N° attribué par la bibliothèque :

THÈSE

PRÉSENTÉE À

L'Université de Pau et des Pays de l'Adour

ÉCOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS

Par **Christine Louberry**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

KALIMUCHO : Adaptation au Contexte pour la Gestion de la Qualité de Service

---

Soutenue le : 10 Septembre 2010

Après avis des rapporteurs :

Claudia Roncancio	Professeur, LIG, Grenoble
Jean-Marc Pierson	Professeur, IRIT, Toulouse

Devant la commission d'examen composée de :

Isabelle Deumeure	Professeur, ParisTech, Paris	Président
Congduc Pham	Professeur, LIUPPA, Pau	Directeur
Marc Dalmau	Maître de conférences HDR, LIUPPA, Anglet	Encadrant
Philippe Roose	Maître de conférences HDR, LIUPPA, Anglet	Encadrant
David Bromberg	Maître de conférences, Labri, Bordeaux	Examinateur

– 2010 –

Résumé

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 smart-phones, les PDA, les ordinateurs portables ou encore les capteurs.

De plus nous devons faire face aujourd'hui 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, ils disposent de dispositifs pouvant mesurer des grandeurs physiques comme la température, la pression, la vitesse de déplacement ou encore l'humidité. 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 afin que les services rendus par celles-ci fonctionnent bien et pendant une durée suffisante.

Dans cette thèse, nous avons choisi d'aborder l'adaptation dynamique au contexte comme un outil de gestion de la qualité de service. nous présentons une plate-forme pour la reconfiguration et le déploiement contextuel d'applications en environnement contraint appelée Kalimucho, (Kalitate Mucho [1]). Kalimucho est une plate-forme distribuée qui dispose d'une représentation globale de l'application. Elle permet des reconfigurations des applications basées composants grâce à cinq actions de base : ajouter, supprimer, migrer, connecter et déconnecter. L'originalité de cette plate-forme est qu'elle exploite le plus possible les ressources de l'application en permettant d'utiliser tous les périphériques disponibles comme supports des composants logiciels de l'application, qu'ils soient ou non en relation avec les fonctionnalités du périphérique.

Pour cela, elle se base sur une *classification du contexte et une définition de la qualité de service* permettant de prendre en compte les différentes propriétés des applications pervasives : les périphériques, la mobilité, l'environnement ainsi que les spécifications de l'application elle-même. La définition de la qualité de service proposée permet de réunir deux notions fortement correlées dans le cadre des applications pervasives : l'utilité de l'application et la disponibilité des ressources. Chacune de ces définitions est associée à un

1. Qualité en langue basque et Beaucoup en langue espagnole

modèle : *un modèle de contexte* et *un modèle de qualité de service. Le modèle de contexte* est un modèle simple, basé sur une connaissance des composants et des périphériques a priori, mais également sur la collecte d'informations pendant l'exécution. La plate-forme est informée des changements de contexte grâce à des composants spécifiques d'écoute du contexte et génère les actions adéquates. Le choix de l'action à mener est aidé par une *méthode de conception* qui guide le concepteur dans la modélisation du contexte. Elle permet d'identifier tous les évènements pouvant engendrer des reconfigurations et de les associer à une action. Enfin, elle permet de décomposer l'application en différentes configurations qui servent de base au choix de reconfiguration. Puis, lorsque la décision de reconfiguration est prise, *le modèle de qualité de service* permet d'évaluer la qualité de service de l'application à deux niveaux : l'adéquation du service rendu par rapport au service souhaité et la durée de vie de l'application. La particularité de ce modèle de qualité de service est qu'il permet de prendre en compte à la fois la disponibilité des ressources matérielles et celle des ressources réseau. Il est associé à une heuristique de choix de la configuration à déployer qui permet de trouver une configuration, et le déploiement associé, qui respectent les critères de qualité de service. Cette heuristique se base sur deux concepts : le poids d'un composant et le poids d'un périphérique, ainsi que sur un classement des périphériques, afin d'établir un ordre d'évaluation permettant de trouver une solution rapidement.

Nous avons validé cette heuristique par un prototype au travers de tests réalisés dans des contextes de variation des ressources des périphériques, de mobilité des périphériques et de fortes contraintes de débit réseau.

Abstract

The recent technological overhangs have focused on the democratization of wireless networks and the miniaturization of communication devices. Nowadays we can find a lot of devices increasingly lightweight, tiny, mobile and to be equipped of one or several wireless communication medium such as mobile phones, smart-phones, PDA, laptops and sensors.

Moreover, we face with the growing request of rich and customized services. The challenge is to offer applications which suit both the users'needs and the physical environment. This type of mobile devices is able to be aware of its hardware and software environment but also, with the arrival of device such as sensors, they can measure temperature, moisture, move speed or pressure. Integration of such devices in applications can provide services better adapted to their current situation. However, theses devices have features (energy independence, mobility, limited resources) which require adaptations in order to provide services that well function and during a satisfaying time.

In this thesis, we chose to address the dynamic context adaptation as a tool for quality of service management. We present a platform for reconfiguration and contextual deployment of applications in constrained environments called Kalimucho (Kalitate Mucho [2]). Kalimucho is a distributed platform that has a global representation of the application. It adapts component-based applications through five basic actions : add, delete, move, connect and disconnect. The original idea of this platform is that it exploits the resources of the application as better as possible to use all available devices to support components.

For this, it is based on a *context categorization* and a *quality of service definition* which take into account the different properties of pervasive applications : devices, mobility, environment and specifications of the application. The definition of quality of service we provide matches two concepts of pervasive applications : the utility of an application and availability of resources. Each of these definitions is associated with a model : *a context model* and *a QoS model*. *The context model* is a simple model based on a knowledge of components and devices but also, the data-gathering during the execution. The platform is informed of context changes about specific context components and generates the appropriate actions. The choice of the action is guided by a *design method* that assists the designer in modeling the context. It identifies all the events implying reconfiguration and their associated actions. Then, it models all the configurations of an application which aided the decision process. When the decision is taken, *the QoS model* evaluates quality of service of the application on two levels : the adequacy between the service offered and the

2. Quality in Basque and Much in Spanish

needed service and the lifetime of the application. The special feature of this QoS model is that it allows to take into account both the availability of hardware resources and network resources. It includes a heuristic which choose the configuration to deploy according QoS criteria. This heuristic is based on two concepts : the weight of a component and the weight of a device, and a classification of devices, to establish an evaluation order to quickly find a solution.

We implemented a prototype to prove this heuristic. We study the choose process through tests in several contexts : resources changes, mobility and network constraints.

