vérifier et sélectionner une solution pour la situation donnée. La solution sélectionnée n'est pas forcément la meilleure mais c'est une solution applicable dans le contexte donné.

Dans cette section, nous présentons d'abord la logique du "Decision-Maker", puis le cycle de vie d'une situation d'adaptation en lien avec la logique de décision. Dans la dernière partie, nous présentons les priorités de situations d'adaptation permettant au "Decision-Maker" de toujours traiter les situations importantes en premier.

13.2.1.1 *Logique du "Decision-Maker"*

Une situation d'adaptation peut être une situation simple ou une situation composée. Une situation composée est une composition de situations problématiques. Une situation problématique peut causer ou être causée par une ou plusieurs situations problématiques. Donc une situation composée est un graphe

de situations problématique liées par la relation "Cause". Nous organisons ce type de graphe par niveau. C'est à dire que si une situation problématique So cause deux situations S1 et S2 (cf. figure.107). So est au niveau 0 tandis que S1 et S2 sont au niveau 1.

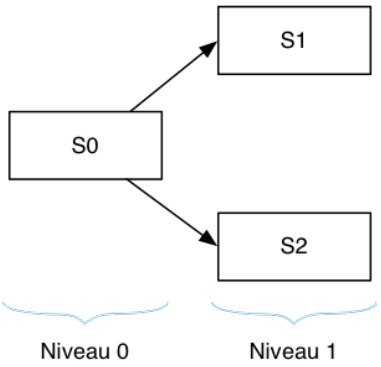

FIGURE 107 – Situation composée

Pour résoudre une situation composée il faut commencer par le plus bas niveau parce que les situations problématiques de bas niveau sont des conditions nécessaires à l'apparition des situations des niveaux supérieurs. Donc, en les résolvant il est souvent possible de résoudre celles des niveaux suivants. C'est pourquoi, nous analysons la situation composée en commençant par trouver les causes originales de la situation. Puis, nous essayons de les résoudre par une suite d'actions d'adaptation et, si nous n'y parvenons pas, nous remontons d'un niveau.

Pour résoudre une situation problématique, tout d'abord, nous regardons l'ensemble des solutions existantes. Puis nous choisissons celles applicables dans le contexte présent et retenons la première pour laquelle les actions en permettant la mise en œuvre sont disponibles. Enfin, nous appliquons cette solution. Il est clair que cette démarche ne vise pas à trouver la meilleure solution. En effet, il faudrait pour cela disposer d'informations supplémentaires permettant de comparer les solutions possibles. Malheureusement dans un contexte mouvant il n'est pas possible de prévoir si une adaptation correcte à un instant T le restera longtemps. Toutefois, dans la mesure où la plateforme mesure le contexte en permanence, tout choix pourra être remis en question à tout moment. La complexité liée à la recherche d'une solution optimale [83] se justifie d'autant moins qu'elle peut induire des temps de calculs rédhibitoires en regard des temps de mise en application d'une solution. Le risque d'oscillations du système de reconfiguration n'est pas totalement écarté mais est considérablement réduit par la prise en compte de situations issues d'informations de contexte de haut niveau. Ainsi le risque, par exemple, de reconfigurer une application pour une baisse temporaire mais brève du débit d'une liaison réseau peut être écarté si l'information de contexte retenue n'est pas la mesure instantanée du débit mais une évaluation du gradient de cette mesure sur une fenêtre ce que les chaînes de raisonnement permettent de faire.

Nous allons maintenant détailler étape par étape à partir du diagramme d'activité de notre Decision-Maker (cf. figure.108) la démarche générale de la prise de décision.

Pour prendre une décision d'adaptation le Decision-Maker suit les trois étapes suivantes :

1. Prendre en compte les notifications de situations d'adaptation
2. Trouver une solution applicable
3. Exécuter les actions

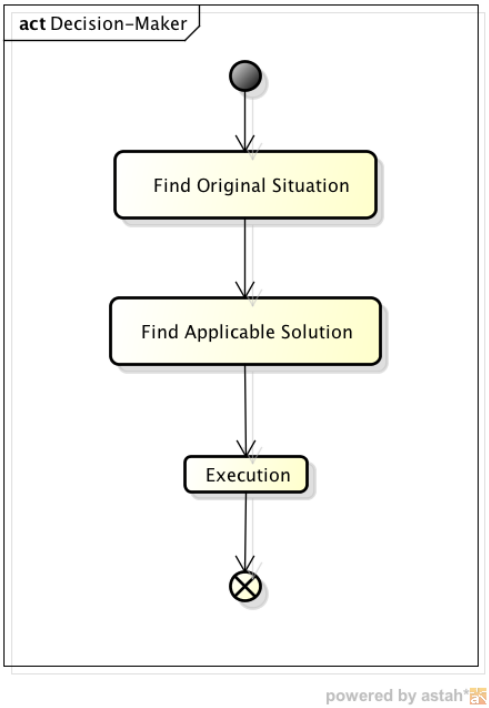

FIGURE 108 – Logique de la prise de décision

Les notifications qui parviennent au Décisionneur sont accompagnées d'une priorité qui sera décrite en (cf. section.13.2.1.3). Afin d'assurer que ces priorités sont respectées, les deux premières étapes sont interruptibles. Lorsque la situation de priorité supérieure a été traitée, elles ne sont pas reprises au point où elles ont été interrompues mais le Décisionneur retourne au début de la première étape. En effet, la prise en compte de la situation de priorité supérieure a modifié le contexte de sorte qu'il est possible que le traitement suspendu n'ait plus lieu d'être. Au contraire, les étapes d'exécution et de vérification sont atomiques car, dans le cas contraire, on risquerait de rendre l'application non opérationnelle en raison d'une reconfiguration partiellement exécutée et vérifiée.

Étape 1 : nous savons qu'une situation d'adaptation peut être composée de situations problématiques. Donc le Decision-Maker doit, tout d'abord, trouver les causes originales d'une situation composée, et tenter de résoudre ces causes. Le Decision-Maker consulte l'ontologie de contexte via l'API OWL pour construire le graphe de la situation composée et pour identifier les situations problématiques. Pendant ce temps il prend en compte les nouvelles situations notifiées. Certaines peuvent suspendre le traitement en cours en raison de la priorité des situations.

Étape 2 : nous utilisons une ontologie de solutions pour représenter les connaissances des solutions d'adaptation. Le Decision-Maker consulte cette ontologie par l'API OWL et par DL Query pour la recherche de solutions. Pour une situation donnée il peut trouver plusieurs solutions, il compare alors les exigences d'action des solutions avec le contexte actuel pour éliminer les solutions inapplicables. Pour finir, le Decision-Maker va vérifier les disponibilités des actions nécessaires à la mise en œuvre de la solution par rapport au contexte actuel. La première solution satisfaite va être choisie et traitée dans l'étape suivante.

Étape 3, le Decision-Maker doit trouver les actionneur pour chaque action de la solution. Il compare les conditions d'exécution des actions avec les informations contextuelles de disponibilité des entités engagées pour vérifier la faisabilité de la solution. Il récupère ces informations via l'API OWL dans l'ontologie de

solutions. Ceci lui permet de choisir et de configurer les actionneurs (version, destination de déploiement, services impliqués, etc.). Lorsque la phase d'exécution démarrera, les opérations seront atomiques

La section suivante présente le cycle de vie d'une situation d'adaptation par rapport au processus de prise de décision.

13.2.1.2 *Cycle de vie d'une situation d'adaptation*

Nous avons identifié cinq états de situation d'adaptation : Identifiée, Cause Originale, En traitement, Résolue et Non résolue (cf. figure.109).

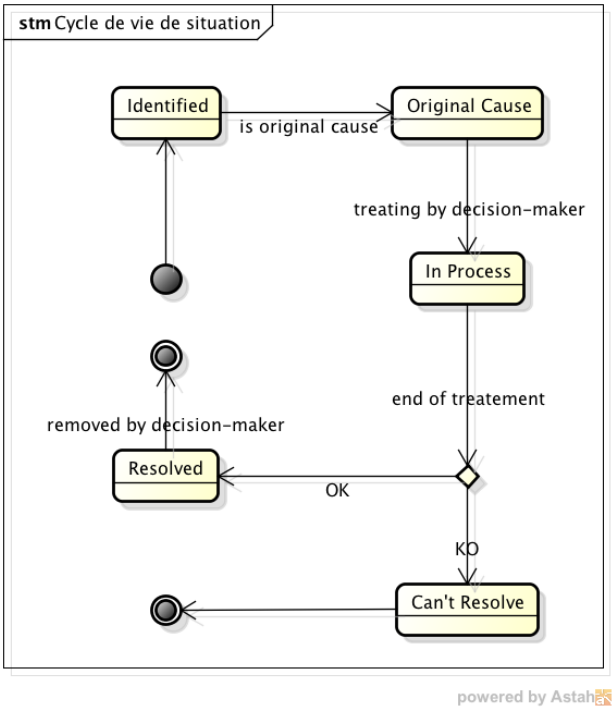

FIGURE 109 – Cycle de vie d'une situation

Une situation contextuelle peut être identifiée comme une situation problématique par les raisonnements d'identification de situation d'adaptation de la plateforme. Une fois qu'une situation contextuelle est identifiée comme une problématique, elle va être notifiée au Decision-Maker et être mise dans l'état "Identifié". Ensuite, le processus de prise de décision va trouver les causes originales de la situation notifiée et ces causes vont passer dans l'état "Cause Originale". Seules les situations dans l'état "Cause Originale" sont traitées par le Decision-Maker. Si une situation notifiée n'a aucune cause originale c'est la situation notifiée elle-même qui est considérée comme une "Cause Originale".

Une situation en cours de traitement est mise dans l'état "En traitement". Durant le traitement d'une situation d'adaptation (une situation d'adaptation peut être une simple situation problématique ou une situation composée c'est à dire contenant plusieurs situations problématiques de différents niveaux liées par une relation "cause"), le Decision-Maker peut continuer de recevoir de nouvelles situations notifiées.

À la fin des traitements, la situation adoptera l'un des deux états : "Résolue" ou "Non résolue". Les situations résolues sont retirées de l'ontologie par le Decision-Maker. Lorsque les solutions dont nous disposons ne permettent de résoudre la situation, elle devient "Non résolue". La plateforme ne peut rien

faire pour le moment mais une situation irrésolue peut être ultérieurement résolue par des changements de contexte, par exemple, une situation d'"Hôte Perdu" en raison d'un niveau de signal wifi insuffisant peut être résolue par un rétablissement de la connexion wifi.

Ce cycle de vie des situations permet leur traitement par le Decision-Maker. La section suivante détaille le système de priorité des situations d'adaptation qui permet au Decision-Maker de traiter les situations plus urgentes en premier.

13.2.1.3 *La priorité des situations d'adaptation*

Durant le traitement de situations d'adaptation peuvent survenir des situations plus importantes que celles en cours de traitement. Dans ce cas nous devons interrompre les traitements en cours pour traiter ces situations prioritaires afin de garantir l'efficacité de la prise de décision d'adaptation.

Comment définir les niveaux de priorité ? Dans notre architecture la plateforme est la base de toutes les applications. Si la plateforme ne marche plus, aucune opération d'adaptation ne peut être effectuée. Ceci justifie que toutes les situations concernant la plateforme elle-même ont le plus haut niveau de priorité. Dans tous les cas, il faut les traiter en premier.

Le deuxième niveau concerne les souhaits de l'utilisateur. Ces situations sont causées par des demandes de l'utilisateur et ont le deuxième niveau de priorité. Par exemple, si une situation problématique cause un redéploiement de service de l'application, mais qu'avant l'exécution de ce redéploiement, l'utilisateur demande l'arrêt de ce service il est clair que la mise en œuvre du redéploiement n'a plus lieu d'être.

Les situations d'application sont placées au troisième niveau de priorité. Elles sont définies par l'application et représentent des besoins spécifiques liés à logique métier de l'application. Nous considérons qu'elles sont moins urgentes que les demandes des utilisateurs dans la mesure où leurs conséquences sont moins directement perceptibles par ces derniers.

Enfin les situations d'adaptation générales liées à la disponibilité des ressources sont mises au dernier niveau de priorité. Ce choix peut paraître contestable dans la mesure où la non disponibilité de ressources peut compromettre de bon déroulement de l'application. Toutefois nous considérons que, dans la mesure où les applications classiques ne résolvent pas du tout ce problème, le fait de ne le prendre en compte qu'en dernier ne place pas les utilisateurs dans des conditions moins acceptables que celles qu'ils ont l'habitude de connaître. Par ailleurs la possibilité de détecter des situations de haut niveau par les chaînes de raisonnement diminue sensiblement le nombre de situations à traiter. De plus la mise en œuvre d'une reconfiguration est une opération relativement rapide (quelques secondes) [41] . Pour ces deux raisons nous considérons que les conséquences de cette priorité minimale ne sont pas inacceptables par les utilisateurs.

13.2.2 *Ontologie de solutions d'adaptation*

L'ontologie de solutions d'adaptation (cf. figure.110) réutilise deux concepts extraits de celle de connaissance du contexte. Ce sont le concept de "ASituation" et celui d'"Action". Le rôle de cette ontologie est de représenter la relation entre une "ASituation" et ses solutions possibles. Une "Solution" est identifiée par les concepts de "ASituation" et de "Cause".

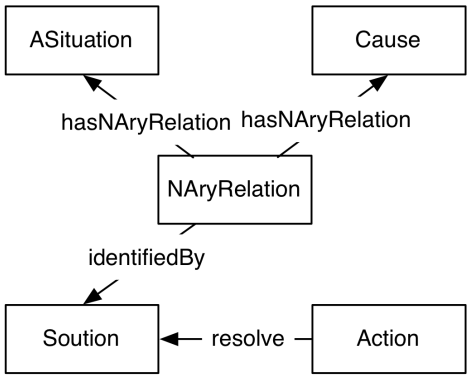

FIGURE 110 – Ontologie de solutions

Comme nous l'avons vu à la page 120, une "ASituation" est une situation de haut niveau d'abstraction comme "Hôte Perdu" ou "Ressource Perdue". Les "Causes" précisent d'où vient le problème. Par exemple, une situation "Ressource Perdue" peut être due à une cause "Batterie faible" sur un hôte. A partir de ces connaissances, nous pouvons trouver différentes solutions pour une même situation et les mêmes causes. Par exemple, pour résoudre la situation "Ressource Perdue" avec pour cause "Batterie faible", nous pouvons migrer certains composants en cours d'exécution vers un autre hôte ou nous pouvons baisser les QoS de certains services d'application pour économiser l'énergie.

Une "ASituation" peut donc avoir plusieurs solutions, chacune étant réalisée par une "Tâche de solution". Une "Tâche de solution" peut avoir une ou plusieurs "Action". Pour les situations ayant plusieurs solutions, nous prenons la première applicable. Par exemple, si dans l'exemple précédent la migration de composants est choisie mais que le contexte courant ne le permet pas parce qu'il n'y a pas d'autre hôte disponible pour cet utilisateur, la solution de changement de QoS sera adoptée.

13.2.3 *Les actions*

Une action est définie par un nom d'action et une exigence d'action, par exemple, à l'action de migration est associée l'exigence d'action de disposer d'un autre la exigence compatible dans le domaine de l'utilisateur. Une action est effectuée par un ou plusieurs actionneurs. Elle peut contenir des sous actions auxquelles sont associées des exigences d'action supplémentaires à celles de l'action père. Les actions sont extensibles dans la mesure où les actionneurs sont des composants Kalimucho que le concepteur peut définir pour des besoins spécifiques. Toutefois un cœur d'actionneurs a été défini qui correspond aux actions de base de la plateforme.

Kalimucho connaît des opérations basiques de reconfiguration au niveau des composants et des connecteurs :

— créer/supprimer/migrer un composant
— créer/supprimer un connecteur
— connecter/déconnecter une entrée ou une sortie d'un composant à un connecteur

C'est à partir de ces opérations de base que sont définis les actions permettant la mise en œuvre des prises de décision, il s'agit des actions suivantes :

— Déployer
— Supprimer
— Migrer
— Changer la QoS

Chaque action correspond à un (re)déploiement qui met en œuvre le plan de déploiement trouvé par le Décisionneur en fonction des ressources actuelles et des exigences logicielles

Par exemple, le service S1 nécessite que l'hôte destinataire ait au moins 20Mo de mémoire libre tandis que le service S2 a besoin d'au moins 30Mo de mémoire pour son exécution. Au moment de la planification, il existe, dans le domaine de l'utilisateur, un hôte D1 ayant 40Mo de mémoire libre. L'unicité du Décisionneur dans le domaine de l'utilisateur et la réalisation séquentielle des actions de reconfiguration sont des conditions nécessaires à la faisabilité de la solution retenue. En effet, si plusieurs solutions s'exécutaient en parallèle, elles pourraient choisir de migrer S1 et S2 sur D1 parce que l'état de D1 satisfait les exigences de mémoire de S1 et de S2. En revanche, une fois l'un des deux services a été déployé, l'autre va manquer mémoire. En évaluant les situations en un point unique du domaine et de façon séquentielle ce problème disparait puisque dès que S1 aura été déployé sur D1 la planification suivante choisira de déployer S2 sur un autre hôte ayant au moins 30Mo de mémoire libre.

L'action "Déployer" consiste à déployer de nouveaux composants de l'application sur des hôtes disponibles et à les relier par de nouveaux connecteurs. Kalimucho est capable de déployer un composant sur tout hôte exécutant la plateforme (KaliHôte) toutefois notre démarche limite les reconfigurations aux ressources disponibles dans le domaine d'un utilisateur.

Le placement des composants sur les différents hôtes du domaine de l'utilisateur tient compte des contraintes d'interactivité. Les composants sont décrits dans l'ontologie de contexte et parmi les informations les concernant celle qui nous intéresse ici concerne l'interactivité. Lorsqu'un composant est décrit par son concepteur comme interactif (propriété statique du composant) cela signifie qu'il ne peut être déployé que sur l'un des hôtes du domaine avec lequel cet utilisateur est en lien direct (cf. figure.46). Par ailleurs, l'ontologie de contexte permet de savoir avec quels hôtes l'utilisateur interagit actuellement (propriété dynamique) comme indiqué dans la figure.38. Les autres composants peuvent être déployés sur un hôte quelconque du domaine.

L'"Actionneur" (Actuator) effectue une planification pour savoir quel composant va être déployé sur quel hôte, celle-ci est limitée aux hôtes du domaine de l'utilisateur et, pour les composants interactifs, aux hôtes en lien direct avec l'utilisateur. Cette planification est calculée pendant l'exécution selon les exigences des composants et les disponibilités des ressources. Ensuite l' "Actionneur" va demander à la plateforme "Kalimucho" de réaliser l'action par envoi de commandes de déploiement de composants aux hôtes identifiés (CreateComponent). Pour finir, l' "Actionneur" établira les liens entre ces composants par création de connecteurs (commande CreateConnector de Kalimucho).

L'action "Supprimer" permet d'arrêter des composants en cours d'exécution. C'est une action simple. Son "Actionneur" va demander à la plateforme "Kalimucho" d'envoyer la commande correspondant à la suppression d'un composant aux hôtes sur lesquels ces composants sont en hébergés (RemoveComponent). Il terminera par la suppression des connecteurs reliés à ces composants (commande RemoveConnector de Kalimucho).

L'action "Migrer" est une action permettant de déplacer un composant d'un hôte vers un autre en conservant son état d'exécution. La migration ressemble à l'action de déploiement. Le choix du nouvel hôte d'accueil du composant est donc similaire et tient compte du fait que le composant est ou non interactif. Dès que l'hôte d'accueil est déterminé il suffit d'envoyer une commande à la plateforme accueillant actuellement le composant à migrer (SendComponent) en lui indiquant ce nouvel hôte d'accueil. La redirection des connecteurs liés à ce composant en fonction de sa nouvelle localisation est prise en compte par Kalimucho.

L'action "Changer QoS" est une action permettant de modifier la QoS d'un service. Cette action se concrétise par une comparaison entre la configuration du service tel qu'il est actuellement implémenté et la nouvelle configuration de ce service. Elle se traduit par des opérations de déploiement de nouveaux composants et connecteurs, de suppression de composants et connecteurs existants et de déplacements de composants d'un hôte à un autre. Elle peut donc être considéré comme une combinaison des trois actions précédentes visant à transformer une configuration de service en une autre.

Cette liste d'"Actions" correspond à celle utilisée par la plateforme pour les reconfigurations automatiques. Elle peut être étendue par un concepteur d'application pour des actions spécifiques gérées par l'application elle-même. Un concepteur peut, par exemple, proposer une action de modification de la QoS nécessitant l'intervention de l'utilisateur pour guider le choix de déploiement. Il lui suffit de créer un "Actionneur" spécifique, de l'ajouter à l'ontologie de solutions d'adaptation et de le lier à une ou plusieurs "Tâches de solution". Ensuite il établit, dans l'ontologie de contexte, la relation entre cet "Action" et une "Action" qu'il développe. Ceci peut être effectué par des éditeurs d'ontologie comme Protége, ou par des API comme OWL API, ou par des éditeurs de texte.

Les adaptations ne concernent pas seulement l'application mais également la plateforme elle-même. Dans la section suivante nous présentons les solutions spécifiques aux situations provoquant des adaptations de la plateforme.