Remerciements

Je tiens tout à d'abord à remercier Marc Dalmau et Philippe Roose, mes encadrants, qui ont été de véritables guides durant ces quatre années de thèse. Je les remercie pour m'avoir donné la chance d'effectuer cette thèse au sein de Laboratoire d'Informatique de l'UPPA, pour leurs précieux conseils tant au niveau de la recheche que de l'enseignement. Je leur adresse toute ma reconnaissance pour toute la confiance qu'ils m'ont accordé, leur investissement et leurs encouragements qui m'ont permis de mener à bien ce projet.

Je tiens également à remercier Congduc Pham pour avoir accepté de diriger cette thèse et pour ses conseils et son suivi durant toutes ces années.

Je remercie particulièrement Claudia Roncancio et Jean-Marc Pierson pour m'avoir fait l'honneur d'accepter d'être les rapporteurs de ce travail. Je n'oublie pas Isabelle Demeure qui a accepté de présider la commission d'examen et ainsi manifesté son intérêt pour ces travaux.

Un grand merci à Sophie Laplace pour son aide et son soutien pendant les bons comme les moins bons moments que j'ai traversé durant cette thèse. J'ai beaucoup apprécié de partager nos idées sur ce projet et j'espère que nous continuerons à travailler ensemble ces prochaines années.

J'adresse ma gratitude à tous mes collègues de l'IUT de Bayonne pour avoir fait de l'IUT un lieu de convivialité et de partage : Patrick, Pierre, Yon, Christophe, Gilles, Jean-Marc, Léo, Kevin, Lopis, Thierry pour avoir goûter avec le sourire à mes gâteaux et autres patisseries pas toujours réussies, Ginette, Ghislaine, Valérie, Pantxika pour leur bonne humeur, Simone, Peio, Corine, Agnes, Jean-Michel, Bruno, Laurent, Estelle, Benoit et les autres. Une pensée pour ceux qui sont passés à l'IUT : Jérome et Denis pour leurs conseils de rédaction et Cyril pour sa contribution technique sur le projet sans oublier nos parties de Yam's du midi.

Merci à tous mes camarades doctorants Eric, Natacha, Damien et particulièrement Julien et Cyril pour avoir partagé les mêmes difficultés, les mêmes motivations et pour leur soutien.

Merci à Christophe et Benjamin que j'ai eu l'occasion de rencontrer lors de déplacements en conférence.

Merci à mes amis, Seb, Séverine, Julien, Guillaume, Laurie, Yves, Aurel, Christophe pour leur joie de vivre à toutes épreuves, pour les soirées bayonnaises improvisées et les week-end toulousains, Sébastien, Mathieu, Guillaume pour les virées shopping, rock et roller, tous ceux que je n'ai pas cité.

Enfin, mes remerciements vont à ma famille, mes parents, mon frère, mes cousines, pour avoir cru en moi et pour m'avoir soutenu tout au long de ces années.

Merci à vous tous.

Table des matières

Table des figures

Liste des tableaux

Listings

Chapitre 1

Introduction

Sommaire

Depuis une quinzaine d'années, nous assistons à une évolution considérable des technologies et des habitudes d'utilisation de l'informatique. La démocratisation des moyens de communication sans-fil couplée aux avancées technologiques matérielles permettent à présent de proposer des dispositifs portables qui tiennent dans la main, dotés d'une capacité de calcul et de moyens de communication sans-fil tels que les mini PC, les PDA, les téléphones portables et même les capteurs sans-fil. L'informatique devient alors disponible partout : c'est l'informatique diffuse.

Ces constats convergent vers la vision de l'informatique du futur que donnait Mark Weiser en 1991 dans son article intitulé "The Computer for the 21st Century" [84] :

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

Les systèmes pervasifs désignent désormais l'informatique nouvelle où l'environnement est naturellement doté de capacités de traitement et de communication, intégrées de façon à devenir invisibles et omniprésents. La finalité de tels systèmes tournés vers l'utilisateur est de fournir un environnement personnalisé qui facilite l'accès aux services et améliore leur utilisation de façon transparente.

La miniaturisation des périphériques et leur mobilité en font des objets personnels à part entière. Les utilisateurs montrent alors un intérêt particulier à pouvoir accéder à une multitude de services et de fonctionnalités depuis n'importe où et n'importe quel dispositif. Ils sont également de plus en plus sensibles à la personnalisation des applications qu'ils utilisent sur leur périphérique comme par exemple l'adaptation de l'interface graphique en fonction des capacités du périphérique, l'adaptation de l'application en fonction de leurs préférences comme la langue ou bien l'adaptation en fonction de leurs déplacements ou de leur environnement physique (luminosité, température, etc).

Cette personnalisation des applications nécessite une sensiblité au contexte telle que

la définissent Schilit et Theimer dès 1994 [67] : *la sensibilité au contexte est la capacité d'une application et/ou d'un utilisateur mobile, de découvrir et réagir aux changements de sa situation*".

Enfin la sensibilité des utilisateurs quant à la perception du service rendu réfère à la qualité de service de l'application telle que la décrit [42] : *l'adéquation entre le service souhaité par l'utilisateur et le service qui lui est fourni.*

Ces deux notions sont fortement corrélées. En effet, afin de pouvoir garder cette adéquation entre le service souhaité et le service fourni malgré les changements de l'environnement, il est nécessaire d'adapter les services. La sensibilité au contexte va permettre de fournir les informations nécessaires pour adapter les services et permetre la continuité de son exécution. Conformément à la finalité des systèmes pervasifs, cette adaptation doit rester transparente à l'utilisateur.

1.1 Problématique

L'objectif est de répondre aux besoins des utilisateurs et aux changements de l'environnement par des adaptations des services avec pour principal critère la qualité de service de l'application. Nous nous attachons particulièrement à la gestion de la qualité de service des applications distribuées face aux contraintes matérielles de leur support, aux exigences de l'utilisateur et aux circonstances courantes d'utilisation. En effet, bien que les caractéristiques proposées par ces périphériques tendent à améliorer la qualité du service rendu à l'utilisateur, leur intégration dans les systèmes traditionnels implique de relever plusieurs défis :

Limitation des ressources Le fonctionnement des systèmes pervasifs réside en partie sur la collaboration de dispositifs petits et mobiles. Cependant l'atout que présente la petite taille et la mobilité de ces dispositifs est également une contrainte. En effet, la mobilité implique une autonomie énergétique. Or, des études ont montré que les transmissions sans-fil ont une consommation énergétique 10 à 100 fois plus élevée qu'un calcul [2]. L'autonomie énergétique est une contrainte critique à prendre en compte dans le développement des applications. En effet, lorsqu'un dispositif n'a plus d'énergie, tous les services qu'il hébergeait deviennent inutilisables et l'application ou la partie d'application qui en dépendait ne peut plus fonctionner. En terme de qualité de service, cette situation doit être évitée ou anticipée. De plus, la petite taille des dispositifs implique des capacités de calcul et des capacités de mémoire réduites qui doivent également être prises en compte.