13.2.4 *Exemple*

Nous reprenons l'exemple du chapitre Kali-MOSA (cf. page.131). Pour expliquer comment le Kali-Adapt réagit à la réception des notifications. Dans l'exemple, il y a deux scénarios. Dans le premier, il va recevoir une notification de la situation "Low Energy Power". Dans le deuxième, Kali-Adapt va recevoir plusieurs notifications pour les situations suivantes : "Device Lost", "Component Lost", "Service Dysfunction", et "Application Dysfunction".

Quand Kali-Adapt a reçu la notification de "Low Energy Power", il interroge le DOCK pour récupérer l'instance de la situation et ses causes. Il recevra un instance de la situation "Low Energy Power" comme résultat de la requête. La situation a une source "Device A" et aucune cause. Kali-Adapt recherche directement une solution dans l'ontologie de solution, parce que cette situation problématique est la racine de la situation d'adaptation. Dans ce cas il n'existe qu'une seule solution pour cette situation : "Migrate All Components of Device Solution (MACD)". Après la réception de la solution, Kali-Adapt vérifie l'exigence d'exécution de la solution "Any Available Device" (c'est une requête en format DL Query). Supposons qu'il existe deux hôte fonctionnels : "Device B" et "Device C". Kali-Adapt cherche l'actionneur ("Actuator") pour

l'action "MACD Action" qui utilise "Migration Action" dans le DOCK et vérifie leur exigence d'exéction. Et puis il exécute l'actionneur de "MACD Action".

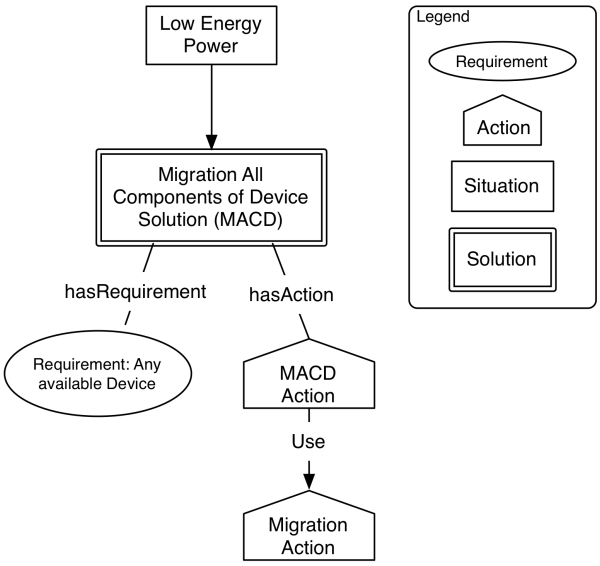

FIGURE 111 – Solution pour la situation "Low Energy Power"

Le scénario 2 diffère du scénario 1 seulement par la première étape. Lors de la prise en compte des notifications de situations d'adaptation, Kali-Adapt va récupérer un graphe de situations d'adaptation (cf. figure.101). Il utilisera ce graphe pour trouver la situation originale. Dans ce cas, la première trouvée est "Low Energy Power" qui a été traitée dans le scénario 1. Il avance donc sur le graphe et trouve "Device Lost" avec comme cause "Low Energy Power" (cf. figure.112). Pour une situation "Device Lost" il peut exister des solutions lorsqu'il s'agit d'un problème de réseau mais pas lorsque la cause en est la batterie. Kali-Adapt va donc avancer d'un niveau supplémentaire dans le graphe et trouvera "Composant Dysfunction" pour lequel est proposée la solution "Component Redeployment Solution" (cf. figure.112). C'est donc cette solution qu'il va tenter d'appliquer. Si c'est possible le processus d'adaptation est terminé sinon il pourra envisager la situation suivante "Service Dysfunction".

13.2.5 *Situations d'adaptation de la plateforme*

Rappelons que la prise de décision se fait sur l'un des hôtes du domaine de l'utilisateur et que l'ontologie de contexte est hébergée sur cet hôte. En revanche, les connaissances dynamiques propres à chacun des hôtes sont stockées et maintenues à jour par des chaînes de raisonnement sur ce même hôte. La prise de décision fait donc appel aux connaissances de contexte, de situations et de solutions mais est susceptible de récupérer des informations localement sur chacun des hôtes du domaine. Pour que l'adaptation puisse fonctionner il est donc nécessaire que l'hôte gérant la prise de décision (DECK) et les ontologies qu'il utilise (DOCK) soient en permanence opérationnelles. Comme un hôte peut disparaître à tout moment du domaine (perte d'énergie, déconnexion du réseau, etc.), la plateforme doit mettre en œuvre des mécanismes d'auto-reconfiguration pour maintenir le service de reconfiguration opérationnel.

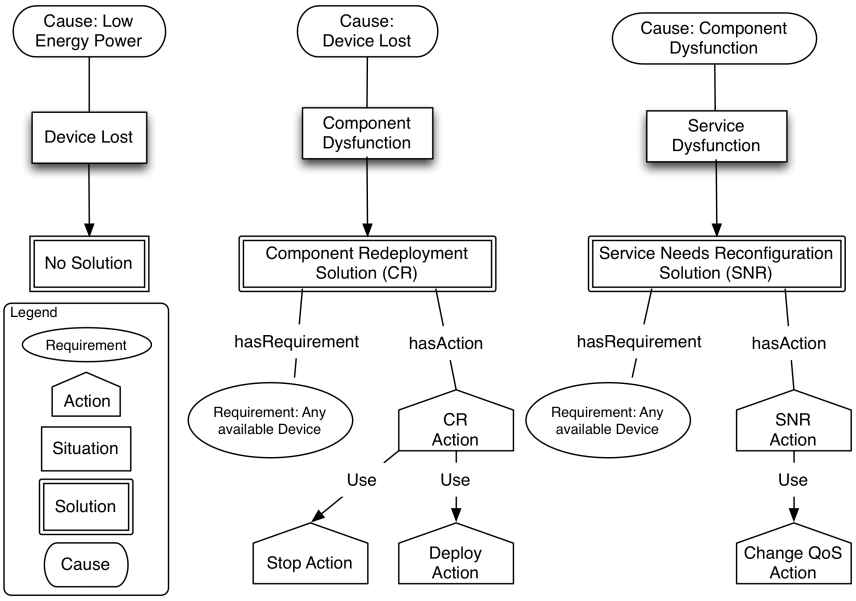

FIGURE 112 – Les solutions du scénario 2

Dans cette section nous présentons les situations d'adaptation de la plateforme. Ces situations sont influencées par la mobilité de l'utilisateur. Notre plateforme est conçue pour faire de l'adaptation supervisée à partir d'un décisionnaire unique par domaine d'utilisateur. Il ne s'agit pour autant pas d'une architecture centralisée traditionnelle car nous n'avons pas besoin d'un serveur centralisé fixe. Le middleware Kali2Much est complètement distribué et les raisonnements de contexte sont répartis sur chaque hôte. Nous avons, dans ce but, conçu un réseau d'ontologies multicouches avec une dépendance faible entre les informations de contexte et les situations sémantiques ainsi qu'entre les situations des hôtes et celles du domaine. Les services de la plateforme sont construits à partir de composants Osagaia et de connecteurs Korrontea de sorte qu'ils sont adaptables, comme le sont les applications, selon le contexte actuel. Le conteneur d'ontologie peut être déplacé durant l'exécution de la plateforme, de même le décisionnaire peut être interrompu et déplacé. Grâce à cette architecture, nous pouvons traiter les situations causées par la mobilité de l'utilisateur.

Dans un domaine d'adaptation, peuvent se produire des situations comme des apparitions ou des disparitions imprévisibles d'hôtes. C'est pourquoi chaque hôte du domaine doit pouvoir prendre des décisions d'adaptation et reconstruire un domaine d'adaptation lorsque les services deviennent indisponibles. Pour ce faire, chaque hôte héberge un service de déploiement du décisionnaire et du DOCK localement. Nous appelons ce service "Device Superviser", il prend en compte les situations suivantes : Utilisateur perdu, Utilisateur isolé, perte du DOCK (cf. page.70). Dans chacun de ces cas, il peut redéployer le décisionnaire et le DOCK.

Perte du DOCK

Le service DOCK contient toutes les connaissances du domaine, il est indispensable au fonctionnement de la plateforme. Toutefois la plateforme peut perdre ce service en cours d'exécution par exemple si l'hôte qui l'héberge perd sa connexion au réseau.

Lorsqu'une telle situation est détectée, par un hôte, celui-ci va lancer un service de DOCK local. Ce service deviendra le service de DOCK du domaine. Toutefois comme cette décision est prise localement il

est possible que plusieurs DOCK apparaissent simultanément sur le domaine. C'est pourquoi après une telle situation des situations de Multi-DOCK peuvent être trouvées.

Multi-DOCK

Comme nous venons de le voir, la situation de Multi-DOCK peut être rencontrée après une disparition de DOCK. Elle peut également se produite quand l'utilisateur retrouve un ancien DOCK qui avait été perdu en raison d'une reconnexion réseau. Dans ce cas il y aurait au moins deux DOCK dans un même domaine de l'utilisateur ce qui rendrait la prise de décision d'adaptation inopérante.

La solution retenue pour cette situation est de comparer les hôtes qui hébergent un service de DOCK pour choisir le meilleur hôte pour ce service (celui offrant les meilleures conditions de ressources d'énergie, de CPU et de mémoire). Il suffit alors de stopper les autres. En pratique, lors du lancement du service de DOCK un message de notification "DOCK Info" est envoyé à tous les hôtes du domaine. Ce message contient les caractéristiques de l'hôte qui l'envoie. Tout hôte qui héberge un DOCK et reçoit ce message identifie la situation Multi-DOCK. La solution de cette situation est "Multi-DOCK Solution" liée à l'action "DOCK Device Comparison Action" qui compare les caractéristique de l'émetteur à celles de l'hôte local. Si le résultat montre que le les caractéristiques de l'émetteur sont meilleures, l'hôte local lance l'action "Stop Action" pour arrêter le DOCK. Ainsi à la fin, il ne reste qu'un seul DOCK dans le domaine.

Utilisateur perdu

La situation d'adaptation "Utilisateur perdu" est définie par l'absence d'interactions avec un utilisateur alors des applications sont en cours ou par le fait que l'utilisateur passe en mode isolé (perte de connexion).

Dans cette situation, aucune reconfiguration n'est mise en oeuvre et la plateforme se contente de maintenir les connexions ouvertes.

Utilisateur isolé

Un utilisateur isolé correspond à une situation d'adaptation où un utilisateur a perdu le lien avec son domaine et donc avec les services principaux. Son hôte détectera donc la situation DOCK perdu et reconstruira un nouveau service localement.

13.3 CONCLUSION

Dans ce chapitre nous avons présenté notre architecture de prise de décision d'adaptation. Cette architecture permet d'avoir une faible dépendance entre la construction des solutions et la décision logique. Elle permet également le choix des solutions et l'exécution des actions en fonction du contexte actuel.

Nous avons présenté le décisionnaire, la logique de prise de décision, le cycle de vie d'une situation d'adaptation et la priorité des situations. Tout ceci fonctionne autour de l'ontologie de solutions d'adaptation liées à des actions mises en œuvre par les actionneurs principaux de la plateforme. Ces mécanismes sont utilisés tant pour adapter l'application que la plateforme elle-même.

Notre proposition présente les avantages suivants :

— Nombre de déclenchements réduit : Le processus de prise de décision ne se déclenche pas à chaque changement de contexte mais seulement lorsque des situations bien identifiés sont détectées par les chaînes de raisonnement. Les interprétations du contexte sont réparties sur chaque hôte du

domaine d'adaptation mais le raisonnement sur les situations est effectué sur l'hôte le plus stable et plus puissant du domaine.

— Possibilité de prendre une décision rapide : dans certaines situations, il peut exister des solutions meilleures que celle que nous mettons en œuvre mais dans un contexte mouvant il nous semble préférable de choisir une solution acceptable rapidement.

— Extensibilité des situations sémantiques faciles à comprendre et à interpréter : la représentation des situations est basé sur une ontologie sous la forme OWL/RDF et de fichiers XML. L'ontologie peut être modifiée en cours d'exécution sans influencer le fonctionnement de la plateforme. Les situations sémantiques peuvent être partagées entre la plateforme et les applications. C'est à dire que les développeurs des applications peuvent ajouter une nouvelle situation dans l'ontologie pour des cas spécifiques liée à la logique métier de l'application.

— Situations sémantiques cohérentes : chaque mise à jour d'instance de situation est soumise à un DL raisonneur [1] pour vérifier la cohérence entre les situations ajoutées.

1. DL reasonner : http ://protegewiki.stanford.edu/wiki/Using_Reasoners

Troisième partie

CONCLUSION ET PERSPECTIVES

14

CONCLUSION

L'objectif de ce travail concerne la prise en compte du contexte en vue de la reconfiguration automatique des applications en cours d'utilisation. L'un des types d'applications visés sont les "long life applications" capables de suivre l'utilisateur dans son quotidien pour lui offrir, à tout moment, les services utiles et ceci dans les meilleures conditions possibles. Toutefois l'Internet of the Things (IoT) relève d'un autre mode d'adaptation lié à la faible capacité des dispositifs mis en œuvre. Il s'agit là d'applications opportunistes de courte durée de vie qui doivent pouvoir être déployées et maintenues en fonctionnement un certain temps puis disparaître.

Notre approche se base sur la plateforme Kalimucho développée dans le laboratoire. Cette plateforme prend en charge le déploiement dynamique de composants et leur interconnexion au travers des réseaux disponibles. Elle permet de constituer des architectures d'application à base de composants réparties sur plusieurs périphériques puis de les modifier par ajout/suppression de composants et de connecteurs ainsi que par migration de composants au runtime. Elle constitue donc une base technique complète pour la prise en compte du contexte par reconfiguration dynamique d'applications.

Notre approche est volontairement centrée sur la qualité de service perçue par les utilisateurs ce qui nous a conduits à définir la notion de domaine de l'utilisateur qui comprend la totalité des ressources auxquelles il a accès. C'est à l'intérieur de chacun de ces domaines que s'effectuent les reconfigurations. L'ensemble des ressources d'un domaine devient ainsi une machine virtuelle à l'entière disposition de l'utilisateur. Le rôle de la plateforme est d'utiliser au mieux ces ressources pour offrir à l'utilisateur les bons services au bon moment et assurer que ces services restent opérationnels.

Si la plateforme peut automatiquement prendre des décisions de reconfigurations en fonction du contexte il n'est pas envisageable de se limiter à cette seule façon de procéder. En effet la plateforme ne peut avoir qu'une vue limitée de la sémantique de l'application (celle que peut lui en fournir le concepteur) ce qui ne lui permet pas d'assurer de prendre la totalité des décisions nécessaires à la satisfaction des utilisateurs de façon automatique. C'est la raison pour laquelle nous avons choisi de faire en sorte que les mécanismes utilisés par la plateforme pour capturer, interpréter et analyser le contexte puis pour décider des reconfigurations adaptées soient totalement accessibles et extensibles par l'application elle-même.

Le choix de définir un domaine comme une machine virtuelle répartie sur laquelle la plateforme exerce son rôle de supervision de l'application nous a naturellement conduits à étendre à la plateforme elle-même la possibilité de reconfiguration. En effet les ressources disponibles sur un domaine sont par nature instables en raison de la mobilité, de la bande passante, de l'énergie disponible, etc. Dans la mesure où la plateforme s'exécute sur ces ressources elle est soumise à ces aléas qui pourraient la rendre

non opérationnelle. Il est donc impératif qu'elle puisse s'appliquer à elle-même les mêmes opérations de reconfiguration que celles qu'elle exerce sur l'application.

Enfin la prise en compte du contexte se concrétise par un ensemble de mesures de l'environnement de l'application (environnement physique, logiciel, de l'utilisateur, etc.). Son analyse peut faire appel à des opérations simples comme des seuils ou des gradients ou à des opérations plus complexes comme des corrélations entre différentes valeurs dans l'espace et/ou dans le temps. Toutefois, lorsque cette analyse doit conduire à une prise de décision de reconfiguration, il est impératif de pouvoir considérer ces mesures non plus comme un ensemble de valeurs mais comme représentatives d'une situation ayant un sens du point de vue de l'objectif visé c'est à dire de la satisfaction de l'usager. Si une telle interprétation est possible pour une application donnée ou pour des situations critiques parfaitement définies elle devient impossible pour une plateforme ayant vocation à supporter tout type d'applications. Ce constat nous a conduits à proposer des mécanismes de transformation des mesures de bas niveau en informations de haut niveau sémantique. Nous distinguons deux types d'informations de contexte de haut niveau : celles que la plateforme peut gérer de façon automatique pour tout type d'application et celles qui ne peuvent être définies que par chacune des applications. Ainsi le concepteur d'application peut enrichir l'ontologie de situations utilisée pour la prise décision par ses propres situations. Tant l'analyse du contexte en vue de la création de situations pertinentes que la prise en compte de ces situations pour reconfigurer l'application ou la plateforme font appel à des chaînes de raisonnement réalisées par un enchaînement de tâches contenues dans des composants Kalimucho. Ceci permet la mise en œuvre de tout type d'analyse aussi complexe soit-il.

14.1 LA PLATEFORME KALIMUCHO-A

La plateforme d'adaptation : Kalimucho Adaptation Plateforme (Kalimucho-A) fonctionne dans le domaine d'adaptation de l'utilisateur. Nous rappelons ici la définition du domaine d'adaptation : pour chaque utilisateur, un domaine d'adaptation qui contient l'ensemble des ressources auxquelles il peut accéder sans limite. Ces ressources sont celles qu'il a accepté de rendre disponibles sur ses machines pour lui-même et celles que les autres utilisateurs ont accepté de partager.

Nous avons conçu le modèle de contexte KaliCOnto spécifiquement pour partager la même structure de connaissances sur le contexte dans le domaine d'adaptation entre les hôtes, les applications et les instances de la plateforme. Ce modèle de contexte est conçu pour la plateforme mais est partagé par les applications pour leur permettre d'être context-aware.

Dans le chapitre "Ontologie de contexte de Kalimucho (KALICONTO)" 9, nous avons présenté ce modèle de contexte basé sur une ontologie. KaliCOnto peut se classer dans trois catégories : c'est à la fois une ontologie d'application, une ontologie de domaine et une ontologie générique (voir figure.54). Durant l'exécution, nous utilisons deux ontologies : le "DEvice Context Knowledge" (DECK) restreint au contexte local à un périphérique et le "DOmain Context Knowledge" (DOCK) représentant le contexte global du domaine. Bien enetendu, il n'y a qu'un seul DOCK par domaine d'adaptation tandis que chaque hôte possède un DECK. Ils sont encapsulés dans un "Device Ontology Container" ou un "Domain Ontology Container" qui en assure le fonctionnement. Le "Device Ontology Container" contient les états d'exécution et les descrip-

tions des ressources qui appartiennent à l'hôte ("KaliDevice"). Quand un nouveau "KaliDevice" entre dans un domaine d'adaptation, il ajoute sa description matérielle au "Domain Ontology Container". KaliCOnto est alimentée par la plateforme et par les applications qui y enregistrent leurs informations contextuelles de haut niveau. La plateforme y met les informations qui lui permettront de faire des reconfigurations contextuelles. De plus, chaque hôte alimente son DECK par les résultats des capteurs formant le middleware de contexte Kali2Much (cf. chapitre.10). Le DOCK est, quant à lui, alimenté par tous les hôtes du domaine.

Le middleware de contexte Kali2Much constitue la couche de gestion de l'acquisition du contexte. Son rôle essentiel est de permettre la récupération des informations contextuelles brutes de tout type. Toutefois l'utilisation de capteurs dans le domaine de l'informatique mobile soulève une problématique spécifique liée au dynamisme. En effet, les capteurs peuvent disparaître ou apparaître à tout moment de façon imprévisible et le service doit être capable de gérer ce dynamisme. Pour les mêmes raisons se pose aussi le problème de la complétude des données récupérées que nous résolvons par le mécanisme d'analyse de ces données par des chaînes de raisonnement. De plus, le middleware de contexte prend également en charge la distribution des données captées à l'intérieur du domaine.

Grâce à lui l'utilisateur peut accéder à la totalité des ressources partagées du domaine. En effet, Kali2Much supporte un fonctionnement multi-consommateurs et permet l'accès à des données captées à distance ainsi que le filtrage et la transformation des informations. Il propose plusieurs types de "Context-Collector" programmables permettant de répondre aux besoins les plus complexes. L'API unifiée de Kali2Much utilise un méta-modèle pour masquer la complexité et l'hétérogénéité des capteurs de contexte dans un environnement pervasif et mobile. Enfin, Kali2Much offre un service de recherche qui permet aux consommateurs de trouver les capteurs actuellement disponibles dans un lieu donné. Ce service supporte la recherche de données contextuelles statiques, dynamiques et sous forme de flux. Un langage de requêtes a été défini pour construire automatiquement la chaîne de collecte (KaliSensor + Context Collector + Context Transformer) et transférer directement les données au consommateur par des connecteurs de la plateforme Kalimucho. A partir de ces mécanismes d'acquisition du contexte des chaînes de raisonnement pour constituer les informations de contexte de haut niveau peuvent être mises en place.

Les raisonnements logiques fournis par OWL permettent de vérifier la cohérence et de raisonner sur les données mais pas de faire des raisonnements complexes qui demandent de calculer ou de communiquer avec d'autres services (par exemple un WebService pour récupérer la carte d'un lieu que l'on souhaite comparer à la position GPS de l'utilisateur). C'est pourquoi nous avons choisi d'utiliser les chaînes de raisonnement présentées dans le chapitre.11 "Raisonnement de contexte information (Kali-Reason)".

Notre architecture permet aux applications d'étendre les ontologies et de mettre en place des chaînes de raisonnement pour traiter les informations de contexte au niveau sémantique adéquat. La plateforme Kalimucho-A les utilise ensuite durant l'exécution. Pour des raisons de dynamique et d'unification de construction, nous avons conçu des chaînes de raisonnement ("Reasoner Chain") basées sur le langage orchestration de services BPMN 2.0 et un moteur d'exécution BPMN "Reasoning Chain Engine" (RCE) basé sur Activiti. Les tâches déclenchées à partir des indications fournies par les fichiers BPMN sont réalisées par des composants Osagaia ce qui permet à la plateforme Kalimucho d'en assurer le contrôle total. Elle peut ainsi les créer, les interconnecter, les lancer, les arrêter et les supprimer. Par ailleurs, faisant partie

des composants métiers gérés par la plateforme, ils peuvent être inclus dans le processus d'adaptation. Ils peuvent, par exemple, être migrés si les ressources de l'hôte qui les accueillent se révèlent insuffisantes à un moment.

Le chapitre "Kali-Situation : Identifier quand avons-nous besoin d'adaptation ?" 12, présente comment la plateforme détecte le moment où il y a besoin d'adaptation. Les utilisateurs souhaitent pouvoir trouver sur leur périphérique les applications correspondant à leurs besoins du moment. Ce sont donc les activités quotidiennes des utilisateurs qui définissent ces besoins. Notre objectif est que la plateforme sélectionne des composants logiciels disponibles et les assemble en services constitutifs d'applications en réponse aux besoins des utilisateurs. Par ailleurs, la notion de domaine de l'utilisateur qui comprend la totalité des ressources auxquelles il peut accéder permet de faire en sorte que l'utilisateur dispose d'une machine virtuelle sur laquelle s'exécute une application qui se modifie en temps réel en fonction du contexte et de ses besoins. Nous interprétons ces moments d'adaptation à partir de situations sémantiques.

Nous avons conçu notre modèle de situation d'adaptation (KaliMOSA) autour d'une ontologie qui partage la structure de connaissances de contexte de KaliCOnto. KaliMOSA est intégré sans le DOCK ce qui lui permet d'accéder aux connaissances de contexte de KaliCOnto pour classifier les situations. Ces situations sont également détectées par des chaînes de raisonnement pour générer des notifications de situation au service de prise de décision.

L'architecture de la prise de décisions d'adaptation est présentée dans le chapitre "Plateforme de l'adaptation (Kali-Adapt)" 13. Notre solution, basée sur des situations sémantiques, traite le contexte selon une vision de haut niveau. La prise de décisions d'adaptation consiste à trouver une nouvelle architecture de l'application applicable dans le contexte actuel pour répondre à la situation d'adaptation identifiée. Cette architecture permet d'avoir une faible dépendance entre la construction des solutions et la décision logique. Elle permet également le choix des solutions et l'exécution des actions en fonction du contexte actuel. Lorsqu'une situation d'adaptation est identifiée, le service de prise de décision trouve dans la base de connaissances du contexte une "ASituation" d'alarme. Il utilise les ontologies de situations et de solutions pour trouver une solution à la notification reçue. L'ontologie de situation contient les informations de situation incluant les "ASituation" tandis que l'ontologie de solutions contient des connaissances de solutions en rapport à ces situations et à leurs causes. Lorsqu'une solution a été choisie, des actionneurs sont déployés pour réaliser l'adaptation conformément à la logique de traitement spécifique à cette situation. Le service de prise de décision travaille dans le domaine d'adaptation de l'utilisateur dans lequel il est unique. Notre proposition présente les avantages suivants : nombre de déclanchements réduit, Possibilité de prendre une décision rapide, Extensibilité des situations sémantiques faciles à comprendre et à interpréter et Situations sémantiques cohérentes.

14.2 PERSPECTIVES

Dans cette partie nous présentons les perspectives à moyen-terme et à long-terme que nous envisageons pour améliorer la reconfiguration contextuelle.

14.2.1 *Moyen-terme*

14.2.1.1 *Apprentissage*

Dans un environnement mobile et dynamique, la prise de décision ne peut pas garantir de choisir la meilleure solution pour une situation d'adaptation donnée durant l'exécution tout en gardant une bonne réactivité temporelle aussi nous sommes nous contentés de choisir la première solution applicable.

Dans l'optique de choisir une meilleure solution nous envisageons la mise en place d'un service passif chargé d'analyser les prises de décision passées (historique) sur le long-terme sans intervenir directement sur le cycle de vie de l'adaptation. Ce service pourrait alors affiner la prise de décision en ajoutant des situations d'adaptation dans l'ontologie de situations et en leur associant des solutions spécifiques dans l'ontologie de solutions. Ces nouvelles informations étant prises en compte lors des décisions futures elles pourraient permettre de faire évoluer le système de prise de décision par apprentissage. Bien entendu, l'absence de ce service n'obèrerait pas le fonctionnement de la plateforme.

Un tel service devrait s'appuyer sur deux informations de base :

— Un historique de contexte

— Un historique de prises de décision

La mise en place de ces historiques soulève toutefois le problème de la grande quantité de données qu'ils sont susceptibles de contenir puisque ces informations proviennent de tous les hôtes du domaine adaptation. Se pose également le problème du lieu de stockage de ces données, de leur accessibilité et de leur actualisation (quels critères utiliser pour décider quelles données sont devenues inutiles).

14.2.1.2 *Notion de domaine plus évoluée*

L'une des limitations de notre solution actuelle est que la plateforme limite son action au domaine d'un utilisateur. Ce domaine ne contient que les ressources des machines de cet utilisateur et les ressources publiques. Nous pourrions étendre cette notion de domaine d'adaptation en lien avec les travaux sur les communautés spontanées menés dans l'équipe ([19]). Ceci conduirait à inclure au domaine d'un utilisateur l'ensemble des ressources partagées par les membres des communautés auxquelles il appartient. Toutefois une telle approche soulève de nombreux problèmes. En effet, dès lors que le domaine d'adaptation inclut plusieurs utilisateurs, la reconfiguration d'une application décidée pour améliorer le service d'un utilisateur peut avoir comme conséquence de dégrader celui d'un autre utilisateur. Il n'est alors plus possible de raisonner au niveau d'un seul utilisateur comme nous le faisons jusqu'à présent mais il devient nécessaire de le faire au niveau d'un groupe d'utilisateurs.

Nous pouvons conserver l'architecture actuelle de Kalimucho-A c'est à dire conserver un seul service de prise de décision par domaine d'utilisateur mais nous devrons lui adjoindre un service d'harmonisation associé au groupe. Ainsi, si le fonctionnement dans chaque domaine d'adaptation d'utilisateur reste autonome il serait sous contrôle d'un service lié aux communautés auxquelles appartient cet utilisateur. Bien entendu, le rôle de ce service ainsi que les informations doit il devrait disposer restent à définir.

14.2.2 *Long-terme*

Notre perspective à long terme est d'offrir un support à la réalisation d'applications longue durée c'est à dire d'applications qui suivraient l'utilisateur dans son quotidien et s'adapteraient tant à ses besoins du moment qu'à sa localisation et qu'aux ressources dont il dispose. Il n'y aurait alors plus de lancement manuel d'applications comme c'est actuellement l'usage. La plateforme fournirait automatiquement les services correspondant aux activités actuelles de l'utilisateur au bon moment et au bon endroit. Ceci change profondément la notion d'application il ne s'agit plus de définir une application comme un logiciel qui sert à cela mais comme le logiciel qui sert à cet utilisateur. Un tel logiciel devient alors une description des besoins en lien avec les différentes situations auxquelles peut être confronté l'utilisateur et un ensemble d'activités associées. Autrement dit, une application est une connaissance des activités d'un utilisateur. Les services utilisés seront automatiquement et dynamiquement déployés en cours d'exécution et supprimés dès qu'ils ne sont plus utilisés. Le choix même des services serait fait automatiquement par la plateforme. Si Kalimucho-A offre une solution technologique, il reste de nombreux verrous à lever entre autres liés aux aspects de génie logiciel (comment concevoir de telles applications), à la sécurité et à l'acceptation par les usagers de telles solutions.

Quatrième partie

ANNEXES

A

REALISATION D'APPLICATIONS POUR KALIMUCHO

Ce chapitre est consacré au développement d'applications pour la plateforme Kalimucho-A. Nous distinguerons 3 types d'applications : applications normales, sensibles au contexte et sensibles aux situations. Une application normale est une application non sensible au contexte, comme les que l'on utilise communément. La logique métier ne dépend pas du tout des informations contextuelles comme par exemple un traitement de texte. Les deux autres types d'applications sont sensibles au contexte comme par exemple les applications de météo qui tiennent compte de la localisation de l'utilisateur. La différence entre applications sensibles au contexte et applications sensibles aux situations porte sur le niveau d'abstraction du contexte. Par applications sensibles au contexte, nous désignons celles qui ne s'intéressent qu'aux données brutes de contexte (cf. page.24) ou à des données de plus haut niveau sémantique directement extraites de ces données brutes. Par applications sensibles aux situations, nous entendons des applications qui s'intéressent à des situations constituées par des états de contexte liés entre eux par des relations logiques.

De nos jours, les applications contextuelles tendent à se développer fortement. Grâce aux nouvelles générations de smartphones, il est possible de capturer les informations contextuelles de l'utilisateur et l'on voit apparaitre sur le marché des applications prenant en compte ce contexte. Mais il s'agit là de la première génération d'applications contextuelles qui ne tiennent compte que des informations brutes directement récupérées par les capteurs de la machine où l'application s'exécute. Celles que nous appelons Application Contextuelle pour Kalimucho (ACK) sont celles qui ne s'intéressent pas seulement aux informations de bas niveau de contexte (spatio-temporel) mais aussi à des informations de contexte de plus haut niveau sémantique. Nous appelons Application Situationnelle pour Kalimucho (ASK) celles qui sont sensibles à des situations spécifiées par les applications elles-mêmes. La plateforme change leurs fonctionnalités, leurs compositions, leurs lieux d'exécution (hôtes) et leur qualité de service en fonction de situations identifiées. Le concepteur a en charge de donner au moins une reconfiguration pour chaque situation spécifiée par l'application.

Dans ce chapitre, nous présenterons d'abord comment programmer des applications normales pour Kalimucho : comment construire des services pour Kalimucho, comment utiliser le moteur de BPMN, etc. puis nous présenterons les étapes de construction d'une application sensible au contexte et, pour terminer, comment définir des situations.

A.1 KALISERVICE FRAMEWORK

A.1.1 *Objectifs*

L'objectif est de créer une API pour simuler un fonctionnement de type services à l'aide de composants. Le moyen de communiquer entre deux composants est par envoi d'informations entre eux au travers de connecteurs. Les composants agissent comme des pairs et non comme des services. Lorsqu'il est nécessaire que certains composants soient des fournisseurs de service, la logique d'implémentation de ces composants et de ceux qui se comporteront comme leurs clients est complexe, c'est la raison pour laquelle cette API a été développée.

A.1.2 *Principes*

Cette API est composée de trois classes.

A.1.2.1 *KaliMethodCall*

Cette classe inclut toutes les informations nécessaires à un composant client pour envoyer une requête de service à un composant fournissant un service. Elle contient le nom de la méthode à exécuter et la liste de ses paramètres. Si la méthode ne retourne pas de valeur, c'est la seule classe dont le client a besoin pour faire une requête de service.

A.1.2.2 *KaliMethodResponse*

Cette classe correspond à la réponse à un appel de méthode ou à une demande de service. Si la méthode ne retourne aucune valeur, cette classe n'est pas utilisée. Une seule instance de cette classe est associée à une instance de la classe KaliMethodCall, c'est pourquoi il n'est pas possible de directement l'instancier. Pour créer une nouvelle instance de cette classe, la méthode createResponse de la classe KaliMethodCall doit être appelée afin que les deux instances soient automatiquement liées. La seule chose que le composant fournissant le service doit faire est de remplir l'objet réponse lorsque la méthode a été exécutée à l'aide de cette réponse (ObjectResponse) et de renvoyer l'objet KaliMethodResponse au client.

A.1.2.3 *Token*

Cette dernière classe permet de savoir à quelle requête antérieure est associée une réponse reçue. Comme les communications entre composants sont asynchrones, la réponse à un appel de méthode est retardée et le composant doit pouvoir savoir récupérer la bonne réponse du service. Ceci est obtenu en mémorisant le Token associé à un objet de classe KaliMethodCall grâce à la méthode getToken de celui-ci. Ces Token peuvent être stockés par le composant dans une collection de sorte que lorsque la réponse arrive il puisse savoir si c'est la réponse attendue en utilisant la méthode belongsTo(Token) de l'objet KaliMethodResponse.

A.1.3 *Exemple*

L'exemple suivant montre une utilisation simple de cette API constituée par un composant fournisseur de service et un composant client.

Un utilisateur prend une photo et souhaite faire une reconnaissance de visage pour obtenir la liste des personnes présentes sur la photo.

Comme la reconnaissance de visages est une opération lourde d'autant plus que dans le cas présent la batterie du périphérique est basse, il décide de déléguer cette opération à un composant de service. Si le composant de service a une méthode faceRecognition(Image img) qui renvoie un objet de classe List<Person>, le code utilisé par les deux composants est le suivant :

Note : la prise en compte des exceptions et des erreurs a été omise pour simplifier le code

A.1.3.1 *Composant de service*

```
addInputListener(0, new InputListener() {
        public void performSample(Sample sample) throws StopBCException,
                InterruptedException {
                        KaliMethodCall request = (KaliMethodCall) sample;
                        Map<String, Object> params = request.getParams();
                        if (request.getMethodName().equals("faceRecognition")) {
                                Image img = (Image) params.get("image");
                                KaliMethodResponse response = request.createResponse();
                                response.setResponse(faceRecognition(img));
                                writeSample(0, response);
                        } else {
                                if (request.getMethodName().equals( ...))
                                        ...
                        }
                }
});
```

A.1.3.2 *Composant client*

```
Image img = takePicture();
Map<String, Object> params = new HashMap<String, Object>();
params.put("image", img);
KaliMethodCall request = new KaliMethodCall(getName(),
        "faceRecognition", params);
final Token token = request.getToken();
writeSample(0, request);

//Wait for the response
addInputListener(0, new InputListener() {
```

```
        public void performSample(Sample sample) throws StopBCException,
        InterruptedException {
                KaliMethodResponse response = (KaliMethodResponse) sample;
                if (response.belongsTo(token)) {
                        List<Person> personList = (List<Person>) response.getResponse();
                }
        }
});
```

Bien que le code pour qu'un composant se comporte comme un service ait été réduit considérablement, il augmente avec le nombre de méthodes que le service fournit. Ainsi, si le service fournit 20 méthodes différences, le code devient trop gros. Il est possible de réduire ce code en utilisant le mécanisme d'introspection de Java.

A.1.4 *Introspection*

Grace à l'introspection, le code Java lui-même peut avoir accès aux informations des classes Java comme les méthodes, les propriétés, les paramètres, etc. Il peut également exécuter une méthode d'une classe. Dans l'exemple précédent, pour choisir la bonne méthode à exécuter, le composant doit vérifier le nom de la méthode dans l'instance du *KaliMethodCall* puis appeler cette méthode. En utilisant l'introspection, la seule chose que le composant doive faire est de transmettre l'instance de *KaliMethodCall* à la plateforme qui se chargera de l'exécution de la méthode et en retournera la valeur. Dans ce but, une nouvelle méthode a été inclue dans l'API de Kalimucho (*public Object executeMethod (KaliMethodCall)*). Grâce à cela, le code du composant de service ne dépend plus du nombre de méthodes qu'il fournit.

Note : la prise en compte des exceptions et des erreurs a été omise pour simplifier le code

```
addInputListener(0, new InputListener() {
        public void performSample(Sample sample) throws StopBCException,
                InterruptedException {
                        KaliMethodCall request = (KaliMethodCall) sample;
                        KaliMethodResponse response = request.createResponse();
                        Object result = executeMethod(request);
                        response.setResponse(result);
                        writeSample(0, response);
                }
});
```

Non seulement ceci diminue de façon importante la taille du code mais ça le rend indépendant de l'application. C'est pourquoi ce mécanisme a été inclus dans une classe appelée *BCService* qui hérite de *BCModel*.

A.1.5 *Le modèle de composant BCService*

La classe *BCService* est le modèle pour les composants ayant un rôle de service, c'est-à-dire, dont le seul objectif est de recevoir des appels de méthodes sur leurs entrées et de renvoyer le résultat de ces méthodes sur leurs sorties. Le code nécessaire au traitement du *KaliMethodCall* et à la construction du *KaliMethodResponse* ou du *KaliMethodException* ainsi que le code pour lire les requêtes en entrée et écrire les résultats en sortie est encapsulé dans la classe *BCService*. Le concepteur d'un composant de service n'a plus qu'à écrire les méthodes que les services fournissent.

Si le service nécessite une initialisation ou une configuration, la classe BCService propose les méthodes abstraites *initService* et execute. La méthode *initService* est exécutée lors du déploiement du composant et une fois seulement (comme la méthode init de *BCModel*). La méthode execute sera exécutée après *initService* et chaque fois que le composant sera migré (comparable à la méthode *runBC* de *BCModel*). Il est à noter que comme *BCService* utilise la méthode *runBC* de *BCModel* pour lire et écrire dans ses connecteurs d'entrée et de sortie, la méthode execute est exécutée dans un Thread séparé.

A.1.5.1 *Example*

L'exemple qui suit, présente un composant de service qui offre 2 méthodes. Il est à noter que le seul code nécessaire est celui de ces 2 méthodes dans la mesure où ce service n'a pas besoin d'initialisation. Le code du client est le même que précédemment.

```
public class MyMathService extends BCService{
        @Override
        protected void initService() throws StopBCException,
        InterruptedException {


        }

        @Override
        protected void execute() throws StopBCException, InterruptedException{


        }

        public double fibonacci(int index) {
                if (index<0) return 0;
                if (index == 0 || index == 1)
                        return index;
                else
                        return fibonacci(index - 1) + fibonacci(index - 2);
        }

        public double factorial(int n) {
                if (n<2)
                        return 1;
```

```
                else return n*factorial(n-1);
        }

        @Override
        public float levelStateQoS() {
                return 1.0f;
        }
}
```

A.2 LE FRAMEWORK KALIBPMN

Avec KaliBPMN la plateforme Kalimucho est maintenant munie d'un moteur d'exécution BPMN qui permet aux composants d'envoyer des modèles BPMN exprimés en XML qui seront exécutés par ce moteur.

Pour pouvoir utiliser cette fonctionnalité, une plateforme dotée du moteur BPMN doit lancer un composant BPMNHost qui est fourni et qui permet aux composants distants de communiquer avec le moteur. Quand une plateforme DOCK est déployée, elle installe automatiquement son moteur BPMN.

Le moteur utilisé est le moteur Activiti qui offre la possibilité d'utiliser du code Java pour implémenter des parties de processus de traitement. Comme l'exécution du modèle implique dans ce cas 2 périphériques (client et hôte), KaliBPMN offre la possibilité d'exécuter ces classes Java sur le client ou sur l'hôte.

Les composants clients BPMN héritent de la classe abstraite BPMNClient, il s'agit d'une classe qui hérite de BCModel et propose des méthodes pour faciliter l'envoi de messages à l'hôte. Ces méthodes sont les suivantes :

```
public int deployModel(String modelName, String model)
public int createProcessInstance(String processId)
public int addEngineExecutionService(int processInstanceId,
String activityId, String className, String sourceCode)
public int addLocalExecutionService(int processInstanceId,
String activityId, String methodName)
public int startProcess(int processInstanceId,
Map<String, Object> processVariables)
```

Remarque : La classe BPMNClient propose une méthode execute dans laquelle le composant implémente sa logique métier. Cette méthode est exécutée au démarrage dans un Thread séparé.

Pour créer et démarrer un processus, le client doit suivre les étapes suivantes :

— Déployer le modèle BPMN dans l'hôte grâce à la méthode deployModel

— Créer une instance du processus qu'il souhaite démarrer grâce à la méthode createProcessInstance.

— Identifier les exécutions nécessaires

— Démarrer l'instance de processus grâce à la méthode startProcess

A.2.1 *Exécution de code Java dans le modèle*

Comme vu précédemment, le code Java peut être exécutée sur l'hôte et le client bien que le processus ne soit pas le même. La façon de faire référence à ces classes dans le modèle XML est d'inclure une balise <serviceTask> contenant le activitiId, un nom et une référence à une classe genericService qui sera toujours la même.

```
<serviceTask id="servicetask1" name="My first Java service" activiti:class="services.GenericService"/>
```

A noter que la seule chose qui change entre différents serviceTask est l'identifiant et le nom tandis que le champs activiti :class sera toujours le même (puisque nous ne pouvons pas faire référence à la classe Java du client car elle n'est pas visible depuis l'hôte et cela produirait un ClassNotFoundException.