Le développement des systèmes devra prendre en compte les capacités restreintes des périphériques notamment lors du déploiement.

Hétérogénéité Bien qu'offrant des propriétés similaires, les dispositifs que nous rencontrons dans les systèmes pervasifs tels que les ordinateurs portables, les téléphones, les PDA, les capteurs, sont différents à tous niveaux : matériel, logiciel et communication.

Le développement des systèmes devra proposer des solutions d'interopérabilité.

Mobilité La mobilité est un atout pour les systèmes pervasifs mais elle soulève néanmoins des problèmes. Le principe des systèmes pervasifs réside sur la collaboration des services et donc des périphériques qui les herbergent. La mobilité des périphériques entraine des variations de connexion qui peuvent avoir de lourdes conséquences sur le fonctionnement des services : faible débit qui entraine la lenteur des transmissions et parfois la déconnexion. Tout comme l'épuisement d'une batterie, les déconnexions provoquent, au moins de manière temporaire, l'indisponibilité des services et par conséquent dégradent la qualité de service de l'application.

Le développement des systèmes devra proposer des solutions pour la gestion de la mobilité et la continuité des services de l'application.

Variation du contexte L'utilisateur et son périphérique évoluent dans un environnement sans cesse changeant. L'utilisateur modifie ses préférences, les ressources du périphérique évoluent tout comme l'environnement physique. Le fonctionnement des applications peut être perturbé par des modifications du contexte. Par exemple, dans une application de visio-conférence, la luminosité et le bruit ambiant peuvent perturber son fonctionnement et se répercuter sur la satisfaction de l'utilisateur et sur l'utilisabilité de l'application, c'est-à-dire modifier la qualité de service perçue par ce dernier.

Il faut alors pouvoir prendre en compte ces évolutions du contexte pour pouvoir agir de manière à maintenir la qualité du service rendu. Cette contrainte est à l'origine de la sensibilité au contexte.

Afin de faire face à ces problèmes et à la complexité des applications, la réponse la plus répandue est la virtualisation. [17] fait un état de l'art des approches par virtualisation et en donne la définition suivante : « *Les approches par virtualisation consistent donc à créer des environnements de développement et d'exécution des applications. Il est alors possible de concevoir les applications en manipulant des objets complexes créés par des couches logicielles de virtualisation (APIs, langages, bibliothèques, services de l'intergiciel). De même ces applications s'exécutent sur des plates-formes qui cachent les supports sous-jacents (systèmes d'exploitation, machines virtuelles, serveurs d'applications).* ». Concernant les dispositifs contraints, [17] soulève la question suivante : Comment continuer à répondre à la complexité croissante des logiciels et développer des applications sur des périphériques contraints tout en proposant des solutions répondant aux demandes des utilisateurs ? Les solutions existantes dans la littérature proposent des plate-formes d'exécution telle que J2ME. Ces plates-formes, outre qu'elles permettent de cacher l'hétérogénéité des matériels, facilitent le dialogue entre ces périphériques et des applications plus traditionnelles.

De plus, bien que de nombreux travaux utilisent le contexte pour l'adaptation des applications, il n'existe aucun consensus autour de sa définition. En effet, chaque application ayant un objectif spécifique, les informations auxquelles elle va réagir sont également spécifiques. C'est pourquoi nous devons proposer une classification des informations contextuelles spécifiques à notre objectif de qualité de service. Il en est de même pour la définition de la qualité de service puisque celle-ci n'utilisent pas les mêmes critères selon le type d'application visé. Tout comme pour le contexte, nous devons proposer une définition de la qualité de service spécifiques aux applications distribuées en environnement contraint.

Dans ce chapitre nous présentons les objectifs de cette thèse que nous illustrons à l'aide d'un exemple : une application de visite d'un musée. Puis nous présentons ce qu'est le contexte accompagné d'une classification permettant la prise en compte des informations contextuelles à trois niveaux : utilisateur, application, infrastructure. Nous présentons ensuite la définition de la qualité de service que nous avons retenu pour ces travaux. Enfin, nous présentons un état de l'art des approches proposées pour l'adaptation dynamique des applications utilisant des plate-formes de reconfiguration.

1.2 Exemple d'application

Tout au long de ce mémoire, nous allons illustrer notre approche par un exemple d'application : la visite d'un musée. Elle s'adresse à deux types d'utilisateurs : d'une part les visiteurs du musée et d'autre part le personnel en charge du musée. Cette application est destinée à être utilisée avec différents appareils mobiles tels que des PDA, des Mobile PC ou des smart-phones. Tout d'abord rescensons les besoins de chaque utilisateur.

Lorsqu'un visiteur est dans le musée, il souhaite, pour chacune des œuvres présentées, pouvoir disposer d'informations sur sa création, son auteur, son histoire, etc. Il souhaite également pouvoir visiter librement, avec ou sans l'aide d'un plan du musée, ou au contraire être guidé de son entrée jusqu'à la sortie du musée pour ne pas perdre de temps et ne pas rater les œuvres incontournables.

Le personnel veut connaître en temps réel le nombre de visiteurs à l'intérieur du musée mais également disposer de statistiques sur le nombre de personnes ayant visité le musée dans la journée ou le mois afin d'évaluer les taux de fréquentation. Il souhaite également connaître le nombre de visiteurs qui ont demandé des informations sur chaque oeuvre afin de mettre à jour les parcours de visites guidées par exemple. Enfin il souhaite pouvoir guider les visiteurs vers la sortie lors de la fermeture du musée ou lors d'un incident.

Afin de répondre à l'ensemble des besoins énoncés précédemment, nous proposons que cette application de visite de musée fournisse trois services :

- un service de description. Ce service fournit à la demande du visiteur des informations sur l'œuvre devant laquelle il est arrêté.
- un service de guidage. Ce service propose deux versions. La première est un service de guidage simple qui fournit un plan du musée à l'utilisateur. La deuxième est un service de visite guidée qui fournit un parcours correspondant à une durée donnée et à une thématique choisie par l'utilisateur.
- un service de statistiques. Ce service permet de recueillir et traiter les données relatives à l'utilisation du musée.

Les services ne sont pas tous visibles par tous les utilisateurs. Les visiteurs ont uniquement accès au service de description et au service de guidage alors que le personnel a accès au service de statistiques ainsi qu'au service de guidage.