A.2.1.1 *Code Java exécuté sur l'hôte*

Pour exécuter du code Java sur l'hôte, la classe doit implémenter l'interface JavaDelegate. Cette interface offre une méthode execute qui sera appelée quand le processus BPMN atteindra la tâche correspondante. La méthode execute a un seul paramètre qui est de classe DelegateExecution et inclut toute l'information sur le processus en cours d'exécution ainsi que les variables du processus. Pour plus d'informations, se reporter à la documentation d'Activiti [1].

Le moyen pour ajouter ce code Java au modèle est d'invoquer la méthode addEngineExecutionService.

A.2.1.2 *Code Java exécuté sur le client*

Dans la mesure où le client n'a pas de contact direct avec le moteur BPMN il est impossible de lui envoyer une instance de delegateExecution. Ainsi, pour offrir la possibilité d'interagir avec les variables et l'état du processus, KaliBPMN propose une version limitée de delegateExecution appelée kaliDelegateExecution. Cette classe inclut outre des informations sur le processus et la tâche, les variables du processus dans une collection (Map). Le client peut consulter, modifier et ajouter des variables dans cette collection et la renvoyer à l'hôte qui appliquera les changements correspondants au moteur.

Pour exécuter une classe Java dans le client, il faut :

— Créer une méthode dans le composant Client qui sera appelée par l'hôte quand le processus BPMN atteint la tâche représentée par activitiId. Cette méthode doit avoir un paramètre de classe KaliDelegateExecution et renvoyer un objet de classe KaliDelegateExecution.

— Appeler la méthode addLocalExecutionService en lui passant l'identifiant du processus, l'identifiant de la tâche (défini dans le modèle XML) et le nom de la méthode définie à l'étape 1.

— Après avoir exécuté les étapes 1 & 2 pour chacune des méthodes à rajouter, on peut démarrer une instance de processus par la méthode startProcess.

Le moteur BPMN prend en charge la logique BPMN et appelle ces méthodes au besoin.

1. http ://www.acvitivi.org/userguide

A.2.2 *Exemple*

Nous prendrons l'exemple pour les 2 sections suivantes d'un client qui souhaiterait connaître le nème nombre de Fibonacci. Comme cette opération peut être longue, elle sera effectuée par le moteur de BPMN et donc sur un autre distant et plus puissant de type PC. A la fin du calcul, le client affichera le résultat à l'écran. Ainsi, le modèle BPMN a une exécution coté moteur (calcul de Fibonacci) et une exécution coté client (l'affichage du résultat).

A.2.2.1 *Exemple coté client*

```
public class MyKaliBPMNClient extends BPMNClient {
        public KaliDelegateExecution printResults (KaliDelegateExecution execution) {
                Console.println("[Client]Fibonacci of "
                + execution.getVariable("index") + " = "
                + execution.getVariable("result"));
                return execution;
        }

        @Override
        protected void execute() throws StopBCException, InterruptedException
        {
                InputStream bpmnXml =
        getResourceAsStream("application/resources/fibonacci.bpmn20.xml");
                try {
                        deployModel("fibonacci", IOUtils.toString(bpmnXml));
                        int processInstanceId =
                                createProcessInstance("fibonacci");
                        InputStream sourceCode =                getResourceAsStream("application/
                            resources/CalculateFibonacci.txt");
                        addEngineExecutionService(processInstanceId,
                                "servicetask1", "CalculateFibonacci",
                                IOUtils.toString(sourceCode));
                        addLocalExecutionService(processInstanceId,"servicetask2",
                                "printResults");
                        Map<String, Object> variables = new HashMap<String,
                                Object>();
                        int index = 45;
                        variables.put("index", index);
                        startProcess(processInstanceId, variables);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

A.2.2.2 *Modèle BPMN*

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
targetNamespace="http://www.bpmnwithactiviti.org"
xmlns:activiti="http://activiti.org/bpmn">

<process id="fibonacci" name="Calculate fibonacci">
<startEvent id="startevent1" name="Start" />

<sequenceFlow id="sequenceflow1" name="Sending work to the Host"
sourceRef="startevent1" targetRef="servicetask1" />

<serviceTask id="servicetask1" name="Calculate fibonacci"
activiti:class="services.GenericService" />

<sequenceFlow id="sequenceflow2" name="Sending response to Client" sourceRef="servicetask1" targetRef=
    "servicetask2" />

<serviceTask id="servicetask2" name="Print results"
activiti:class="services.GenericService" />

<sequenceFlow id="sequenceflow3" name="Ending the process"
sourceRef="servicetask2" targetRef="endevent1" />

<endEvent id="endevent1" name="End" />
</process>

</definitions>
```

A.2.2.3 *Exemple du service coté serveur*

```
import org.activiti.engine.delegate.DelegateExecution;
import org.activiti.engine.delegate.JavaDelegate;
import java.security.InvalidParameterException;
import platform.Console;

public class CalculateFibonacci implements JavaDelegate{

@Override
public void execute(DelegateExecution execution) throws Exception {
```

```
Integer index = (Integer) execution.getVariable("index");
if (index==null){
execution.setVariable("result", 0d);
}else{
Console.println("Calculating fibonacci of " + index + "...");
Double result = fibonacci(index);
Console.println("Calculations done");
execution.setVariable("result", result);
}
}

private double fibonacci(int i) {
if ((i == 0) || (i == 1))
return i;
else
return fibonacci(i - 1) + fibonacci(i - 2);
}
}
```

A.3 L'ÉDITEUR GRAPHIQUE D'ACTIVITI

Ecrire à la main un composant client BPMN (à la fois le code et le fichier BPMN) est long et suppose l'implémentation de code n'ayant pas de relation avec le modèle BPMN lui-même (non fonctionnel). C'est pourquoi un éditeur graphique qui crée un composant client BPMN incluant le code nécessaire au démarrage du processus est proposé. Ainsi, le développeur a seulement besoin d'implémenter le code des tâches définies dans le modèle BPMN. Cet éditeur graphique a été construit sur un éditeur graphique BPMN créé par les créateurs d'Activiti [2]. A l'origine, cet éditeur graphique ne permettait de produire qu'un fichier BPMN. La version spécifique à Kalimucho-A de l'éditeur que nous avons réalisée produit un fichier compressé contenant un dossier source de composant Kalimucho. Ce dossier contient un fichier BPMN, la classe du composant BPMNClient et toutes les classes des tâches définies. Le développeur peut alors importer ce dossier dans son IDE et terminer l'implémentation du code métier (fonctionnel).

Dans la mesure où cet éditeur graphique est basé sur celui de Activiti, la plupart de ses fonctionnalités peuvent être trouvées dans la documentation de Activiti [3]. Pour illustrer les modifications apportées à cet éditeur, nous allons construire pas à pas un exemple de modèle BPMN.

Notre exemple capture une photo à partir de l'appareil photo, et réalise la détection et reconnaissance de visage, puis sauvegarde le résultat dans un fichier. Parmi ces 4 tâches, la plus complexe est celle de la reconnaissance de visage. Elle sera réalisée par l'hôte sur lequel se trouve le moteur. Comme la détection de visage et la reconnaissance de visage peuvent être faites simultanément, la détection sera faire en parallèle sur le périphérique du client. De même, la prise de photo et la sauvegarde des résultats doivent avoir accès à des ressources locales et seront exécutés sur le périphérique du client.

2. http ://www.activiti.org/components.html
3. http ://www.acvitivi.org/userguide

La première étape est la réalisation de notre modèle BPMN compatible Kalimucho. Ceci est réalisé en sélectionnant la propriété de modèle IsforKalimucho comme on le voit sur la figure 113.

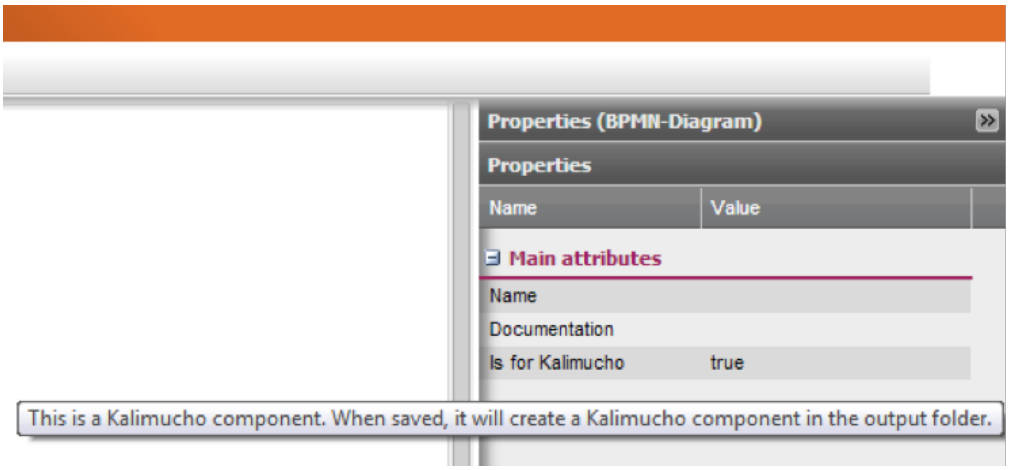

FIGURE 113 – Propriété du modèle IsforKalimucho

L'étape suivante consiste à définir les 4 tâches. Le flux d'exécution de notre modèle est montré à la figure 114.

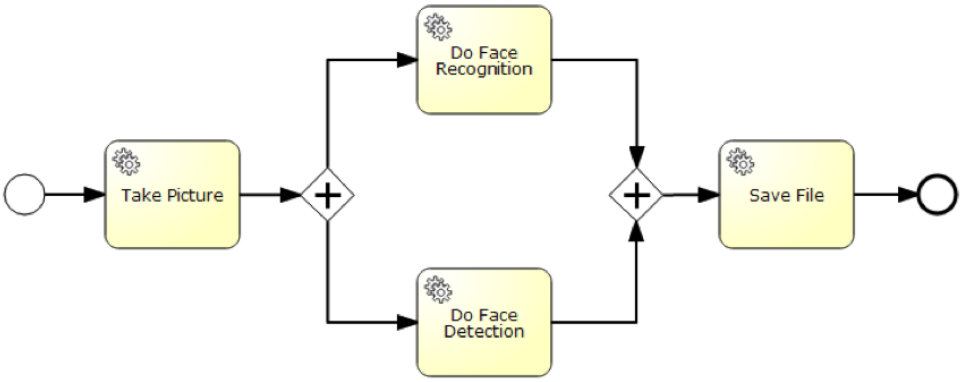

FIGURE 114 – Diagramme BPMN

Nous devons maintenant modifier les propriétés de chacune des tâches. Les propriétés importantes pour nous sont le type de tâche, la classe Java, et l'exécution (hôte d'exécution). Le type doit être "service", la classe Java sera celle du composant Kalimucho. La propriété exécution indique si la tâche doit être exécutée sur l'hôte supportant le moteur (Engine) ou sur l'hôte local (Local).

La figure 116 montre les propriétés de la tâche de reconnaissance de visage.

A.4 COMPOSANTS PARAMÉTRÉS

Lorsque le modèle BPMN a été enregistré, le fichier compressé contient l'arborescence (présentée à la figure.115).

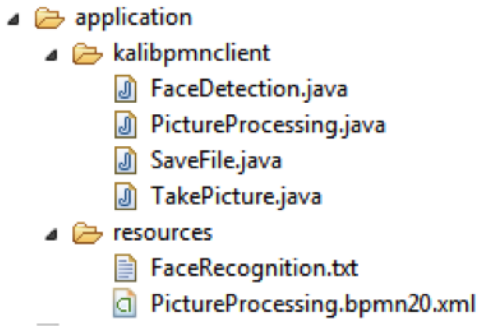

FIGURE 115 – L'arborescence des fichiers

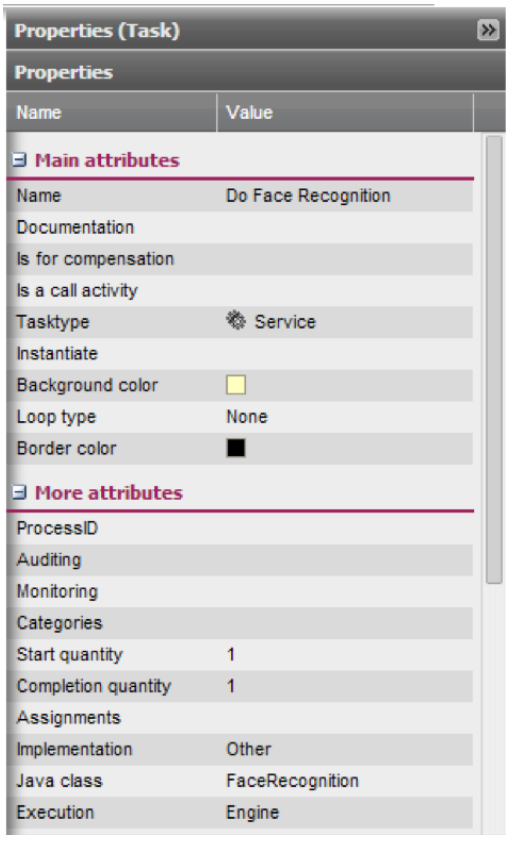

FIGURE 116 – Task properties

Le développeur doit alors implémenter le code des 4 tâches. Trois d'entre-elles s'exécutent sur le client et sont donc des fichiers Java. La tâche de reconnaissance de visage est exécutée sur l'hôte supportant le moteur (Engine) et est représenté par un fichier texte dans le répertoire de ressources (Resources). Le fichier PictureProcessing.java contient un composant Kalimucho qui hérite de BPMNClient. Elle inclut le code nécessaire au déploiement du modèle et des tâches du moteur BPMN et à son exécution.

Un composant Kalimucho est a priori défini pour être déployé indépendamment et s'initialiser lui-même. Mais ceci n'est pas suffisant pour la composition de composants comme un service Kalimucho-A. Dans le cas de composition de composants, un composant peut avoir des paramètres d'exécution qui ne peuvent pas être extraits de l'environnement du composant et doivent lui être fournis par un autre composant avant son déploiement. C'est pourquoi Kalimucho-A propose un mécanisme permettant le déploiement de composants paramétrés. Si un composant doit en déployer un autre, il peut utiliser la classe Component. Cette classe définit un composant Kalimucho et des méthodes pour le déployer et le supprimer. Outre les propriétés normales nécessaires à la définition d'un composant Kalimucho, elle permet aux développeurs d'inclure des objets paramétrés qui seront encapsulés et joints aux composants par la méthode addParameter(String Object).

Le composant peut alors retrouver ses objets dans sa méthode d'initialisation. Pour récupérer un paramètre, il utilise la méthode statique : ParameterManager.getParameter (BCModel, String) qui retourne un Object contenant la valeur du paramètre. Le paramètre BCModel de cette méthode sert à identifier le composant et à récupérer le paramètre identifié par le nom donné dans le second paramètre de la méthode.

A.4.1 *Exemple*

A.4.1.1 *Positionner un paramètre*

```
Component myComponent = new Component();
...
myComponent.addParameter("situationID", getSituationID());
...
myComponent.deploy(this);
```

A.4.1.2 *Récupération d'un paramètre*

```
public void init() {
        String situationID = (String)ParameterManager.getProperty(this,
        "situationID");
        ...
}
```

A.5 ACCÈS AU SERVICE KALIMUCHO-A

A.5.1 *Comment réaliser une recherche dans le contexte*

Le service de recherche dans le contexte est situé dans le Dock. C'est un composant Kalimucho appelé ContextSearchEngine qui est automatiquement déployé au démarrage du Dock. Un composant de l'application peut se connecter au composant ContextSearchEngine en utilisant la méthode connectToContectSearchEngine de la classe DockManager.

Lorsque le composant est connecté au ContextSearchEngine, le DockManager peut appeler les méthodes distantes executeStaticContextQuery ou executeDynamicContextQuery pour accéder au contexte.

Chaque fois qu'une requête est envoyée, il cherche l'origine du contexte, créé un ContextCollector associé à cette origine et déploie les adaptateurs nécessaires. En fonction du type de requête (statique ou dynamique), la valeur de retour de la méthode sera différente.

A.5.1.1 *Recherche statique de contexte*

Pour une recherche statique de contexte, les paramètres requis sont :

— Le nom du ContextScope qui indique dans quelle partie du contexte rechercher, par exemple "Battery". Une liste de noms de ContextScope est fournie en constante dans la classe ContextScope. Par exemple ContextScope.BATTERY.

— La localisation qui spécifie d'où doit venir la donnée de contexte. Ce paramètre peut être soit un nom d'hôte, soit un nom de lieu stocké dans le modèle de contexte.

— La représentation est une chaîne de caractères qui indique sous quelle forme doivent être représentées les données de contexte. Une liste des représentations possibles est disponible en constante dans la classe Representation. Par exemple, Representation.STRING.

— Les filtres sont une collection (Map < ContextDataSignification, String>) contenant la liste des ContextDataSignification qui doivent être retournés dans l'unité associée. La liste des valeurs possibles est trouvée dans la classe ContextDataSignification. Par exemple ContextDataSignification.BATTERY_QUANTITY. Une liste des unités est défini en constante dans la classe Unit. Par exemple Unit.FAHRENHEIT.

Si les listes de représentation ou d'unité fournies par Kalimucho-A ne satisfont pas l'application, le développeur peut définir ces propres représentations et ses propres transformations d'unités et les ajouter à la plateforme. Ceci sera expliqué dans la section A.7 (Comment étendre Kalimucho-A).

Dans le cas d'une requête statique, la valeur de retour de la méthode sera une liste de ContextMetaDataItem représentant les données de contexte demandées selon la représentation et l'unité spécifiée.

A.5.1.2 *Recherche dynamique de contexte*

Pour une recherche dynamique de contexte aux paramètres déjà décrits dans la section précédente, vient s'ajouter : la condition de notification qui définit comment les notifications de contexte sont envoyées à l'application.

Lors d'une recherche dynamique, le moteur de recherche de contexte renverra l'identifiant du composant réalisant la transformation et le nom du périphérique sur lequel ce composant a été déployé.

L'application se connectera à ce composant pour recevoir la donnée de contexte. La figure 117 illustre l'architecture déployée à l'issue d'une recherche dynamique de contexte.

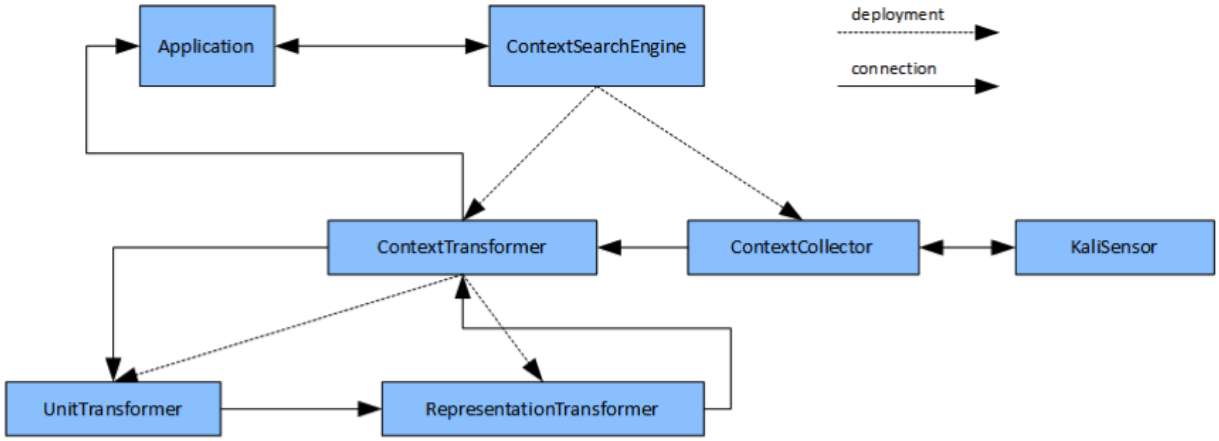

FIGURE 117 – Service de recherche de contexte

A.5.1.3 *Exemple de code d'une requête dynamique*

```
//Connect to the context searching service
DockManager manager = DockManager.getInstance();
manager.connectToQuerier(this, getMyIPAddress(), 0, 0);

//Build the query (a dynamic context search for the phone
//battery quantity whenever the battery is below 40%)
ContextScope batteryScope = ContextScope.BATTERY;
String contextLocation = getMyPhoneID();
Representation stringRepresentation = Representation.STRING;
ContextDataSignification quantitySigni = ContextDataSignification.BATTERY\_QUANTITY;
Unit noUnit = null;
Map<ContextDataSignification, Unit> filters = new HashMap<ContextDataSignification,Unit>();
filters.put(quantitySigni, noUnit);
NotificationCondition<Boolean> condition = new LessThanEqual<Integer>(new SensorValueElement<Integer>(
    quantitySigni), new Element<Integer>(40));

//Build the method call
List<Object> params = new ArrayList<Object>();
params.add(batteryScope);
params.add(contextLocation);
params.add(stringRepresentation);
params.add(filters);
params.add(condition);
```

```
KaliMethodCall call = new KaliMethodCall(getName(), "executeDynamicContextQuery", params);

//Send the query
writeSample(0, call);

//Wait for the response
setInputClassFilter(0, KaliMethodResponse.class.getName());
KaliMethodResponse response = (KaliMethodResponse) readSample(0);

RuntimeComponentInfo info = (RuntimeComponentInfo) response.getResponse();
String componentID = info.getComponentID();
String componentIP = info.getComponentIP();

//Connect this component to the received context transformer
//Refer to the Kalimucho documentation for more info about using the BCCommandSender
BCCommandSenderPlugin sender = (BCCommandSenderPlugin)waitForKalimuchoService(Services.APPLICATION\
    _COMMAND\_SERVICE);
String connectorName = "queryConnector";
NetworkAddress componentAddress = new NetworkAddress(componentIP);
sender.createExternalInputConnector(this, connectorName, componentAddress);
sender.duplicateOutputComponent(componentAddress, this, componentID, 0, connectorName);
sender.reconnectInputComponent(componentAddress, this, getName(), 1, connectorName);

//Create an input listener to receive data from the context transformer
addInputListener(1, new InputListener() {
        @Override
        public void performSample(Sample sample) throws StopBCException,
        InterruptedException {
                // TODO Treat the context data
        }
});
```

A.5.2 *Réception des notifications de situation*

Le middleware Kalimucho-A offre un service de notification des applications lorsqu'une situation spécifique a été identifiée dans le domaine. Les applications peuvent s'abonner à des situations qu'elles veulent connaître. L'accès à ce service se fait par la classe SituationManager. Cette classe propose des méthodes pour s'abonner et se désabonner à des situations. Les noms des situations existant dans le middleware Kalimucho-A sont stockés en constante dans la classe Situation par exemple : "Situation.LOW_BATTERY_SITUATION".