L'objectif est que chaque service s'adapte à la fois aux souhaits de l'utilisateur, aux capacités du périphérique qu'il utilise ainsi qu'aux évènements extérieurs qui peuvent influer sur le rendu final des services comme par exemple :

- lorqu'un utilisateur demande un service, celui-ci doit lui être fourni en fonction des

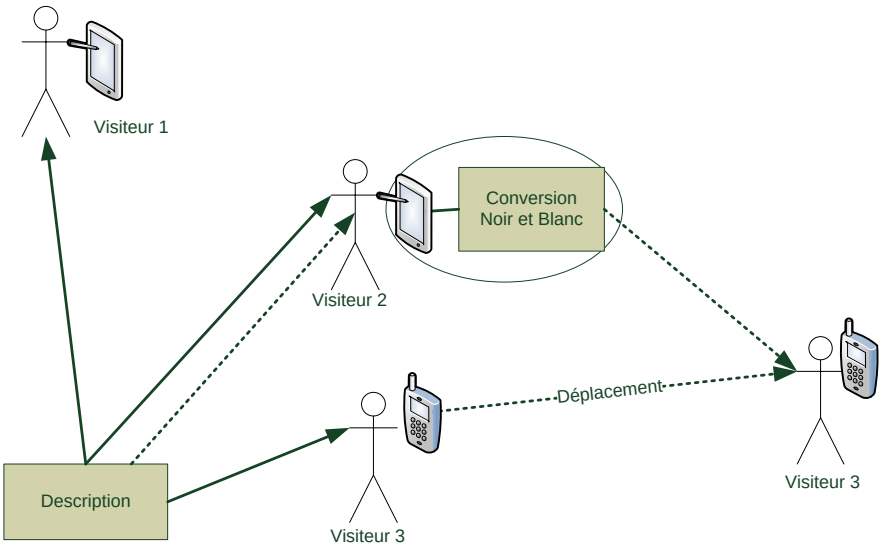

Figure 1.1 – Scenario d'utilisation de l'application de visite d'un musée

capacités du périphérique (vidéo, audio, texte), et des ressources physiques disponibles (énergie, mémoire).
- lorsque les ressources physiques d'un périphérique atteignent un seuil critique, il faut pouvoir évaluer la qualité du service rendu et proposer une nouvelle solution.
- lorsqu'un évènement survient comme une demande de la part de l'utilisateur, la fermeture du musée ou encore une conférence, il faut pouvoir le traiter afin d'adapter les services en conséquence et de proposer une application qui fonctionne le mieux possible.
- enfin lorsque l'utilisateur se déplace, il faut pouvoir assurer la continuité du service tout en respectant les souhaits de l'utilisateur et les limitations de ressources du périphérique.

Décrivons un scénario possible d'utilisation de l'application afin d'illustrer les adaptations que nous souhaitons traiter dans cette thèse. Trois visiteurs sont dans le musée et utilisent le service de description. La figure 1.1 décrit la situation spaciale des visiteurs dans le musée.

Parmi les différents choix pour le service de description, nous estimons que dans le contexte de l'application de visite d'un musée, la meilleure qualité est de fournir le service sous sa forme vidéo en couleurs. Lorsque le visiteur 3 se déplace dans le musée, trois cas de figure se présentent. Premièrement, bien qu'il se soit déplacé, le périphérique du visiteur 3 est toujours à portée du serveur de vidéo et le débit permet toujours de transmettre une vidéo en couleur. Deuxièmement, le débit est faible mais le périphérique est toujours à portée. Afin d'assurer la continuité du service, une solution est de réduire le nombre de

données à transmettre. Réduire le nombre de données consiste à effectuer des traitements sur ces données afin de transmettre moins, ou moins souvent. Une solution est d'effectuer un traitement sur une vidéo telle qu'une conversion en noir et blanc, une réduction du nombre de couleurs ou encore une réduction de la résolution des images. Ceci implique de proposer une nouvelle structure incluant un traitement supplémentaire. Dans le cas de la transmission de la vidéo, nous pouvons ajuster un traitement de conversion en noir et blanc sur le serveur. Le serveur ne transmet plus de couleurs au périphérique 3 mais du noir et blanc. Troisièmement, le périphérique n'est plus à portée du serveur, nous devons donc chercher une nouvelle route afin d'atteindre le périphérique 3 et d'assurer la continuité du service. Une solution peut être d'utiliser le périphérique du visiteur 2, qui utilise également le service de description, afin de lui faire jouer le rôle de relais pour le périphérique 3.

Découvrir une nouvelle route pour transmettre une information est un problème classique que les protocoles réseau savent résoudre. Il existe des protocoles réseau dédiés aux périphériques contraints. Un exemple est le protocole AODV de [60]. AODV propose des solutions de routage rapides en réponse à la mobilité des périphériques et à leurs faibles ressources de traitement et de mémoire. Toutefois, nous souhaitons proposer des solutions différentes plus intéressantes et non uniquement basées sur des critères techniques de faisabilité. En effet puisque la vidéo en couleurs est d'ores et déjà diffusée aux utilisateurs 1 et 2, il est tout à fait envisageable que l'un de ces deux utilisateurs puisse assurer un rôle de relais vers l'utilisateur 3 qui n'est plus directement à portée du service vidéo. Toutefois si l'énergie disponible sur le périphérique servant de relais est trop faible, la diffusion d'une vidéo en couleurs n'est pas envisageable. Il est alors possible de proposer l'installation un composant de conversion couleur/noir et blanc sur le périphérique servant de relais afin qu'il ne diffuse qu'une version en noir et blanc compatible avec l'énergie disponible. Ainsi, par exemple, dans le souci d'assurer la continuité du service, avant de retransmettre la vidéo au périphérique 3, une conversion en noir et blanc est effectuée au niveau du périphérique 2. En effet, il faut préciser que pour de tels périphériques, les transmissions de données sont démesurément plus coûteuses en énergie que le calcul. Selon le périphérique, une transmission coûte entre dix à cent fois plus qu'un calcul de même durée. De même, plus les périphériques communicants sont distants l'un de l'autre, plus ils consommeront d'énergie afin de maintenir un signal réseau assez puissant pour transmettre des données. La consommation d'énergie est fonction du nombre de données transmises ainsi que de la distance avec le récepteur. Afin de limiter la consommation d'énergie et par conséquent d'allonger la durée de vie de l'application, il est généralement préférable de traiter les données pour en transmettre moins. Ainsi la conversion de la vidéo en noir et blanc permet de réduire le nombre de données à transmettre au périphérique 3 et donc de préserver son énergie ainsi que celle de l'émetteur. Délocaliser une partie des services permet un équilibrage des charges au niveau du réseau et des périphériques. Rapprocher un traitement de la source peut éviter des transmissions ce qui, comme nous l'avons mentionné, permet de préserver l'énergie des périphériques. Lorsque le réseau sature, que la bande passante est faible à la sortie d'un périphérique, il est préférable de délocaliser un service vers un périphérique où la bande passante est plus élevée. Enfin, en raison de leurs faibles ressources, les périphériques peuvent voir leur mémoire ou leur capacité de traitement saturées. Il faut alors les soulager. Pour cela, nous pouvons choisir de déplacer un ou plusieurs services vers d'autres périphériques plus disponibles. On peut

remarquer qu'une telle solution fait appel à une connaissance de l'application en termes de services que les protocoles réseau ne peuvent pas avoir. Dans une situation identique, les protocoles réseaux comme AODV [60] n'auraient pu proposer que des solutions permettant de continuer à transmettre la vidéo en couleurs à l'utilisateur 3. Ceci aurait été fait au détriment de la pérennité de l'application puisque cette solution aurait abouti à un épuisement rapide de la batterie du périphérique assurant le relais. Par sa connaissance des services demandés par les utilisateurs, de ceux actuellement disponibles et surtout par la possibilité qu'elle offre de substituer à un service défaillant un service de qualité différente mais de même rôle, notre approche par reconfiguration permet de proposer des solutions viables du point de vue de l'infrastructure et acceptables du point de vue de la qualité de service.

1.3 Objectif

Le principal objectif de cette thèse se focalise autour de la qualité de service des applications. Les applications que nous visons sont soumises à trois contraintes.

D'une part elles s'exécutent sur des périphériques contraints, c'est-à-dire dont les ressources sont limitées et particulièrement l'énergie. Lorsqu'un périphérique cesse de fonctionner, tous les services dont il est le support d'exécution cessent de fonctionner à leur tour. L'application subit alors une rupture de service. L'un des objectifs de cette thèse est d'éviter ce type de situation et de proposer des solutions qui permettent aux périphériques de fonctionner le plus longtemps possible afin de pérenniser l'exécution de l'application. Il s'agit également d'anticiper certaines situations comme la fin de vie de la batterie d'un périphérique. La préoccupation de la consommation d'énergie est une problématique abordée jusque là en termes de protocoles de routage essentiellement. Nous souhaitons y répondre en intégrant les préoccupations de QdS de l'informatique ambiante. Réduire la consommation d'énergie d'un périphérique par l'application peut se faire en répartissant la charge de l'application sur les périphériques voisins. Dans ce cas, il faut pouvoir proposer des solutions telles que la délocalisation de services en cours d'exécution afin d'anticiper les déconnexions dues au déchargement complet de la batterie.

D'autre part le contexte dans lequel elles évoluent varie constamment au cours du temps. Comme nous l'avons décrit dans l'introduction, une application qui fonctionne bien à un instant t peut ne plus fonctionner ou mal fonctionner à l'instant t+1. A chaque modification du contexte, l'objectif est d'assurer que l'application puisse fonctionner le mieux possible. Dans le scénario que nous avons décrit (figure 1.1), un utilisateur souhaite utiliser le service de description. Dans un premier temps, le service lui est proposé sous sa forme vidéo en couleurs. Cependant, au cours de l'exécution, le niveau d'énergie du périphérique a fortement diminué. Continuer d'utiliser la forme vidéo en couleurs devient critique à ce niveau de batterie, le service n'est plus adapté aux ressources du périphériques. Une solution est de modifier le service afin, par exemple, qu'il fournisse une vidéo en noir et blanc ou un nombre d'images par seconde réduit.

Enfin, nous devons également gérer la mobilité des périphériques. Comme nous l'avons décrit dans le scénario, des situations de mobilité peuvent mettre en péril la continuité de l'application d'une part et la durée de vie des périphériques d'autre part. L'objectif est de

pouvoir détecter la mobilité et de réagir en conséquence.

Pour répondre à ces objectifs nous proposons de raisonner en termes de qualité de service. Le but est de fournir à l'utilisateur une application de la meilleure qualité possible. Ceci implique de raisonner dans la globalité car une amélioration partielle en un point de l'application peut se révéler catastrophique en un autre point.

Si nous résumons, l'objectif de nos travaux est de faire en sorte que les applications fonctionnent le mieux possible et le plus longtemps possible quels que soient les changements de contexte. Le principe est de reconfigurer l'application lorsque la qualité du service en cours d'exécution n'est plus adaptée aux exigences des différents acteurs, des utilisateurs, du matériel et de l'application elle-même. Pour cela, nous avons besoin d'avoir une vue globale du fonctionnement de l'application ainsi qu'une grande flexbilité.

C'est pourquoi nous avons choisi d'utiliser des applications composées qui, grâce à leur modularité, offrent un grand choix de solution pour garantir une application continue et de qualité et de les superviser par une plate-forme. Cette plate-forme doit fournir des mécanismes permettant de fournir des applications offrant la meilleure QdS en fonction du contexte :

1. en proposant une classification du contexte et une définition de la qualité de service associée à des modèles d'évaluation permettant de refléter l'influence de la totalité des participants, utilisateur, environnement, composants, matériel et réseau, sur la qualité de service de l'application.
2. en proposant d'accroître les possibilités de garantir la meilleure QdS en considérant l'ensemble des périphériques disponibles comme des supports potentiels de déploiement de l'application.
3. le tout supppose de prendre en compte les propriétés des composants et des connexions dans le choix de déploiement.

Pour pouvoir s'adapter dynamiquement, l'application doit avoir une connaissance d'elle-même (réflexivité) afin de pouvoir substituer un service pratiquement équivalent à un service défectueux ou inadapté au contexte. **Dans ce but nous avons choisi d'utiliser une plate-forme d'exécution qui connait l'application en cours et son contexte**. Cette plate-forme peut alors prendre des décisions de reconfiguration dynamique en toutes connaissances de cause. Elle assure ainsi la continuité des services tout en prenant en compte la pérennité globale de l'application. Une telle solution permet au concepteur de définir un ensemble de services dont certains sont substituables et à l'utilisateur de disposer de ces services selon ses besoins. Les problèmes non fontionnels induits par la mobilité, l'énergie et, de façon plus générale, le contexte, sont alors pris en compte par la plate-forme qui modifie, remplace et déplace des services dans une optique globale de QdS de l'application.

Nous devons tout d'abord capturer tous les changements de contexte qu'ils soient liés à l'utilisateur, aux ressources ou à la mobilité. Ensuite nous devons interpréter ces changements afin de réagir de la meilleure façon. Nous proposons d'établir un modèle de contexte permettant de le décrire, le capturer et l'interpréter.

C'est pourquoi nous devons prévoir la prise en compte de l'évolution du contexte le plus tôt possible dans la conception de l'application. La section suivante propose un état de l'art des définitions du contexte et propose une classification de ce dernier permettant

d'inclure un large spectre d'informations auquel les applications doivent être adaptées.