A.5.2.1 *Abonnement*

La méthode Subscribe de la classe SituationManager prend en paramètre une liste de noms de situations et le nom du composant qui s'inscrit (obtenu par la méthode getName de BCModel). Elle crée dans le Dock un composant qui surveille les modifications de l'ontologie de situations et notifie l'application dès que l'une des situations choisies est identifiée dans le domaine. De plus, elle connecte le composant qu'elle vient de créer à celui qui s'est inscrit. Par cette connexion, le composant d'applications recevra les notifications.

Les notifications de situation sont encapsulées dans un objet de classe SituationSample pour être conforme aux spécifications de Kalimucho. L'objet de classe SituationSample contient un objet de classe Situation qui peut être obtenu par la méthode getValue. La figure 118 montre une application utilisant le service de notification de situation.

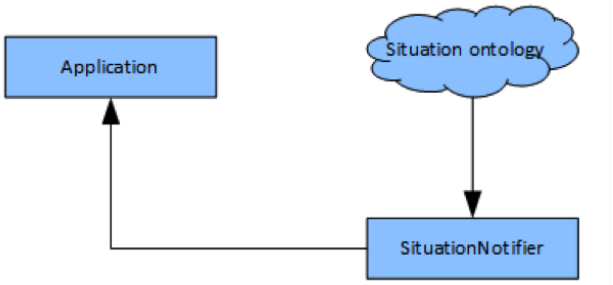

FIGURE 118 – Service de notification de situation

A.5.2.2 *Désabonnement*

La classe SituationManager propose une méthode unsubscribe qui prend en paramètre le RunTimeComponentInfo renvoyé par la méthode Subscribe et supprime le composant de notification de situation qui avait été créé par Subscribe.

A.6 COMMENT DÉVELOPPER DES APPLICATIONS UTILISANT KALIMUCHO-A

Développer une application fonctionnant sur le middleware Kalimucho-A nécessite quelques étapes de plus que développer de simples composants Kalimucho. Comme les applications sont constituées de plusieurs composants et peuvent être exécutées dans différentes configurations, le middleware doit connaître ces informations pour pouvoir déployer, redéployer, reconfigurer et stopper les applications.

A une application est associée une liste de configurations. Chaque configuration contient une liste de services. Un service contient une liste de configurations de ce service. Chaque configuration de ce service contient une liste de composants Kalimucho. Comme le développeur d'applications peut ne pas avoir d'expérience dans les ontologies, ces informations peuvent être écrites dans des fichiers de configuration et stockés dans le répertoire Apps de Kalimucho. La plateforme détectera la présence de ces fichiers et les enverra au Dock qui les stockera dans l'ontologie.

L'application ainsi que chacun des services doit avoir un fichier de configuration. La syntaxe XML de ces fichiers sera détaillée à la fin de cette partie.

A.6.1 *Qu'est-ce qu'un service Kalimucho-A*

Un service Kalimucho-A est une combinaison de composants Kalimucho-A. Du point de vue de l'utilisateur du service, il peut être vu comme un simple composant Kalimucho-A puisque sa structure interne est cachée et qu'il propose des ports d'entrées/sorties pour s'y connecter. Il revient au développeur de service d'en définir la structure interne et de relier les entrées et les sorties du service aux entrées et sorties de ses composants internes.

Comme indiqué précédemment, un service peut avoir plusieurs configurations. Une configuration représente un ensemble de composants Kalimucho-A et leurs interconnections. Les différentes configurations d'un service représentent le même service avec des différences de qualité.

Par exemple, un service de prise de photos peut proposer une configuration avec reconnaissance de visage et une configuration avec seulement une détection de visage en fonction des ressources disponibles à l'exécution. Dans les deux cas, le service renverra une image mais la richesse du résultat sera plus grande dans un cas que dans l'autre.

La figure 119 montre un exemple de service ayant 2 entrées et une sortie et offrant 2 configurations.

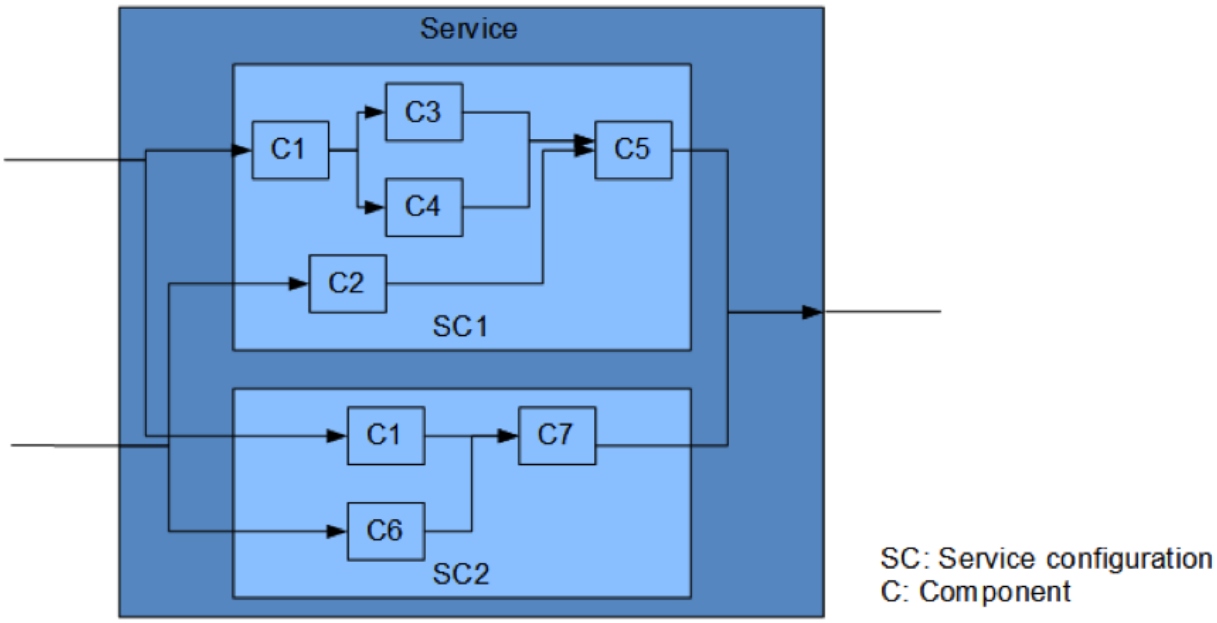

FIGURE 119 – Exemple de structure de service

A.6.2 *Fichiers de configurations de service*

A.6.2.1 *Composants*

Le nom de balise pour un composant est <Component>.

Informations nécessaires : Un élément 'Component' doit contenir les informations suivantes : un attribut 'id', donnant le nom du composant dans l'application et un attribut 'class' indiquant le nom complet de la classe du composant.

Informations facultatives : Un élément <Component> peut aussi avoir un attribut 'inputs' indiquant le nombre de ports d'entrée et un attribut 'outputs' indiquant le nombre de ports de sorties. La valeur par défaut des attributs s'ils ne sont pas précisés est 0. De plus, un élément <Component> peut inclure un élément <ComponentRequirements>. Cet élément contiendra une liste de <ComponentRequirement>. Enfin, il peut aussi inclure un élément <DesirableConditions>. Cet élément contient une liste de conditions souhaitées pour le composant. Ces conditions ne sont pas indispensables et un périphérique qui ne satisfait pas à ces conditions peut tout de même exécuter ce composant, mais on essaiera de préférence de déployer ce composant sur un périphérique qui satisfait les conditions.

A.6.2.2 *Exigences d'éxecution des composants*

Le nom de balise pour une exigence d'éxecution de composant est 'ComponentRequirement'. Elle représente une condition que le périphérique de destination doit remplir pour pouvoir y déployer le composant. Cette condition sera évaluée avant le déploiement d'un composant et lors d'une migration.

Informations nécessaires : L'élément <ComponentRequirement> doit au minimum inclure un attribut 'Name' indiquant le nom de l'attribut. Il peut appartenir à une liste prédéfinie d'exigences ou être défini par le développeur. Pour en savoir plus sur la façon de créer une exigence de composant, se référer à la partie "Comment étendre Kalimucho-A".

A.6.2.3 *Preférence d'exécution*

Un élément <DesirableCondition> doit seulement contenir un attribut 'Name' indiquant le nom de la condition et un ContextEntity avec une description de preférence, par exemple, ContextEntity : Screen, Description : Biggst.

A.6.2.4 *Connecteurs*

Le nom de balise pour un connecteur est 'Connector'.

Informations nécessaires : Un élément <Connector> doit au minimum inclure un attribut 'Name' indiquant le nom du connecteur, un attribut 'From' indiquant le nom du composant connecté à l'entrée de ce connecteur et un attribut 'To' indiquant le nom du composant connecté à la sortie de ce connecteur.

Informations facultatives : Un élément <Connector> peut aussi inclure un attribut 'From port' indiquant le numéro de la sortie du composant défini par 'From' et un attribut 'To Port' indiquant le nom de l'entrée du composant définie par 'To'. La valeur par défaut de ces propriétés si elles ne sont pas précisées est 0.

A.6.2.5 *Configuration de service*

Une configuration de service consiste en une liste de composants. De l'extérieur, le service apparait comme un seul composant avec ses entrées et ses sorties. Le développeur du service devra définir les liens entre les entrées (resp. sorties) du service et les entrées (resp. sorties) des composants à l'intérieur de ce service.

Informations nécessaires : Un élément <ServiceConfiguration> doit avoir au minimum un paramètre 'Name' indiquant le nom de la configuration, un attribut 'Inputs' indiquant le nombre de ports d'entrées du service et un attribut 'Outputs' indiquant le nombre de sorties du service. Il contient également la liste des composants et des connexions de la configuration. La liste des composants est donnée par <Components> et la liste des connexions par une balise <Connections>. Il est à noter que la liste des composants dans la balise <Components> est une liste de liens vers des descriptions de composants. Ces descriptions de composants sont stockés en dehors de la balise <ServiceConfiguration>, dans une liste différente afin que différentes configurations qui utiliseraient le même composants n'aient pas à les définir plusieurs fois ce composants. Un lien de composant est représenté par une balise <ComponentLink>. Elle ne contient que l'attribut 'Name' indiquant le nom du composant. Ce nom doit être le même que celui utilisé dans l'attribut 'Name' de la balise <Component>.

Une balise <ServiceConfiguration> inclut également les liens entre les entrées (resp. les sorties) du service et les entrées (resp. les sorties) des composants le constituant. Ceci se fait par les balises <InputLink> et <OutputLink>. Une balise <InputLink> a une propriété 'Number' indiquant le numéro de l'entrée du service, une propriété 'ToComponent' indiquant le nom du composant qui y est relié, et une propriété 'InputPort' indiquant le numéro de l'entrée de ce composant. Par exemple, pour relier l'entrée 0 d'un service à l'entrée 1 du composant photoTaking, le développeur utilisera la syntaxe suivante : <inputLink> Number = "0" ToComponent = "photoTaking" InputPort = "1" />. La balise <OutputLink> suit une syntaxe analogue dans laquelle 'Number' indique la sortie du service, 'FromComponent' indique le nom du composant lié et OutputPort le nom de sortie de cet élément.

Informations facultatives : Une balise <ServiceConfiguration> peut inclure un élément <ServiceRequirements> qui définit une liste d'exigences du service. Une balise <ServiceRequirement> inclut un attribut 'nom' indiquant le nom de l'exigence. Il peut appartenir à une liste prédéfinie d'exigences ou être défini par le développeur. Pour en savoir plus sur la façon de créer une exigence de composant, se référer à la partie "Comment étendre Kalimucho-A".

A.6.2.6 *Services*

Une balise <Service> consiste en une liste de balises de configurations du service.

Informations nécessaires : Une balise de <Service> doit contenir au minimum un attribut 'name' indiquant le nom du service. Un attribut 'Inputs' indiquant le nombre d'entrées du service, un attribut 'Outputs' indiquant le nombre de sorties du service, une balise <ServicesConfigurations> contenant une liste de configurations du service et une balise <ComponentsDefinitions> dans laquelle les composants utilisés dans les différentes configurations du service sont définis.

Les listes de définition de composants et de configurations de service doivent avoir au moins un élément.

A.6.2.7 *Exemple de fichiers de configuration*

```
<service name="PhotoTaking" inputs='1' outputs='1'>
        <componentDefinitions>
                <component name="PhotoTaking"
```

```
                className="application.kalicamera.PhotoTaking"
inputs='1' outputs='1'>
                        <componentRequirements>
                                <componentRequirement name="GUI"/>
                        </componentRequirements>
                </component>
                <component name="FaceDetection"
                className="application.kalicamera.FaceDetection"
inputs='1' outputs='1'>
                        <componentRequirements>
                                <componentRequirement name="Android"/>
                        </componentRequirements>
                </component>
        </componentDefinitions>
        <serviceConfiguration name="FaceDetectionPhotoTaking">
                <components>
                        <componentUsed name="PhotoTaking"/>
                        <componentUsed name="FaceDetection"/>
                </components>
                <connectors>
                        <connector name="PhotoTaking\_to\_FaceDetection"
from="PhotoTaking" to="FaceDetection"/>
                </connectors>
                <inputLink number='0' toComponent="PhotoTaking" inputPort='0'/>
                <outputLink number='0' fromComponent="FaceDetection"
outputPort='0'/>
                <serviceRequirements>
                        <serviceRequirement name="AvailableSmartphone"/>
                </serviceRequirements>
        </serviceConfiguration>
        <serviceConfiguration name="SimplePhotoTaking">
                <components>
                        <componentUsed name="PhotoTaking"/>
                </components>
                <inputLink number='0' toComponent="PhotoTaking" inputPort='0'/>
                <outputLink number='0' fromComponent="PhotoTaking"
outputPort='0'/>
        </serviceConfiguration>
</service>
```

A.6.3 *Qu'est qu'une application Kalimucho-A*

Une application Kalimucho-A peut être définiE comme un ensemble de services Kalimucho-A interconnectés et une interface graphique. Comme les services Kalimucho-A, une application Kalimucho-A peut avoir plusieurs configurations. La plateforme essaiera toujours de déployer la meilleure configuration possible. Une configuration d'application consiste en un ensemble spécifique de services interconnectés, une interface graphique et une liste de chaînes de raisonnement de contexte qui seront déployées avec les services de l'application.

Une application peut aussi contenir ses propres ontologies, ainsi, les chaînes de raisonnement peuvent produire des données de contexte de plus haut niveau qui n'étaient pas prévues dans l'ontologie de la plateforme.

La figure 120 montre la structure d'une application Kalimucho-A ayant 2 configurations, 2 services, 2 chaînes de raisonnement et une ontologie propre. Le service de nom S1 est utilisé dans les 2 configurations.

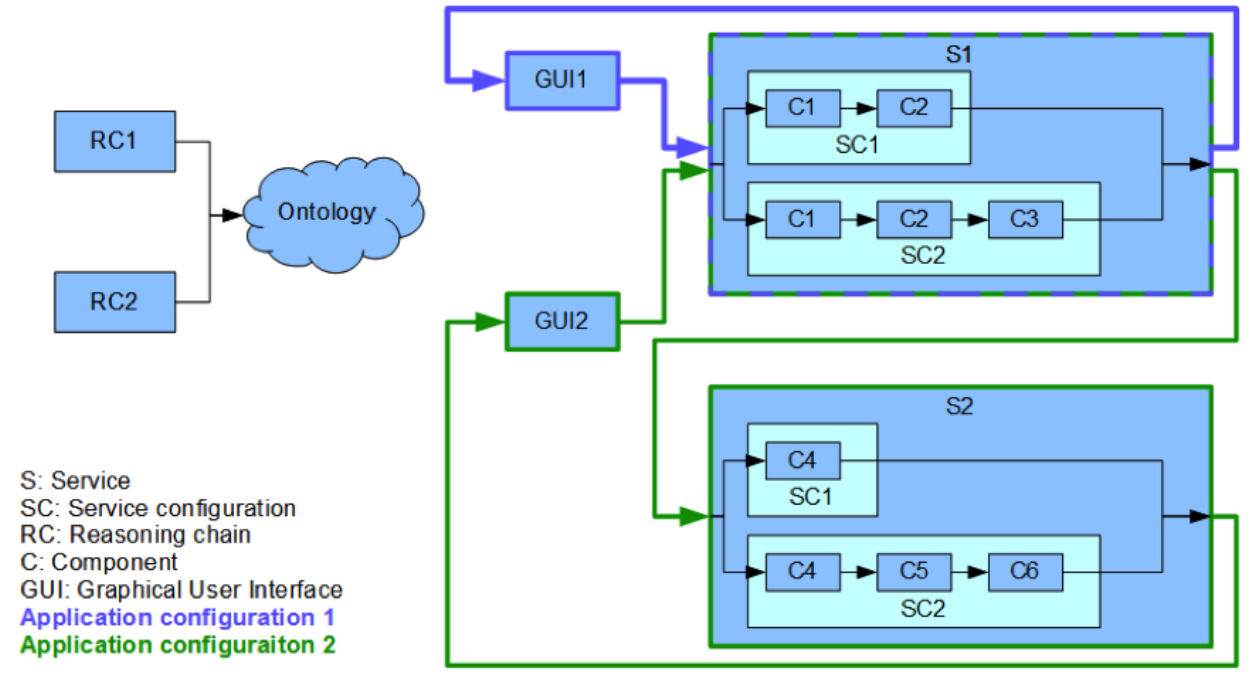

FIGURE 120 – Exemple de structure d'application

A.6.4 *Définition d'une application Kalimucho-A*

A.6.4.1 *Interface graphique*

Une interface graphique est un simple composant Kalimucho-A qui assure l'interface entre l'utilisateur et l'application. La façon de définir une interface est quasi identique à celle permettant de définir un composant dans un fichier de configurations de service.

Une balise <GUI> contient les informations suivantes : un attribut 'Name' indiquant le nom du composant gérant l'interface, et un attribue 'Class' indiquant le nom complet de la classe de ce composant. Elle

peut aussi avoir un attribut 'Inputs' et un attribut 'Outputs' indiquant le nombre d'entrées et de sorties. La valeur par défaut de ces propriétés est de 0 si elles ne sont pas définies.

A.6.4.2 *Services*

Une balise <Service> défini l'usage d'un service dans une application Kalimucho-A. Elle doit contenir un attribut 'Name' indiquant le nom du service.

A.6.4.3 *Connexion de services*

Un service peut être connecté à un autre service ou à l'interface graphique. Ces connexions sont représentées par la balise <Connection> qui a un attribut 'Name' indiquant le nom de la connexion, un attribut 'From', indiquant le nom du service ou l'interface qui envoie le nom des données et un attribut 'To' indiquant le nom du service ou de l'interface qui reçoit les données. Une balise <Connection> peut aussi avoir des attributs 'FromPort' et 'ToPort' indiquant quelle entrée (resp. sortie) du service ou de l'interface est reliée. La valeur par défaut de ces attributs est 0.

A.6.4.4 *Exigences de l'application*

Une exigence d'application définiT les conditions qui doivent être satisfaites avant de déployer une configuration de l'application. Les développeurs de l'application peuvent définir leurs propres exigences. Pour plus d'informations sur la définition des exigences, se référer à la section A.7 (Comment étendre Kalimucho-A).

Une balise <ApplicationRequirement> doit avoir un attribut 'Name' définissant le nom de l'exigence.

A.6.4.5 *Configuration d'applications*

Une configuration d'application consiste en une IHM, une liste d'exigences d'applications, une liste de services et une liste de connexions entre les services. La liste d'exigences d'applications sera placée dans une balise <ApplicationRequirements>, la liste des services sera placée dans une balise <ServiceList> et la liste des connexions sera placée dans une balise <Connexions>.

Une balise <ApplicationConfiguration> doit avoir un attribut 'Name' indiquant le nom de la configuration de l'application.

A.6.4.6 *Ontologie*

Une application peut utiliser sa propre ontologie pour raisonner sur le contexte. Cette ontologie est hébergée sur la plateforme du DOCK et sera déployée avec l'application. La durée de vie des ontologies d'applications peut être liée à l'application ou à la plateforme.

Une ontologie dont la durée de vie est liée à l'application sera supprimée quand l'application se termine. Une ontologie liée à la plateforme survit à la fermeture de l'application mais sera supprimée lors de l'arrêt de la plateforme.

La définition d'une ontologie est faite par une balise <Ontology>. Cette balise doit avoir un attribut 'Name' indiquant le nom de l'ontologie, un attribut 'Path' indiquant le chemin du fichier contenant l'onto-

logie relativement à celui du fichier de configuration et un attribut 'Lifetime' indiquant la durée de vie de l'ontologie. Cet attribut peut être "Application" ou "Plateforme".

A.6.4.7 *Chaîne de raisonnement*

Un élément chaîne de raisonnement d'une application doit avoir un attribut 'Name' indiquant le nom de la chaîne de raisonnement et un attribut 'BpmnPath' indiquant le chemin du fichier BPMB relativement au fichier de configuration.

A.6.4.8 *Application*

Une balise <Application> doit avoir un attribut 'Name' indiquant le nom de l'application, une balise <ApplicationConfigurations> dans laquelle sont définies les configurations possibles de l'application et une balise <ReasoningChains> contenant toutes les chaines de raisonnement.

A.6.4.9 *Exemple de fichier de configuration*

Le fichier XML suivant définit l'application représentée par la figure 120.

```
<application name="MyApplication">
        <applicationConfigurations>
                <applicationConfiguration name="AC1">
                        <gui name="GUI1"
                        class="application.myApplication.gui.GUI1"
inputs='1' outputs='1'/>
                        <serviceList>
                                <service name="S1"/>
                        </serviceList>
                        <connections>
                                <connection name="Gui\_to\_S1" from="GUI1" to="S1" />
                                <connection name="S1\_to\_Gui" from="S1" to="GUI1" />
                        </connections>
                </applicationConfiguration>
                <applicationConfiguration name="AC2">
                        <gui name="GUI2"
class="application.myApplication.gui.GUI2"
inputs='1' outputs='1'/>
                        <serviceList>
                                <service name="S1"/>
                                <service name="S2"/>
                        </serviceList>
                        <connections>
                                <connection name="Gui\_to\_S1" from="GUI2" to="S1" />
                                <connection name="S1\_to\_S2" from="S1" to="S2" />
                                <connection name="S2\_to\_Gui" from="S2" to="GUI2" />
                        </connections>
```

```
                </applicationConfiguration>
        </applicationConfigurations>

        <reasoningChains>
                <reasoningChain name="RC1" bpmnPath="./RC1.bpmn" />
                <reasoningChain name="RC2" bpmnPath="./RC2.bpmn" />
        </reasoningChains>

        <ontology name="MyOntology" path="./myOntology.owl" />
</application>
```

A.7 COMMENT ÉTENDRE KALIMUCHO-A

A.7.1 *Comment ajouter ses propres transformations d'unités et de représentation*

Des transformations personnalisées d'unités et de représentations peuvent être utilisées par les applications qui ont besoin pour leur recherche dans le contexte d'unités ou de représentations spécifiques. Kalimucho-A fourni d'ores et déjà une liste prédéfinie de transformateurs d'unités et de représentations.

L'ajout de transformateurs et d'unités de représentations peut être fait de façon statique par le développeur ou dynamiquement au runtime par l'application. Dans les deux cas, le développeur doit définir les transformateurs comme des composants Kalimucho. Il existe des classes abstraites qui permettent la réalisation de ces transformateurs.

Pour un transformateur d'unités, le développeur implémente un composant Kalimucho qui hérite de la classe abstraite UnitTranslator. Cette classe possède une méthode abstraite translate qui doit être implémentée par le développeur.

De façon identique, un transformateur de représentation est un composant Kalimucho qui hérite de la classe abstraite RepresentationTranslator. Cette classe possède aussi une méthode abstraite translate qui doit être implémentée par le développeur.

L'étape d'addition est différente si elle est réalisée de façon statique ou dynamique.

A.7.1.1 *Addition statique*

Pour ajouter un transformateur de façon statique au domaine, le développeur utilise le panneau d'administration et fournit les informations nécessaires sur le transformateur (le nom de la classe). Pour plus d'informations sur le panneau d'administration, se référer à la section A.7.4.

A.7.1.2 *Addition dynamique*

Le développeur peut aussi ajouter les informations du transformateur à l'intérieur de l'application au runtime. Bien entendu, les composants de transformation doivent avoir été compilés et stockés dans un dépôt de composants avant de lancer l'application. Pour ajouter les informations du transformateur au domaine, l'application doit se connecter au composant OntologyUpdater du DOCK. Ceci peut être fait en

utilisant la classe DockManager. L'application doit appeler la méthode connectToOntologyUpdater de la classe DockManager. Lorsque la connexion est établie, l'application peut envoyer un appel de méthode (en utilisant le KaliService KaliMethodCall) à l'OntologyUpdater concernant les méthodes addUnitTranslator ou addRepresentationTranslator. Ces deux méthodes acceptent en paramètre le nom de la classe du composant transformateur. Elles retournent un booléen indiquant si l'opération d'addition a été un succès ou pas.

A.7.2 *Comment ajouter des composants d'identification de situation*

Les composants d'identification de situation permettent aux applications de définir leurs propres situations spécifiques à partir des informations de contexte, d'événements d'applications ou d'autres situations déjà identifiées.

Un composant d'identification de situations est un composant qui reçoit des événements et produit une situation particulière de sorte que le service de prise de décisions peut les traiter et les résoudre.

A.7.2.1 *Développement d'un composant d'identification de situations*

Un composant d'identification de situations de l'application peut, mais n'est pas obligé, d'utiliser les services de notification de situations et de recherche dans le contexte pour détecter des événements hors de la portée de l'application.

Pour permettre de stocker les situations identifiées dans l'ontologie de situations, le composant d'identification de situations peut étendre la classe SituationIdentifier qui hérite de BCModel. Cette classe possède une méthode createSituation qui stocke une situation dans l'ontologie. Elle a aussi une méthode abstraite eventReceived qui sera appelée chaque fois que le composant recevra un événement de l'application.

Un événement envoyé par l'application au composant d'identification de situation doit être encapsulé dans un objet de classe EventSample. Cet objet contient un objet de classe KaliEvent qui inclut un nom et une liste de paramètres.

La figure 121 montre un composant d'identification de situations utilisant à la fois le service de notification de situations et le service de recherche dans le contexte. Il reçoit par ailleurs des événements de l'application

A.7.2.2 *Exemple de composant d'identification de situations*

```
@Override
protected void execute() throws StopBCException, InterruptedException {
        //The component will receive context information about the
//battery from the input port number 1.
        //The connection of this component to the context searching service
//and the query is not included in this code.
        //If you need to know how to use the context searching service,
        //refer to the How to perform a context search chapter in the
        //development guide of Kalimucho-A
        addInputListener(1, new InputListener() {
```

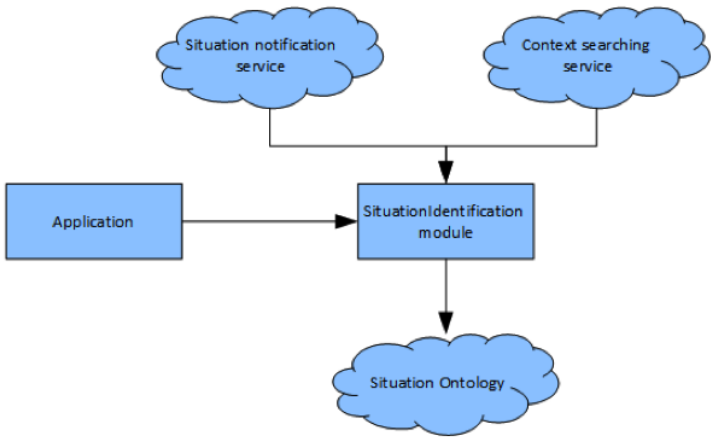

FIGURE 121 – Composant d'identification de situations

```
                @Override
                public void performSample(Sample sample) throws StopBCException,
                                        InterruptedException {
                        SensorDataSample sensorData = (SensorDataSample)sample;
                        ContextMetadataItem item = sensorData.getItem();
                        if (((Integer)item.getValue().getValue())<40){
                                runningOnLowBattery = true;
                                //Check for the other condition
                                if (runningOnBackground){
                                        createStopSituation();
                                }
                        }else{
                                runningOnLowBattery = false;
                        }
                }
        });
}

//Creates a need to stop application situation
private void createStopSituation() {
        Situation situation = new Situation("needToKillMyApp",
                Situation.NEED\_STOP\_APPLICATION);
        situation.addDataProperty("hasApplicationName", APP\_NAME);
        createSituation(situation);
}

@Override
protected void eventReceived(KaliEvent event) throws StopBCException,
InterruptedException {
        if ("RunningOnBackground".equals(event.getName())){
```

```
                runningOnBackground = true;
                //Check the other condition
                if (runningOnLowBattery){
                        createStopSituation();
                }
        }else{
                if ("ApplicationBackOnTop".equals(event.getName())){
                        runningOnBackground = false;
                }
        }
}
```

A.7.2.3 *Utilisation d'un composant d'identification de situation*

Pour pouvoir utiliser un composant existant le développeur peut faire appel à la classe SituationManager qui propose une méthode addSituationIdentificationModule pour ajouter un composant d'identification de situation et une méthode removeSituationIdentificationModule pour enlever un composant d'identification de situation. La première méthode prend en paramètre le nom du composant courant (récupéré de la méthode getName de BCModel), la classe du composant d'identification de situation et un numéro de sortie. Elle déploie le composant d'identification de situations et le connecte à la sortie indiquée du composant courant. Le composant de l'application pourra alors utiliser cette sortie pour envoyer des événements au composant d'identification de situation.

La méthode removeSituationIdentificationModule déconnecte et supprime le composant d'identification de situation désigné par son identifiant (retourné lors de sa création par la méthode addSituationIdentificationModule.

A.7.3 *Ajout d'exigences sur les composants, les services, l'application et les solutions*

A.7.3.1 *Exigences sur les composants*

Les exigences sur les composants sont contrôlées lorsque la plateforme cherche un périphérique compatible avec les composants à déployer. Le développeur peut définir son propre composant de vérification des exigences.

Kalimucho-A possède une classe abstraite ComponentRequirementChecker qui hérite de BCModel permettant l'implémentation de composants personnalisés de vérification d'exigences. Cette classe abstraite définit une méthode abstraite checkRequirement qui sera appelée lorsque l'exigence devra être vérifiée. Cette méthode reçoit en paramètre les informations sur le composant et le périphérique sur lequel Kalimucho tente de déployer ce composant. Elle renvoie un booléen indiquant si le périphérique est compatible ou pas avec ce composant.

C'est la seule méthode que le développeur doit écrire pour vérifier une exigence. Lorsque ce composant a été écrit et compilé, le développeur peut utiliser le panneau d'administration pour l'inclure à la plateforme.

Les exigences de composants peuvent également être utilisées pour indiquer les conditions souhaitées pour le déploiement d'un composant. Dans ce cas, ces exigences seront vérifiées seulement lorsqu'il reste plus d'un périphérique pouvant accepter le composant après vérification des exigences des composants.

A.7.3.2 *Exigences sur les services*

Les exigences sur les services peuvent être réalisées de façons similaires à celles sur les composants. Les exigences sur les services seront vérifiées lors de la sélection de la configuration offrant la meilleure qualité de service lors d'un déploiement ou d'un redéploiement. Kalimucho-A offre une classe abstraite ServiceRequirementChecker qui définit une méthode abstraite checkRequirements qui sera appelée lorsqu'une exigence doit être vérifiée. Cette méthode n'a pas de paramètres et doit renvoyer un booléen indiquant si l'exigence est vérifiée ou pas. De la même façon que pour les exigences de composants, il s'agit de la seule méthode que le développeur doit implémenter. Lorsque ce composant a été écrit et compilé, le développeur peut utiliser le panneau d'administration pour l'inclure à la plateforme.

A.7.3.3 *Exigences sur l'application*

Le développement des exigences sur l'application est identique à celui des exigences sur les services. Dans ce cas, la classe abstraite proposée par Kalimucho-A est ApplicationRequirementChecker. L'exigence sera vérifiée lors de la sélection de la meilleure configuration possible de l'application. Ceci se produira lors d'un déploiement ou d'un redéploiement. La méthode abstraite checkRequirement ne reçoit aucun paramètre et retourne un booléen indiquant si l'exigence est satisfaite ou pas. Lorsque ce composant a été écrit et compilé, le développeur peut utiliser le panneau d'administration pour l'inclure à la plateforme.

A.7.3.4 *Exigences sur les solutions*

Une exigence concernant une solution sera prise en compte lorsqu'une situation qui peut être résolue par une solution est identifiée. Le middleware Kalimucho-A offre une classe abstraite SolutionRequirementChecker qui définit une méthode checkRequirement appelée lorsqu'une exigence sur la solution doit être vérifiée. Cette méthode reçoit en paramètre les informations sur la situation et la solution. Elle retourne un booléen indiquant si l'exigence est satisfaite ou non. Lorsque ce composant a été écrit et compilé, le développeur peut utiliser le panneau d'administration pour l'inclure à la plateforme.

A.7.4 *Le panneau d'administration*

Le panneau d'administration (cf. figure.122) permet au développeur d'étendre le middleware Kalimucho-A en lui fournissant un moyen convivial pour inclure des informations et des composants dans les ontologies du middleware. Le panneau d'administration est accessible dès le démarrage de la plateforme et propose au développeur une liste d'opérations d'extension de la plateforme.

Cette partie ne présente pas comment construire les composants mais comment les inclure dans la plateforme. Pour des informations sur la façon de construire un type spécifique de composant, se référer à la section A.7.3.

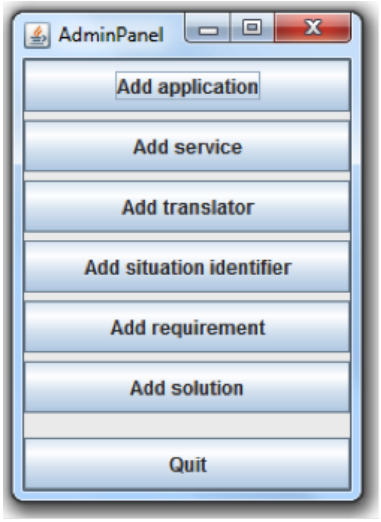

FIGURE 122 – Panneau d'administration

A.7.4.1 *Ajout d'applications*

Le bouton Add application ouvre une fenêtre dans laquelle le développeur peut choisir le fichier XML de définition de l'application qui décrit l'application ajoutée pour l'inclure dans le domaine.

A.7.4.2 *Ajout de services*

De la même façon, le bouton Add service ouvre une fenêtre dans laquelle le développeur peut choisir le fichier XML de définition du service qui décrit le service et l'inclut dans le domaine.

A.7.4.3 *Ajout de transformateurs*

Le bouton Add translator ouvre une fenêtre dans laquelle le développeur peut choisir le type de transformateur (d'unité ou de représentation), le nom de l'unité ou de la représentation et inclure le nom de la classe du composant qui sera démarré avec l'application.

A.7.4.4 *Ajout d'exigences sur les composants, les services et les applications*

Le bouton Add requirement checker ouvre une fenêtre dans laquelle le développeur peut sélectionner le type d'exigences (composant, service ou application), le nom de l'exigence, et le nom de la classe du composant qui sera démarré avec l'application.

A.7.4.5 *Ajout de composants d'identification de situations*

Le bouton Add situation identifier ouvre une fenêtre dans laquelle le développeur peut inclure le nom d'un composant d'identification de situations et le nom de la classe du composant qui sera démarré avec l'application.

A.7.4.6 *Ajout de solutions, d'exigences de solutions et d'actions liées aux solutions*

Le bouton Add solution ouvre une fenêtre permettant d'ajouter une nouvelle solution à la plateforme. Cette fenêtre permet de définit les informations nécessaires à cette nouvelle solution. Une solution doit

être liée à au moins une situation et avoir une action liée à la solution. Elle peut également être associée à une ou plusieurs exigences de solution, soit prédéfinie, soit personnalisée.

A.8 CAS D'UTILISATION

Pour montrer les possibilités du middleware, nous avons défini 2 cas d'utilisation. Ces 2 cas permettent de montrer comment le middleware simplifie le travail du développeur pour implémenter une application adaptable. L'application utilisée dans les deux cas est la même.

A.8.1 *L'application*

L'application choisie est un chat vidéo (cf. figure.123). Elle peut être déployée sur un PC, un périphérique Android ou une combinaison des 2 dans la mesure où elle est constituée de composants. Chaque composant assure une partie fonctionnelle de l'application. Pour cet exemple, il y a un composant qui capture de la vidéo à partir de la caméra, un composant qui capture du son à partir du micro et un composant présentant l'interface graphique. De plus, il existe des composants pour transférer l'information et pour recevoir le flux vidéo et audio. Toutefois, la transmission de flux vidéo et audio ne sera pas envisagée dans la suite. Le premier déploiement sur le smartphone de l'utilisateur est présenté dans la figure.124.

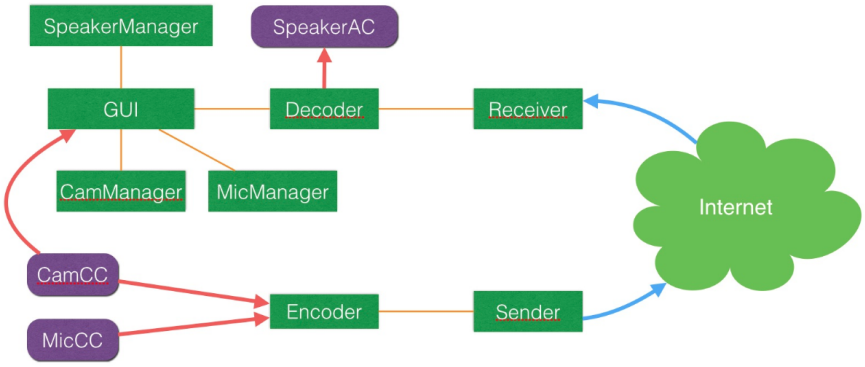

FIGURE 123 – Architecture de l'application exemple

A.8.2 *Cas d'utilisation : adaptation à une situation*

Dans ce cas d'utilisation, la plateforme va détecter deux situations d'adaptation dans la scène 2 : "New Device" et "User Interaction Change". La figure.125 montre la composition de l'application avant l'adaptation. Quand utilisateur a rentré chez lui, son smartphone a connecté avec le réseau locale. La plateforme a détecté le PC de l'utilisateur, à traverser le middleware de contexte Kali2Much et la chaine de raisonnement de "NewDeviceRC" (cf. section "Détection d'un hôte", page.128), l'instance de la situation "New Device" est ajouté dans l'ontologie KaliMOSA comme "IdentifiedASituation". La chaîne de raisonnement de "AppStateRC" a reçu une notification de cette situation "New Device". Elle a lancé une DL Query pour

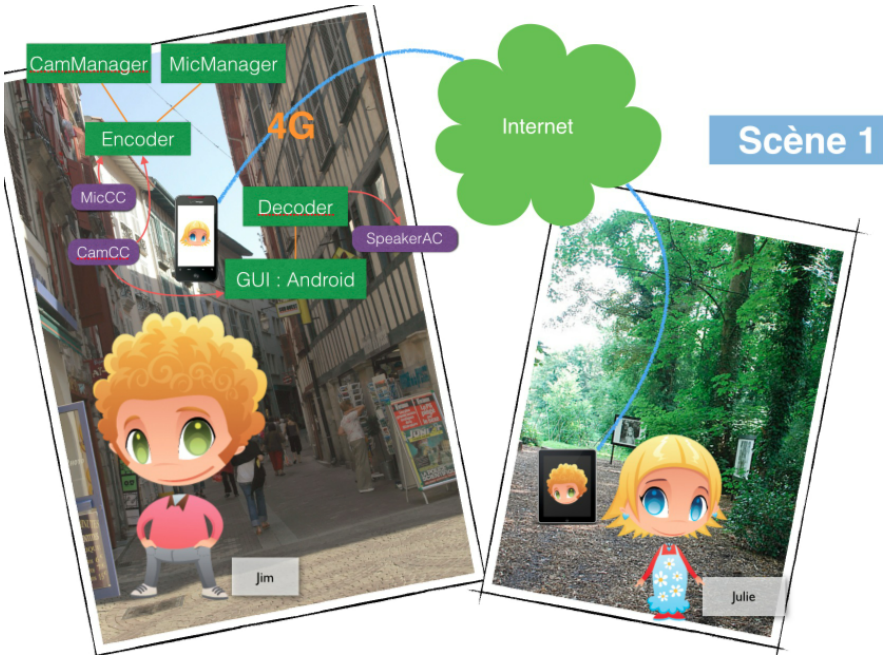

FIGURE 124 – Premier déploiement de l'application

chercher des composant exécuter dans l'état "ComponentRunsOnUndesiredCondition" (préférence d'exécution, page.66). La "AppStateRC" a trouvé que les deux composants "Decoder" et "Encoder" préférent de exécuter sur des hôte qui a la meilleur puissance de calcule. Elle a créé deux instances de la situation "Deploy Composant" avec une cause "ComponentRunsOnUndesiredCondition" pour redéployer les deux composants. Le Decision-Maker a reçu une notification de ces deux situations. Il a cherché dans l'ontologie de solution avec la situation et sa cause. Il a trouvé une solution qui a une action "Deploy Action". Le Decision-Maker a déployé son actionneur pour redéployer les deux composant. Le résultat de actionneur (i.e. calculer par notre heuristique algorithme) est déplacer les deux composants ver le PC comme dans la figure.126.