1.4 Conclusion

Face aux défis qu'engendre le problème de l'adaptation des applications pervasives, nous proposons Kalimucho, Kalitate [1] Mucho [2], une plateforme de reconfiguration et de déploiement contextuel et dynamique d'applications basées composants. Kalimucho prend en charge la reconfiguration et le déploiement des applications sur les différents périphériques en les adaptant aux contraintes. L'originalité de cette plateforme est qu'elle exploite le plus possible les ressources de l'application en permettant d'utiliser tous les périphériques disponibles, y compris les capteurs, comme support des composants logiciels de l'application. Les contributions de ses travaux résident en ces éléments :

1. *Une classification du contexte et une définition de la qualité de service* permettant de prendre en compte les différentes propriétés des applications pervasives. La classification du contexte proposée permet d'inclure un large spectre des informations disponibles dans l'environnement de l'application. Elle permet d'introduire la prise en compte des spécifications de l'application elle-même alors que les définitions les plus courantes sont centrées sur une entité comme l'utilisateur ou le périphérique. La définition de la qualité de service proposée permet de réunir deux notions fortement correlées dans le cadre des applications pervasives : l'utilité de l'application et la disponibilité des ressources.
2. *Un modèle de contexte* simple basé sur une connaissance des composants et des périphériques a priori mais également sur la collecte d'information pendant l'exécution. La plateforme est informée des changements de contexte grâce à un mécanisme constitué de composants spécifiques d'écoute du contexte et génère les actions adéquates. Le modèle de context est associé à une *méthode de conception* permettant d'aider le concepteur de l'application dans sa démarche de modélisation du contexte. Cette méthode permet d'identifier les spécifications de l'application, des composants et des périphériques. Elle permet également d'identifier tous les évènements pouvant engendrer des reconfigurations. Enfin, elle permet de décomposer l'application en différentes configurations qui servent de base au choix de reconfiguration.
3. *Un modèle de qualité de service* associé à des heuristiques d'évaluation de déploiement qui permettent d'évaluer la qualité de service d'une application à deux niveaux : l'adéquation du service rendu par rapport au service souhaité et la durée de vie de l'application. La particularité de ce modèle de qualité de service est qu'il permet de prendre en compte à la fois la disponibilité des ressources matérielles et la disponibilité des ressources réseaux. Le but de ce modèle est de fournir une application qui propose le meilleur compromis entre une application utile et une durée de vie la plus longue possible.
4. *Une architecture logicielle distribuée* de la plateforme Kalimucho qui permet d'avoir une représentation globale de l'application. La distribution de la plateforme prend en compte les contraintes des périphériques. Les composants et les connecteurs des

1. qualité en langue basque
2. plus en langue espagnole

applications sont construits selon un même modèle permettant une vue et une manipulation unifiée des applications grâce à des trois actions : ajouter, supprimer et migrer.

Le manuscrit est organisé de la façon suivante :

Le **chapitre 2** présente un état de l'art des défis à relever dans le domaine de la sensibilité au contexte et des périphériques contraints et mobiles. Il inclut une classification du contexte et une définition de la qualité de service permettant de conclure que la gestion de la QdS d'une application peut se faire par une adaptation au contexte. Nous étudions les différentes approches proposées pour répondre à ces problèmes tels que les intergiciels de reconfiguration, les intergiciels de déploiement contextuel et les architectures logicielles pour la gestion de la qualité de service. Nous étudions également les différents mécanismes proposés pour la gestion du contexte. Nous concluons ce chapitre par une classification des approches étudiées permettant de mettre en exergue leurs points communs et leurs différences. Nous nous basons sur cette classification pour donner les propriétés que doit offrir la plateforme Kalimucho.

Le **chapitre 3** présente le modèle de contexte retenu et le modèle de qualité de service pour l'évaluation des applications. Ce chapitre détaille les deux fonctions d'évaluation de la qualité de service : la première étudie le coût de déploiement matériel d'une configuration sur les périphériques et la seconde étudie le coût de déploiement réseau d'une configuration. Enfin, nous détaillons la méthode de conception associée au modèle de contexte permettant au concepteur d'identifier tous les évènements de reconfiguration et de produire les cartes d'identité des composants et des périphériques.

Le **chapitre 4** présente l'architecture de Kalimucho avec ses modèles de composants et de connecteurs et ses services de supervision. Dans ce chapitre nous détaillons les différents moyens d'action dont dispose la plateforme pour réaliser les reconfigurations. Nous détaillons également l'heuristique de choix de reconfiguration et du déploiement associé aux évènements reçus par Kalimucho.

Le **chapitre 5** propose un prototype implémenté dans le cadre de ces travaux afin de valider le fonctionnement de l'heuristique inclut dans Kalimucho. Nous présentons le fonctionnement le l'heuristique de choix de reconfiguration face à différents évènements de reconfiguration tel que l'évolution des ressources et la mobilité d'un périphérique et face à un environnement très contraint.

Enfin, nous concluons par une synthèse des travaux présentés et nous donnons les percpectives et les directions de recherches possibles à ce travail.

Chapitre 2

Adaptation dynamique et Qualité de Service

Sommaire

2.1 Introduction

Les solutions les plus répandues pour résoudre le problème de l'adaptation dynamique au contexte et à celui de la gestion de la qualité de service, sont basées sur des plate-formes d'exécution (section 1.1). De la même façon, nous proposons d'adapter les applications par des reconfigurations. Dans la section suivante, nous présentons un état de l'art des approches relevant les défis des environnements contraints, de la gestion de la qualité de service et de la gestion du contexte.

2.2 Contexte

Il existe de multiples définitions du contexte. Elles sont pour la plupart une énumération de paramètres qui les rendent très abstraites, difficile à exploiter ou au contraire très spécifiques à un domaine. A partir de plusieurs définitions que nous trouvons dans la littérature, nous allons exposer notre propre classification du contexte. Plusieurs tentatives de définition du contexte ont été menées dans plusieurs domaines ces dernières années et plus particulièrement en informatique ubiquitaire et pervasive avec l'émergence des systèmes sensibles au contexte. [22] fait un parallèle intéressant avec le discours humain :

> *Quand les humains parlent avec d'autres humains, ils sont capables d'utiliser des informations implicites de la situation, ou contexte, pour augmenter la bande passante de la conversation [22].*

En effet, pour enrichir la conversation et améliorer notre discours, nous y intégrons des informations sur la situation courante à l'instant où nous parlons. De la même façon, une application pourrait intégrer des informations sur sa situation afin d'enrichir, adapter et améliorer le service fourni. Les personnes à l'origine du terme « sensible au contexte »(context-awareness) sont Schilit et Theimer [67] qu'ils définissent comme « la capacité d'une application et/ou d'un utilisateur mobile de découvrir et réagir aux changements de sa situation ». De très nombreuses autres définitions ont été données [58] [22] mais aucune ne fait pour l'instant autorité.