L'utilisateur login avec son PC, ceci est provoqué une identification de la situation "User Interaction Change". Cette situation va être notifié la "AppStateRC". Elle a trouvé la même situation comme pour les deux composants "Decoder" er "Encoder", mais cette fois-ci c'est pour le GUI. Le composant GUI de cette application préfére de s'exécuter sur des hôtes qui équipent de grande écran. La même processus a effectué une migration de GUI ver le PC. Finalement la composition de l'application a complètement changé comme dans la figure.126.

A.8.3 *Cas d'utilisation : situation d'application*

Dans ce second cas, l'application reste la même mais le développeur veut s'assurer que le microphone source change quand l'utilisateur s'en éloigne. Donc le développeur qu'il faut fournir une situation d'adaptation "AppAStiation", au moins une solution spécifique pour cette situation et une ou plusieurs chaîne(s) de raisonnement pour l'identification de situation. Dans ce cas, le développeur définit une situation nomé nomé "AppSitUserGoAway", une chaîne de raisonnement d'identification de situation pour cette situation et une solution nomé "SoluSitUserGoAway". Cette chaîne reçoit le niveau sonore du périphérique en utili-

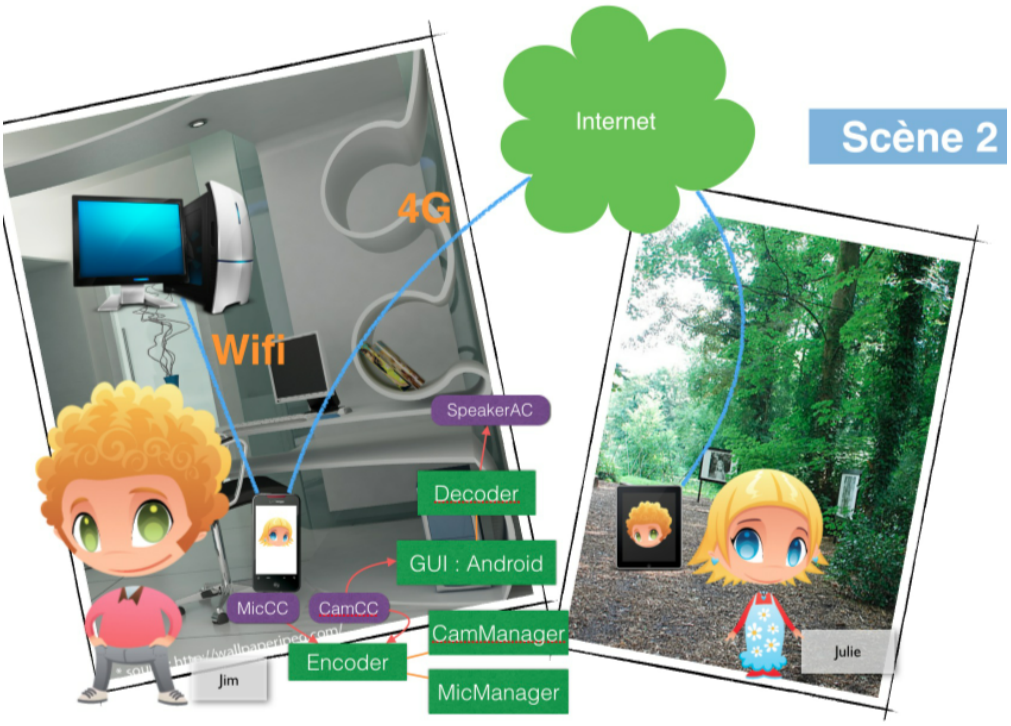

FIGURE 125 – Composition d'application au début de la scène 2

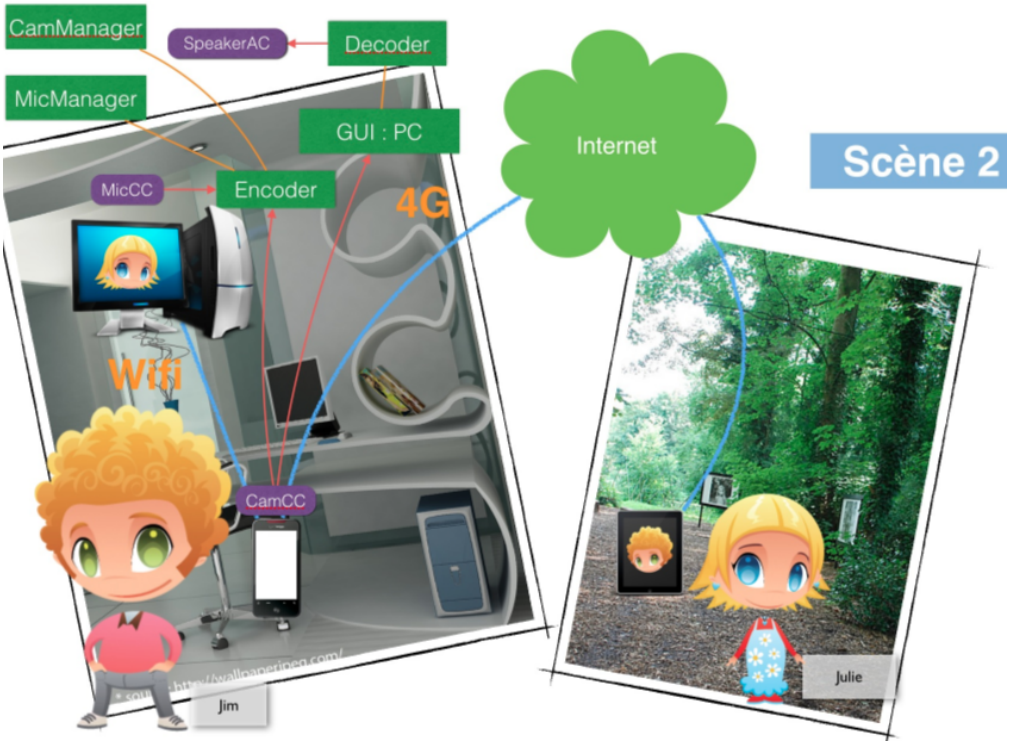

FIGURE 126 – Composition d'application après une adaptation dans la scène 2

sant le service de recherche du contexte. Lorsque le volume sonore est au-dessous d'un point critique, il créé un instance de la situation d'application "AppSitUserGoAway".

Cette situation sera réceptionnée par le service de notification de situation et notifié à l'application. L'application, par exemple, aura recours à une recherche dans le contexte pour trouver le périphérique ayant le meilleur volume sonore et elle migrera le composant d'enregistrement audio vers ce périphérique.

BIBLIOGRAPHIE

[1] Alessandra Agostini, Claudio Bettini, and Daniele Riboni. Hybrid reasoning in the care middleware for context awareness. *International journal of Web engineering and technology*, Volume 5(1) :3–23, 2009.

[2] Erwin Aitenbichler, Jussi Kangasharju, and Max Mühlhäuser. MundoCore : A light-weight infrastructure for pervasive computing. *Pervasive and Mobile Computing*, 3(4) :332 – 361, 2007. ISSN 1574-1192. doi : DOI:10.1016/j.pmcj.2007.04.002. URL http://www.sciencedirect.com/science/article/B7MF1-4NJ20N5-2/2/191cc8ab366f15eeb09a62f06d2a84d3.

[3] OSGi Alliance. *OSGi Technology*. OSGi Alliance, 2011. URL http://www.osgi.org/About/Technology.

[4] Mehdi Amoui, Mazeiar Salehie, Siavash Mirarab, and Ladan Tahvildari. Adaptive Action Selection in Autonomic Software Using Reinforcement Learning. pages 175–181, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3093-2. doi : 10.1109/ICAS.2008.35. URL http://portal.acm.org/citation.cfm?id=1395980.1396120.

[5] C.B. Anagnostopoulos, Y. Ntarladimas, and S. Hadjiefthymiades. Situational computing : An innovative architecture with imprecise reasoning. *Journal of Systems and Software*, 80(12) :1993 – 2014, 2007. ISSN 0164-1212. doi : DOI:10.1016/j.jss.2007.03.003. URL http://www.sciencedirect.com/science/article/B6V0N-4N7XP3X-1/2/fa0bdd0b00f3ebb914729fc4c4aac1cc.

[6] Michalis Anastasopoulos, Holger Klus, Jan Koch, Dirk Niebuhr, and Ewoud Werkman. DoAmI- A Middleware Platform facilitating (Re-)configuration in Ubiquitous Systems. Irvine, 2006.

[7] Francoise Andre, Guillaume Gauvrit, and Christian Perez. Dynamic Adaptation of the Master-Worker Paradigm. CIT '09, pages 185–190, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3836-5. doi : http://dx.doi.org/10.1109/CIT.2009.40. URL http://dx.doi.org/10.1109/CIT.2009.40.

[8] Apple. *About iOS Application Design*. Apple, 2011. URL http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html%23//apple_ref/doc/uid/TP40007072.

[9] Naveed Arshad, Dennis Heimbigner, and Alexander L Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Control*, 15 :265–281, September 2007. ISSN 0963-9314. doi : 10.1007/s11219-007-9019-2. URL http://portal.acm.org/citation.cfm?id=1286061.1286070.

[10] Marco Autili, Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. A conceptual model for adaptable context-aware services. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, page 15, 2006.

[11] Dhouha Ayed, Chantal Taconet, Guy Bernard, and Yolande Berbers. CADeComp : Context-aware deployment of component-based applications. *Journal of Network and Computer Applications*, 31(3) :224 – 257, 2008. ISSN 1084-8045. doi : DOI:10.1016/j.jnca.2006.12.002. URL http://www.sciencedirect.com/science/article/B6WKB-4MY0MPP-1/2/aebc55549668a48efc5f51a6d89b0bc1.

[12] N Badr, A Taleb-Bendiab, and D Reilly. Policy-based autonomic control service. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 99–102. IEEE, 2004.

[13] Faruk Bagci, Jan Petzold, Wolfgang Trumler, and Theo Ungerer. Ubiquitous mobile agent system in a P2P-Network. In *System Support for Ubiquitous Computing Workshop at 6th Annual Conference on Ubiquitous Computing*, pages 12–15, Nottingham, England, 2004.

[14] C. Ballagny, N. Hameurlain, and F. Barbier. Mocas : A state-based component model for self-adaptation. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, pages 206–215, Sept 2009. doi : 10.1109/SASO.2009.11.

[15] Ling Bao and Stephen S Intille. Activity recognition from user-annotated acceleration data. In *Pervasive computing*, pages 1–17. Springer, 2004.

[16] Joseph Bauer. Identification and modeling of contexts for different information scenarios in air traffic. In *Diplomarbeit*. 2003.

[17] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. BASE " A Micro-Broker-Based Middleware for Pervasive Computing. PERCOM '03, pages 443–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1893-1. URL http://portal.acm.org/citation.cfm?id=826025.826336.

[18] Amira Ben Hamida, Frédéric Le Mouël, Stéphane Frénot, and Mohamed Ben Ahmed. Déploiement adaptatif d'applications orientées services sur environnements contraints. *Technique et Science Informatiques*, 30 :59–91, 2011. URL http://hal.inria.fr/inria-00534596/PDF/TSI-AxSeL-lemouel.pdf.

[19] Ghada. Ben Nejma, P. Roose, D. Marc, J. Gensel, and G.M. Amine. A semantic social software for mobile environment. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 723–727, Oct 2013. doi : 10.1109/3PGCIC.2013.123.

[20] Ghada Ben Nejma, Philippe Roose, Jérôme Gensel, and Marc Dalmau. Taldea : une application communautaire avec géolocalisation. In *31ème Conférence INFORSID*, volume 1, pages 165–180, Paris, France, May 2013. INFORSID. URL https://hal.archives-ouvertes.fr/hal-00829311.

[21] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2) :161–180, April 2010. ISSN 15741192. doi : 10.1016/j.pmcj.2009.06.002. URL http://linkinghub.elsevier.com/retrieve/pii/S1574119209000510.

[22] Guy Bieber and Jeff Carpenter. Introduction to service-oriented programming (rev 2.1). *OpenWings Whitepaper (April 2001)*, 2001.

[23] David L Black, David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, et al. Microkernel operating system architecture and mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–31, 1992.

[24] G. S Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. Middleware '98, pages 191–206, London, UK, 1998. Springer-Verlag. ISBN 1-85233-088-0. URL http://portal.acm.org/citation.cfm?id=1659232.1659249.

[25] Daniel G Bobrow, Richard P Gabriel, and Jon L White. CLOS in context : the shape of the design space. pages 29–61. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-16136-2. URL http://portal.acm.org/citation.cfm?id=166848.166854.

[26] Cristiana Bolchini, Fabio A Schreiber, and Letizia Tanca. A methodology for a very small data base design. *Information Systems*, 32(1) :61–82, 2007.

[27] Emmanuel Bouix, Marc Dalmau, Philippe Roose, and Franck Luthon. A multimedia oriented component model. In *The IEEE 19th International Conference on Advanced Information Networking and Applications, AINA 2005*, volume 1, pages 3–8. IEEE, 2005.

[28] Emmanuel Bouix, Philippe Roose, and Marc Dalmau. The Korrontea data modeling. Ambi-Sys '08, pages 6 :1–6 :10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-16-5. URL http://portal.acm.org/citation.cfm?id=1363163.1363169.

[29] Amel Bouzeghoub, Chantal Taconet, Amina Jarraya, Ngoc Kien Do, and Denis Conan. Complementarity of process-oriented and ontology-based context managers to identify situations. In *Digital Information Management (ICDIM), 2010 Fifth International Conference on*, pages 222–229. IEEE, 2010.

[30] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.98. *Namespace Document*, Volume 9, 2012.

[31] Pier Luigi Buttigieg, Norman Morrison, Barry Smith, Christopher J Mungall, and Suzanna E Lewis. The environment ontology : contextualising biological and biomedical entities. *Journal of biomedical semantics*, 4(1) :43, January 2013. ISSN 2041-1480. doi : 10.1186/2041-1480-4-43. URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3904460&tool=pmcentrez&rendertype=abstract.

[32] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19 :332–383, August 2001. ISSN 0734-2071. doi : http://doi.acm.org/10.1145/380749.380767. URL http://doi.acm.org/10.1145/380749.380767.

[33] Djalel Chefrour. Developing component based adaptive applications in mobile environments. SAC '05, pages 1146–1150, New York, NY, USA, 2005. ACM. ISBN 1-58113-964-0. doi : http://doi.acm.org/10.1145/1066677.1066935. URL http://doi.acm.org/10.1145/1066677.1066935.

[34] Harry Chen, Tim Finin, and Anupam Joshi. Using OWL in a Pervasive Computing Broker. In *Workshop on Ontologies in Agent Systems, AAMAS-2003*, 2003.

[35] Harry Chen, Tim Finin, and Anupam Joshi. Semantic Web in the Context Broker Architecture. PERCOM '04, pages 277–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2090-1. URL http://portal.acm.org/citation.cfm?id=977406.978667.

[36] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. Soupa : Standard ontology for ubiquitous and pervasive applications. In *Mobile and Ubiquitous Systems : Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 258–267. IEEE, 2004.

[37] Betty H Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-02160-2.

[38] Keith Cheverst, Keith Mitchell, and Nigel Davies. Design of an object model for a context sensitive tourist guide. *Computers & Graphics*, 23(6) :883–891, 1999.

[39] Diane J Cook, Juan C Augusto, and Vikramaditya R Jakkula. Ambient intelligence : Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4) :277–298, 2009. URL http://www.sciencedirect.com/science/article/B7MF1-4W2W5PH-1/2/7ac602bb16ca409e2f70c13aa24ae0d5.

[40] Keling Da, Philippe Roose, and Marc Dalmau. WaterCOM : An Architecture Model of Context-Oriented Middleware. *FINA Workshop help at The 26th IEEE International Conference on Advanced Information Networking and Applications (AINA-2012)*, 2012. URL http://ieeexplore.ieee.org/iel5/6184371/6185080/06185099.pdf?arnumber=6185099.

[41] Keling Da, Marc Dalmau, and Philippe Roose. Kalimucho : Middleware for mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 413–419, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2469-4. doi : 10.1145/2554850.2554883. URL http://doi.acm.org/10.1145/2554850.2554883.

[42] Marc Dalmau and Philippe Roose. Kalimucho : Plateforme logicielle distribuée de supervision d'applications. *7ème conférence francophone sur les architectures logicielles (CAL 2013)*. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Kalimucho+:+Plateforme+logicielle+distribuée+de+supervision+d+\T1\textquoteright+applications#0.

[43] Marc Dalmau, Philippe Roose, and Sophie Laplace. Context aware adaptable applications-a global approach. *arXiv preprint arXiv :0909.2090*, 2009.

[44] Romain Rouvoy Denis Conan. Scalable Processing of Context Information with COSMOS. *Distributed Applications and Interoperable Systems*, pages 210–224, 2007.

[45] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1) :4–7, 2001.

[46] Nicholas DiGiuseppe, Line C. Pouchard, and Natalya F. Noy. SWEET ontology coverage for earth system sciences. *Earth Science Informatics*, January 2014. ISSN 1865-0473. doi : 10.1007/s12145-013-0143-1. URL http://link.springer.com/10.1007/s12145-013-0143-1.

[47] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1 :223–259, December 2006. ISSN 1556-4665. doi : http://doi.acm.org/10.1145/1186778.1186782. URL http://doi.acm.org/10.1145/1186778.1186782.

[48] Jim Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, University of Dublin, Trinity College. Department of Computer Science, 2004.

[49] Frank Eliassen, Eli Gj\orven, Viktor S. Wold Eide, and J\orgen Andreas Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. ARM '06, pages 1–, New York, NY, USA, 2006. ACM. ISBN 1-59593-419-7. doi : http://doi.acm.org/10.1145/1175855.1175856. URL http://doi.acm.org/10.1145/1175855.1175856.

[50] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35 :114–131, June 2003. ISSN 0360-0300. doi : http://doi.acm.org/10.1145/857076.857078. URL http://doi.acm.org/10.1145/857076.857078.

[51] J Floch, R Fricke, and K Geihs. Playing MUSIC — building context-aware and self-adaptive mobile applications. *Software : Practice*, (7465), 2012. doi : 10.1002/spe. URL http://onlinelibrary.wiley.com/doi/10.1002/spe.2116/full.

[52] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software : A Research Roadmap. FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi : http://dx.doi.org/10.1109/FOSE.2007.14. URL http://dx.doi.org/10.1109/FOSE.2007.14.

[53] Andrew U Frank. Tiers of ontology and consistency constraints in geographical information systems. *International Journal of Geographical Information Science*, 15(7) :667 – 678, 2001. URL http://www.informaworld.com/10.1080/13658810110061144.

[54] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U Khan, A. Mamelli, G. A Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation. *Softw. Pract. Exper.*, 39 :385–422, March 2009. ISSN 0038-0644. doi : 10.1002/spe.v39:4. URL http://portal.acm.org/citation.cfm?id=1527067.1527070.

[55] Asuncion Gomez-Perez, Oscar Corcho-Garcia, and Mariano Fernandez-Lopez. *Ontological Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 1852335513.

[56] Google. *What is android*. Google, 2011. URL http://developer.android.com/guide/basics/what-is-android.html.

[57] Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. Technical report, Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS.

[58] Tao Gu, H. K. Pung, D. Q. Zhang, Hung Keng Pung, and Da Qing Zhang. A Bayesian Approach For Dealing With Uncertain Contexts. In *Austrian Computer Society*, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.6.3509.

[59] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *Proceedings of communication networks and distributed systems modeling and simulation conference*, volume 2004, pages 270–275, 2004.

[60] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28 :1–18, January 2005. ISSN 1084-8045. doi : 10.1016/j.jnca.2004.06.002. URL http://portal.acm.org/citation.cfm?id=1053030.1053031.

[61] N. Guarino. *Formal Ontology in Information Systems : Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1st edition, 1998. ISBN 9051993994.

[62] Pari Delir Haghighi, Shonali Krishnaswamy, Arkady Zaslavsky, and Mohamed Medhat Gaber. Reasoning about context in uncertain pervasive computing environments. In *Smart Sensing and Context*, pages 112–125. Springer, 2008.

[63] P.D. Haghighi, A. Zaslavsky, and S. Krishnaswamy. An Evaluation of Query Languages for Context-Aware Computing. *17th International Conference on Database and Expert Systems Applications (DEXA'06)*, pages 455–462, 2006. doi : 10.1109/DEXA.2006.25. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1698385.

[64] Terry Halpin. *Information Modeling and Relational Databases : From Conceptual Analysis to Logical Design*. The Morgan Kaufmann Series in Data Management Systems, 2001. ISBN 1558606726.

[65] Amira Ben Hamida, Frédéric Le Mouël, Stéphane Frénot, Mohamed Ben Ahmed, et al. Déploiement adaptatif d'applications orientées services sur environnements contraints. *Revue Technique et Science Informatiques*, 30(1) :59–91, 2011.

[66] Md Kamrul Hasan, Husne Ara Rubaiyeat, Yong-Koo Lee, and Sungyoung Lee. A reconfigurable hmm for activity recognition. In *ICACT*, volume 8, pages 843–846, 2008.

[67] Dennis Heimbigner and Alexander Wolf. Intrusion Management Using Configurable Architecture Models. Technical report, DTIC Document, 2002.

[68] Karen Henricksen, Steven Livingstone, and Jadwiga Indulska. Towards a hybrid approach to context modelling, reasoning and interoperation. In *Proceedings of the First International Workshop on Advanced Context Modelling, Reasoning And Management, in conjunction with UbiComp*, 2004.

[69] Klaus Herrmann, Gero Mühl, and Michael A. Jaeger. MESHMdl event spaces – A coordination middleware for self-organizing applications in ad hoc networks. *Pervasive and Mobile Computing*, 3(4) : 467–487, August 2007. ISSN 1574-1192. doi : doi:DOI:10.1016/j.pmcj.2007.04.003. URL http://www.sciencedirect.com/science/article/B7MF1-4NJG450-2/2/796a13f5a01dedd70f4e8b773ed24077.

[70] Jerry R Hobbs and Feng Pan. An ontology of time for the semantic web. *ACM Transactions on Asian Language Information Processing (TALIP)*, 3(1) :66–85, 2004.

[71] Xin Hong, Chris Nugent, Maurice Mulvenna, Sally McClean, Bryan Scotney, and Steven Devlin. Evidential fusion of sensor data for activity recognition in smart homes. *Pervasive and Mobile Computing*,

5(3) :236–252, June 2009. ISSN 1574-1192. doi : doi:DOI:10.1016/j.pmcj.2008.05.002. URL http://www.sciencedirect.com/science/article/B7MF1-4SFXKCT-2/2/059ef0f63d47472d5104cdfae18038e1.

[72] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. In *2007 Future of Software Engineering*, pages 244–258. IEEE Computer Society, 2007.

[73] Takayuki Kanda, Dylan F Glas, Masahiro Shiomi, Hiroshi Ishiguro, and Norihiro Hagita. Who will be the customer ? : a social robot that anticipates people's behavior from their trajectories. UbiComp '08, pages 380–389, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-136-1. doi : http://doi.acm.org/10.1145/1409635.1409686. URL http://doi.acm.org/10.1145/1409635.1409686.

[74] John Keeney and Vinny Cahill. Chisel : A policy-driven, context-aware, dynamic adaptation framework. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14. IEEE, 2003.

[75] Jeffrey O Kephart and William E Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12. IEEE, 2004.

[76] Charles H. Kepner and Benjamin B. Tregoe. *The Rational Manager : A Systematic Approach to Problem Solving and Decision-Making*. McGraw-Hill, 1965.

[77] An Phung Khac. *A model-driven feature-based approach to runtime adaptation of distributed software architectures*. PhD thesis, Télécom Bretagne, 2010.

[78] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irwin, Mehmet Akşit, and Satoshi Matsuoka. Aspect-oriented programming. volume 1241, pages 220–242. Springer-Verlag, 1997. ISBN 3-540-63089-9. URL http://dx.doi.org/10.1007/BFb0053381.

[79] Danny B Lange and Oshima Mitsuru. *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1998. ISBN 0201325829.

[80] Sophie Laplace, Marc Dalmau, and Philippe Roose. Kalinahia-Modèle de qualité de service pour les applications multimédia reconfigurables. *Numéro Spécial Revue ISI'Conception : patrons et spécifications formelles*, 4(12) :ISBN : 978–2–7462–1968–7, 2007. URL https://hal.archives-ouvertes.fr/hal-00346866.

[81] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM : a component framework for the construction of resource-aware, reconfigurable MOMs. ARM '04, pages 250–255, New York, NY, USA, 2004. ACM. ISBN 1-58113-949-7. doi : http://doi.acm.org/10.1145/1028613.1028625. URL http://doi.acm.org/10.1145/1028613.1028625.

[82] Seng W. Loke. Incremental awareness and compositionality : A design philosophy for context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2) :239–253, April 2010. ISSN 1574-1192. doi : doi:DOI:10.1016/j.pmcj.2009.03.004. URL http://www.sciencedirect.com/science/article/B7MF1-4VYP9FC-1/2/b230c752476051a4c91aa8b2f59ee46f.

[83] Christine Louberry, Philippe Roose, and Marc Dalmau. Kalimucho : Contextual deployment for qos management. In *Distributed Applications and Interoperable Systems*, pages 43–56. Springer, 2011.

[84] Pattie Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6(1-2) :49–70, June 1990. ISSN 0921-8890. doi : doi:DOI:10.1016/S0921-8890(05)80028-4. URL http://www.sciencedirect.com/science/article/B6V16-4KBW026-6/2/f54b164b2d9b79a8aaad4602a812b498.

[85] Marco Mamei and Franco Zambonelli. Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. PERCOM '04, pages 263–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2090-1. URL http://portal.acm.org/citation.cfm?id=977406.978680.

[86] John McCarthy. Notes on formalizing context, 1993. URL http://cogprints.org/418/.

[87] John McCarthy and Sasa Buvac. Formalizing context (expanded notes), 1997. URL http://cogprints.org/419/.

[88] Susan Mckeever, Juan Ye, Lorcan Coyle, Chris Bleakley, and Simon Dobson. Activity recognition using temporal evidence theory. *J. Ambient Intell. Smart Environ.*, 2 :253–269, August 2010. ISSN 1876-1364. URL http://portal.acm.org/citation.cfm?id=1834668.1834671.

[89] Microsoft. *Microsoft .NET Framework*. Microsoft, 2011. URL http://www.microsoft.com/net/default.aspx.

[90] Alina Dia Miron. *Semantic association discovery for the Geospatial Semantic Web - the ONTOAST framework*. Theses, Université Joseph-Fourier - Grenoble I, December 2009. URL https://tel.archives-ouvertes.fr/tel-00635118.

[91] Darnell Moore and Irfan Essa. Recognizing multitasked activities from video using stochastic context-free grammar. pages 770–776, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence. ISBN 0-262-51129-0. URL http://portal.acm.org/citation.cfm?id=777092.777211.

[92] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. VieDAME - flexible and robust BPEL processes through monitoring and adaptation. ICSE Companion '08, pages 917–918, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi : http://doi.acm.org/10.1145/1370175.1370186. URL http://doi.acm.org/10.1145/1370175.1370186.

[93] Daniela Nicklas and Bernhard Mitschang. The Nexus Augmented World Model : An Extensible Approach for Mobile, Spatially-Aware Applications. In Yingxu Wang, Shushma Patel, and Ronald Johnston, editors, *Proceedings of the 7th International Conference on Object-Oriented Information Systems : OOIS '01 ; Calgary, Canada, August 27-29, 2001*, pages 392–401, London, January 2001. Springer-Verlag. ISBN 1-85233-546-7. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2001-43&engl=1.

[94] OMG. *Unified Modeling Language : Infrastructure*, 2005. URL http://www.omg.org/spec/UML/2.0/.

[95] OMG. *Model Driven Architecture (MDA) Specification*. OMG, 2010. URL http://www.omg.org/mda/specs.htm.

[96] OMG. *Business Process Model and Notation (BPMN)*, 2013. URL http://www.omg.org/spec/BPMN/2.0.2/.

[97] Open SOA. Service Component Architecture Specifications, 2007. URL http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

[98] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, 14(3) :54–62, 1999.

[99] Ming Ouyang, Aaron T. Garnett, Tina M. Han, Kotaro Hama, Amy Lee, Yun Deng, Nancy Lee, Hsing-Yin Liu, Sharon L. Amacher, Steven A. Farber, and Shiu-Ying Ho. A web based resource characterizing the zebrafish developmental profile of over 16,000 transcripts. *Gene Expression Patterns*, 8(3) :171 – 180, 2008. ISSN 1567-133X. doi : http://dx.doi.org/10.1016/j.gep.2007.10.011. URL http://www.sciencedirect.com/science/article/pii/S1567133X07001445.

[100] Shwetak Patel, Thomas Robertson, Julie Kientz, Matthew Reynolds, and Gregory Abowd. At the Flick of a Switch : Detecting and Classifying Unique Electrical Events on the Residential Power Line (Nominated for the Best Paper Award). pages 271–288. 2007. URL http://dx.doi.org/10.1007/978-3-540-74853-3_16.

[101] Donald Patterson, Lin Liao, Dieter Fox, and Henry Kautz. Inferring High-Level Behavior from Low-Level Sensors. pages 73–89. 2003. URL http://www.springerlink.com/content/k3x004g773qj80kg.

[102] Renaud Pawlak, Laurence Duchien, Gérard Florin, and Lionel Seinturier. Jac : A flexible solution for aspect-oriented programming in java. In *Metalevel architectures and separation of crosscutting concerns*, pages 1–24. Springer, 2001.

[103] C Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10) :46–52, 2003. ISSN 0018-9162. URL http://dx.doi.org/10.1109/MC.2003.1236471.

[104] Gian Pietro Picco, Gianpaolo Cugola, and Amy L Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. ICDCS '03, pages 234–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1920-2. URL http://portal.acm.org/citation.cfm?id=850929.851945.

[105] María Poveda-villalón, Mari Carmen Suárez-figueroa, and Raúl García-castro. A context ontology for mobile environments. *Workshop on Context, Information and Ontologies - CIAO 2010 Co-located with EKAW 2010*, 2010. URL http://oa.upm.es/5414/.

[106] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an extensible context ontology for ambient intelligence. In *Ambient intelligence*, pages 148–159. Springer, 2004.

[107] Anand Ranganathan, Robert E McGrath, Roy H Campbell, and M Dennis Mickunas. Use of ontologies in a pervasive computing environment. *The Knowledge Engineering Review*, 18(03) :209–220, 2003. URL http://portal.acm.org/citation.cfm?id=991804.991807http://srg.cs.uiuc.edu/2K/Gaia/papers/new080405/iaai-03.pdf.

[108] Daniel Romero, Romain Rouvoy, Lionel Seinturier, and Sophie Chabridon. Enabling Context-Aware Web Services : A Middleware Approach for Ubiquitous Environments. pages 113–135, 2010. URL http://hal.inria.fr/inria-00414070/PDF/chapitre.pdfhttp://hal.archives-ouvertes.fr/inria-00414070/.

[109] Philippe Roose. De la réutilisation à l'adaptabilite (HDR). Université de Pau et des Pays de l'Adour, 2008.

[110] Romain Rouvoy, Mikaël Beauvois, Laura Lozano, Jorge Lorenzo, and Frank Eliassen. MUSIC : an autonomous platform supporting self-adaptive mobile applications. MobMid '08, pages 6 :1–6 :6, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-362-4. doi : http://doi.acm.org/10.1145/1462689.1462697. URL http://doi.acm.org/10.1145/1462689.1462697.

[111] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns : A revised view. Technical Report BPM-06-22, BPM Center, 2006. URL http://bpmcenter.org/wp-content/uploads/reports/2006/BPM-06-22.pdf.

[112] Stuart Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach (2nd Edition)*. Prentice Hall, December 2002. ISBN 0137903952. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0137903952.

[113] MS Ryoo and JK Aggarwal. Recognition of Composite Human Activities through Context-Free Grammar Based Representation. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 2 :1709–1718, October 2006. ISSN 1063-6919. URL http://dx.doi.org/10.1109/CVPR.2006.242.

[114] Daniel Salber, Anind K Dey, and Gregory D Abowd. The context toolkit : aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 434–441. ACM, 1999.

[115] Mazeiar Salehie, Sen Li, Reza Asadollahi, and Ladan Tahvildari. Change Support in Adaptive Software : A Case Study for Fine-Grained Adaptation. pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3623-1. doi : 10.1109/EASe.2009.11. URL http://portal.acm.org/citation.cfm?id=1545015.1545700.

[116] Mazeiar Salehie, Sen Li, and Ladan Tahvildari. Employing aspect composition in adaptive software systems : a case study. PLATE '09, pages 17–21, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-453-9. doi : http://doi.acm.org/10.1145/1509847.1509851. URL http://doi.acm.org/10.1145/1509847.1509851.

[117] André C Santos, João MP Cardoso, Diogo R Ferreira, Pedro C Diniz, and Paulo Chaínho. Providing user context for mobile and social networking applications. *Pervasive and Mobile Computing*, 6(3) :324–341, 2010. URL http://www.sciencedirect.com/science/article/B7MF1-4Y4R4H2-1/2/54da1778cea288847658b400b954043d.

[118] Ichiro Satoh. Physical Mobility and Logical Mobility in Ubiquitous Computing Environments. MA '02, pages 186–202, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00085-2. URL http://portal.acm.org/citation.cfm?id=647631.732730.

[119] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.

[120] Albrecht Schmidt, Michael Beigl, and Hans-W Gellersen. There is more to context than location. *Computers & Graphics*, 23(6) :893–901, 1999.

[121] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2 : Patterns for Concurrent and Networked Objects*. Wiley, 2000. ISBN 0471606952. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0471606952.

[122] Holger Schmidt. *SAMProc : A Middleware for Highly Dynamic and Heterogeneous Environments*. PhD thesis, Universität Ulm, 2010. URL http://en.scientificcommons.org/53767580.

[123] Holger Schmidt and Franz J Hauck. SAMProc : middleware for self-adaptive mobile processes in heterogeneous ubiquitous environments. MDS '07, pages 11 :1–11 :6, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-933-3. doi : http://doi.acm.org/10.1145/1377934.1377935. URL http://doi.acm.org/10.1145/1377934.1377935.

[124] Dianxi Shi, Bo Ding, Gang Yin, Jin Feng, and Huaimin Wang. Research on policy-driven software self-adaptation mechanism. *Journal of Frontiers of Computer Science and Technology*, 4(2) :115–123, 2010.

[125] João Pedro Sousa and David Garlan. Aura : an architectural framework for user mobility in ubiquitous computing environments. In *Software Architecture*, pages 29–43. Springer, 2002.

[126] Thomas Strang and Claudia Linnhoff-Popien. A Context Modeling Survey. *Workshop Proceedings. First International Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp*, 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.2060.

[127] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge engineering : principles and methods. *Data Knowl. Eng.*, 25 :161–197, March 1998. ISSN 0169-023X. doi : 10.1016/S0169-023X(97)00056-6. URL http://portal.acm.org/citation.cfm?id=290547.290553.

[128] Sun. *JSR 220 :Enterprise JavaBeansTM*. Sun, version 3 edition, 2005.

[129] Sun. *Jini framework*. Sun, 2007. URL http://www.jini.org/wiki/Main_Page.

[130] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201745720.

[131] EmmanuelMunguia Tapia, StephenS. Intille, and Kent Larson. Activity recognition in the home using simple and ubiquitous sensors. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive Computing*, volume 3001 of *Lecture Notes in Computer Science*, pages 158–175. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21835-7. doi : 10.1007/978-3-540-24646-6_10. URL http://dx.doi.org/10.1007/978-3-540-24646-6_10.

[132] G Tesauro. Reinforcement Learning in Autonomic Computing : A Manifesto and Case Studies. *IEEE Internet Computing*, 11(1) :22–30, 2007. URL http://dx.doi.org/10.1109/MIC.2007.21.

[133] Gerald Tesauro, David M Chess, William E Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O Kephart, and Steve R White. A Multi-Agent Systems Approach to Autonomic Computing. AAMAS '04, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 1-58113-864-4. doi : http://dx.doi.org/10.1109/AAMAS.2004.23. URL http://dx.doi.org/10.1109/AAMAS.2004.23.

[134] Graham Thomson, Sotirios Terzis, and Paddy Nixon. Situation Determination with Reusable Situation Specifications. PERCOMW '06, pages 620–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2520-2. doi : http://dx.doi.org/10.1109/PERCOMW.2006.126. URL http://dx.doi.org/10.1109/PERCOMW.2006.126.

[135] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. WComp middleware for ubiquitous computing : Aspects and composite event-based Web services. *Annals of Telecommunications - Annales Des Télécommunications*, 64(3-4) :197–214, January 2009. ISSN 0003-4347. doi : 10.1007/s12243-008-0081-y. URL http://www.springerlink.com/index/10.1007/s12243-008-0081-y.

[136] P Turaga, R Chellappa, VS Subrahmanian, and O Udrea. Machine Recognition of Human Activities : A Survey. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(11) :1473–1488, 2008. URL http://dx.doi.org/10.1109/TCSVT.2008.2005594.

[137] Douglas L Vail, Manuela M Veloso, and John D Lafferty. Conditional random fields for activity recognition. AAMAS '07, pages 235 :1–235 :8, New York, NY, USA, 2007. ACM. ISBN 978-81-904262-7-5. doi : http://doi.acm.org/10.1145/1329125.1329409. URL http://doi.acm.org/10.1145/1329125.1329409.

[138] Wil MP van der Aalst. Patterns and xpdl : A critical evaluation of the xml process definition language. *BPM Center Report BPM-03-09, BPMcenter.org*, pages 1–30, 2003. URL http://www.ww.workflowpatterns.com/documentation/documents/ce-xpdl.pdf.

[139] T van Kasteren and B Krose. Bayesian activity recognition in residence for elders. pages 209–212, 2007. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4449935.

[140] Tim van Kasteren, Athanasios Noulas, Gwenn Englebienne, and Ben Kröse. Accurate activity recognition in a home setting. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 1–9, Seoul, Korea, 2008. ACM. ISBN 978-1-60558-136-1. URL http://dx.doi.org/10.1145/1409635.1409637.

[141] Kunal Verma and Amit Sheth. Autonomic Web Processes. volume 3826 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg, 2005. URL http://dx.doi.org/10.1007/11596141_1.

[142] W3C. *Composite Capabilities / Preferences Profile (CC/PP)*. W3C. URL http://www.w3.org/Mo-bile/CCPP.

[143] Edwin JY Wei and Alvin TS Chan. Campus : A middleware for automated context-aware adaptation decision making at run time. *Pervasive and Mobile Computing*, 9(1) :35–56, November 2013. URL http://linkinghub.elsevier.com/retrieve/pii/S1574119211001283.

[144] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7) :75–84, 1993.

[145] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, J. Perry, S. Herzog, A.-N. Huynh, and Carlson. Terminology for policy-based management. 2000.

[146] Danny Weyns. *An architecture-centric approach for software engineering with situated multiagent systems*. PhD thesis, KATHOLIEKE UNIVERSITEIT LEUVEN, 2006.

[147] Christian Wojek, Kai Nickel, and Rainer Stiefelhagen. Activity recognition and room-level tracking in an office environment. In *Multisensor Fusion and Integration for Intelligent Systems, 2006 IEEE International Conference on*, pages 25–30. IEEE, 2006.

[148] Haiping Xu and Sol M Shatz. A Framework for Model-Based Design of Agent-Oriented Software. *IEEE Trans. Softw. Eng.*, 29 :15–30, January 2003. ISSN 0098-5589. doi : 10.1109/TSE.2003.1166586. URL http://portal.acm.org/citation.cfm?id=642965.642967.

[149] Jhun-Ying Yang, Jeen-Shing Wang, and Yen-Ping Chen. Using acceleration measurements for activity recognition : An effective learning algorithm for constructing neural classifiers. *Pattern Recognition Letters*, 29(16) :2213–2220, December 2008. ISSN 0167-8655. doi : doi:DOI:10.1016/j.patrec.2008.08.002. URL http://www.sciencedirect.com/science/article/B6V15-4T7083W-1/2/0f4426315ba178a6ebd982dca1cf3ea7.

[150] Juan Ye, Simon Dobson, and S McKeever. Situation identification techniques in pervasive computing : A review. *Pervasive and Mobile Computing*, In Press,, 2011. ISSN 1574-1192. doi : doi:DOI:10.1016/j.pmcj.2011.01.004. URL http://www.sciencedirect.com/science/article/B7MF1-522SHTF-1/2/5f9de7a6e6322a87e365acea94b371bbhttp://www.sciencedirect.com/science/article/pii/S1574119211000253.

[151] Hee Youn, Minkoo Kim, Hiroyuki Morikawa, Shinyoung Lim, and Abdelsalam Helal. Encapsulation and Entity-Based Approach of Interconnection Between Sensor Platform and Middleware of Pervasive Computing. volume 4239 of *Lecture Notes in Computer Science*, pages 500–515–515. Springer Berlin / Heidelberg, 2006. URL http://dx.doi.org/10.1007/11890348_38.

[152] Zhou Yu, Ma Xiao-xing, Cao Jian-nong, Yu Ping, and Lü Jian. Software Agent-Virtualized application mobility in pervasive environments. 2008. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.4939.

[153] Daqing Zhang, Tao Gu, and Xiaohang Wang. Enabling Context-aware Smart Home with Semantic Web Technologies. *International Journal of Humanfriendly Welfare Robotic Systems*, 6(4) :12–20, 2005.