Depuis quelques années, les avancées sur la mobilité des périphériques et sur la technologie sans-fil ont ouvert de nouvelles perspectives dans divers domaines de recherche liés à la communication. Cette évolution de l'utilisation de l'informatique a mis en évidence, pour les applications ainsi distribuées, le besoin d'informations supplémentaires à celles habituellement nécessaires aux traitements. Alors que traditionnellement, les applications se contentaient de produire de nouvelles données en sortie à partir de données en entrée, aujourd'hui les traitements peuvent dépendre également de la situation géographique de l'utilisateur, de ses souhaits ou encore des données physiques telles que la température extérieure ou le taux de luminosité. Ces données annexes font partie d'un ensemble appelé

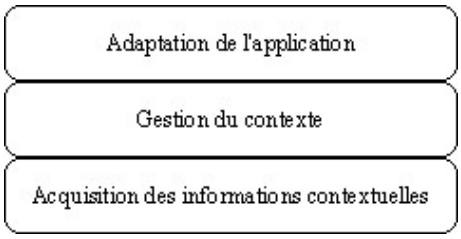

Figure 2.1 – Architecture en couche des applications sensibles au contexte [74]

données contextuelles ou contexte de l'application. [19] donnent une définition informelle mais assez représentative :

> *Le contexte d'exécution d'une application regroupe toutes les entités et situations externes qui influent sur la Qualité de service/Performances (qualitative et quantitative) telle que perçue par les utilisateurs.*

L'utilisateur n'est pas le seul à percevoir l'influence du contexte sur le service rendu, nous pouvons étendre cette définition au système si nous lui donnons les moyens de prendre conscience de ces influences. Ces définitions montrent qu'à présent les applications doivent intégrer des mécanismes d'acquisition du contexte permettant de le prendre en compte afin de s'adapter par des reconfigurations pour répondre au mieux aux besoins du service rendu. Cependant le contexte regroupe une multitude de données que nous devons organiser afin d'évaluer comme il se doit les besoins d'adaptation.

Le principe architectural de la figure 2.1 permettant la prise en compte du contexte dans les applications est assez classique. On en trouvera un exemple dans [74]. Il se « résume »à une superposition de couches correspondant à l'acquisition des informations contextuelles, la gestion du contexte puis l'application (et/ou son adaptation).

Ces informations contextuelles proviennent le plus souvent de différentes sources. Tout comme [12] nous distinguons les informations de contexte des informations de l'application. En aucun cas une information contextuelle ne peut être fournie en entrée d'un service de l'application. L'application ne traite pas les informations contextuelles. Ces informations sont recueillies et utilisées par la plate-forme afin d'adapter le comportement et la structure des services. [74] va plus loin en distinguant plusieurs types d'informations contextuelles.

Chacune des informations de contexte a un impact différent sur la QdS et donc sur l'adaptation à mener. Nous distinguons les informations reçues de l'utilisateur, de l'environnement, des périphériques et du réseau.

2.2.1 Contexte d'utilisateur

Ce sont toutes les informations qui concernent ce que l'utilisateur veut pouvoir faire avec l'application. L'utilisateur est très souvent intégré dans le contexte puisqu'il est, la plupart du temps, le seul à percevoir le rendu final de l'application.

Une application destinée à un utilisateur doit lui fournir des services. L'utilisateur veut pouvoir émettre des souhaits quant à l'utilisation de l'application. L'application pourra

alors être modifiée en conséquence afin de lui fournir un service le mieux adapté possible.

Nous définissons le contexte comme étant l'ensemble des entités caractérisant les souhaits de l'utilisateur comme par exemple :

- la demande d'un service. Par exemple, dans l'application de visite du musée, un visiteur demande à pouvoir utiliser le service de guidage.
- l'arrêt d'un service. Dans notre exemple de visite du musée, un visiteur ne souhaite plus utiliser le service de guidage.
- la demande d'une adaptation de service. Un visiteur utilise le service de description. Cependant la langue dans laquelle la description est donnée ne lui convient pas, il demande alors une traduction dans la langue de son choix.

2.2.2 Contexte d'utilisation

Alors que pour l'utilisateur nous identifions ce qu'il souhaite pouvoir faire avec l'application, nous nous intéressons ici aux spécifications de l'application.

Nous entendons par spécifications tout ce qu'elle doit permettre et ce qu'elle ne doit pas permettre dans son utilisation. Ce sont toutes les situations où ni l'utilisateur, ni son appareil n'interviennent mais qui demandent tout de même une adaptation comme par exemple quand une alarme se déclenche, quand un niveau sonore est trop élevé, etc.

Il s'agit de modifications de l'environnement qui entrainent, à chaque occurence, la même adaptation. Chaque fois qu'une situation ainsi décrite est rencontrée, il faudra appliquer une règle bien définie.

Nous définissons le *contexte d'utilisation* comme l'ensemble des entités intervenant dans les prérequis de toutes les obligations et restrictions sur le fonctionnement des services proposés par l'application. Ces entités ne font en aucun cas partie d'un autre contexte. Elles sont indépendantes de l'utilisateur et des périphériques.

2.2.3 Contexte d'exécution

Dans la plupart des travaux sur la prise en compte du contexte, c'est le *contexte d'exécution* qui est le plus souvent cité. Comme le dit [74], lorsqu'on parle de *contexte d'exécution*, il s'agit très souvent d'adapter l'application à l'évolution de l'état des ressources du système sur lequel elle s'exécute. Dans ces ressources nous incluons toutes les propriétés liées au réseau. Tout comme pour des propriétés matérielles comme la mémoire ou l'énergie, nous pouvons mesurer le débit d'un composant ou d'un périphérique et nous pouvons détecter la mobilité. Prenons l'exemple de deux composants A et B reliés entre eux par la relation suivante : A produit des données consommées par B. Lorsqu'il n'y a aucun problème réseau, A produit régulièrement des données que B consomme de façon régulière également. A un instant donné, B ne reçoit plus de données de la part de A, il détecte une pénurie dont il va informer la plate-forme. S'il y a pénurie cela veut dire que le composant A s'est déplacé ou qu'il a cessé de fonctionner. La plate-forme va alors chercher à joindre le composant A et en cas de réponse, proposer une solution afin de continuer le service. De la même manière, A détecte une saturation de données à sa sortie, B ne les

consomme plus. Il en informe la plate-forme qui va engager un processus de reconfiguration. Ainsi la mobilité fait partie des caractéristiques du *contexte d'exécution* au même titre que les caractéristiques matérielles et logicielles.

Nous définissons le *contexte d'exécution* comme l'ensemble des propriétés permettant de caractériser le support de l'application c'est à dire l'infrastructure du système qui supporte les composants logiciels, les périphériques et le réseau. Ce sont toutes les propriétés mesurables et mathématiquement comparables relatives aux ressources physiques du système.

2.3 Gestion de la qualité de service

L'objectif est de proposer aux utilisateurs des applications qui fonctionnent le mieux possible et le plus longtemps possible sur leur appareil mobile favori et ce quelles que soient les évolutions du contexte.

Ce type d'application concerne un large public, par conséquent il nécessite une production à moindre coût. Une solution est d'utiliser des composants logiciels qui, de par leur particularité de réutilisation, permettent de réduire à la fois le temps et le coût de production.

De plus, des travaux déjà menés sur la gestion de la qualité de service ont montré que nombre de ces types de système utilisent deux approches. La première est l'auto-adaptation. Elle ne nécessite pas de plate-forme, les modes d'adaptation ne sont pas limités à ce que la plate-forme sait faire. Le concepteur maîtrise toutes les parties de l'application ce qui permet de pouvoir agir plus finement sur la partie métier. L'inconvénient de l'auto-adaptation est que pour la majorité, l'adaptation ne se fait que de façon locale. L'application n'a pas conscience d'elle-même dans sa totalité, elle réagit uniquement là où un problème a été détecté. Ce fonctionnement réduit considérablement le nombre de solutions d'adaptation. L'auto-adaptation impose de devoir prévoir le mécanisme de reconfiguration lors de son implémentation, il n'existe pas de frontière entre la logique métier et le système de gestion des propriétés non fonctionnelles. De plus ce mécanisme demande de prédéfinir des solutions correspondant à des évènements bien précis et laisse peu de place à l'imprévu. La deuxième est le pilotage d'application par des intergiciels ou plate-formes . Ce type de système permet la séparation des préoccupations. L'application s'occupe de la logique métier, les propriétés fonctionnelles, alors que tout ce qui concerne les propriétés non fonctionnelles comme la qualité de service est laissé à la charge de l'intergiciel. Une telle supervision permet également d'obtenir une vision complète de la structure de l'application et de pouvoir envisager des solutions de façon globale.

De nombreux travaux traitant des périphériques contraints se limitent à proposer des solutions basées sur des critères réseau et matériels ne se préoccupant pas de la qualité du service final rendu par l'application. Les données doivent continuer à parcourir le réseau tout en préservant l'énergie des nœuds, cependant, le respect des spécifications de l'application ou des besoins de l'utilisateur sont rarement mentionnés et pris en compte.

Afin de résoudre à la fois le problème de l'adaptation au contexte et de la durée de vie de l'application, nous proposons d'utiliser une plate-forme de supervisison servant de support à l'application (figure 2.2). Cette plate-forme permettra d'obtenir une vision globale du

fonctionnement de l'application et de percevoir les évolutions du contexte. Cette vision globale permet de proposer des solutions différentes qui ne sont pas uniquement basées sur des critères techniques de faisabilité mais guidées par des critères qualitatifs. La plate-forme permettra alors d'obtenir la meilleure QdS possible en modifiant dynamiquement :

- le comportement des composants. A la conception, les composants peuvent proposer différentes qualités pour une fonctionnalité. Par exemple, un composant peut proposer différentes résolutions d'images, 1024x768 pixels ou 800x600 pixels.
- l'ordonnancement des composants. L'application est constituée d'un ensemble de composants ordonnés. Pour des raisons d'équilibrage de charge par exemple, les liaisons entre les composants peuvent être modifiées. Pour une transmission vidéo nous pouvons avoir des traitements de réduction de taille d'images et de réduction du nombre de couleurs entre l'émission et la restitution de la vidéo. Selon le contexte, l'ordre entre ces deux traitements peut être modifié.
- la composition de l'application en ajoutant ou en supprimant des composants.
- le déploiement des composants en délocalisant des composants ou lors d'une modification de la composition de l'application.

Ces adaptations se feront en fonction des informations contextuelles que la plate-forme aura recueillies.

La section suivante présente les évolutions de la définition de la qualité de service et ses interactions avec le contexte ainsi que la définition que nous avons retenu pour ces travaux.

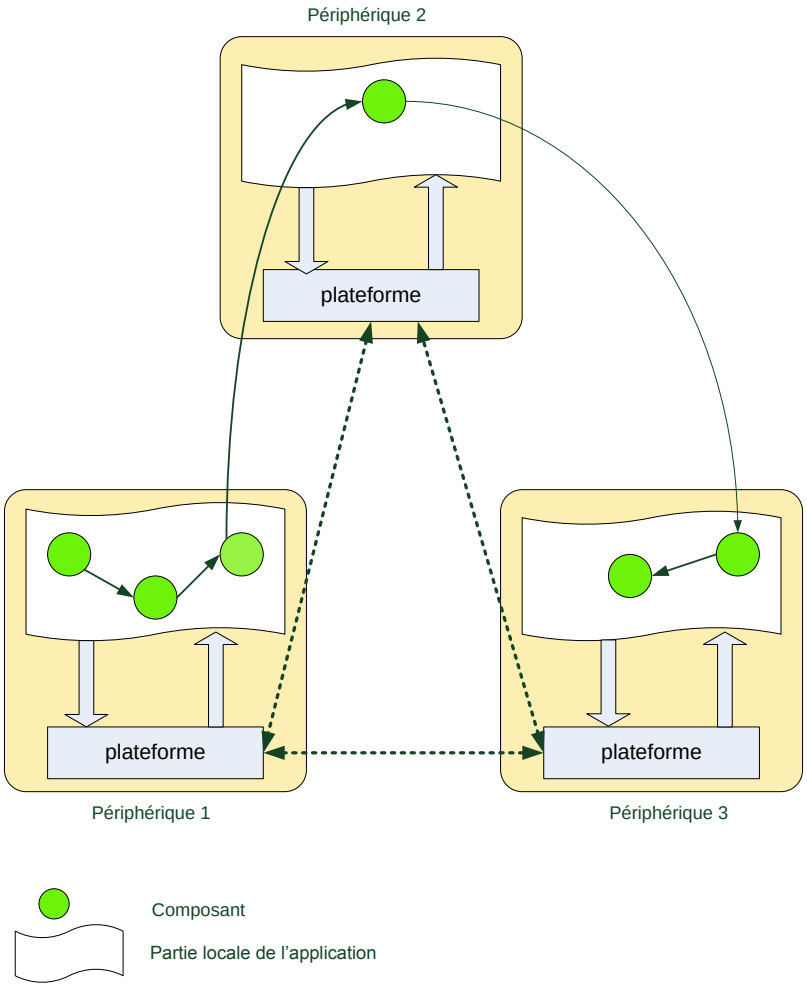

Figure 2.2 – plate-forme de supervision

2.4 Qualité de service

2.4.1 Définition de la Qualité de Service

Il n'existe pas de consensus autour de la définition de la qualité de service. Une unique définition ne convient pas à tous les domaines. Néanmoins il existe un standard X.902 défini par l'International Telecommunication Union [78] qui décrit la qualité de service comme un ensemble d'exigences concernant le comportement collectif d'un ou de plusieurs objets.

Cette notion est habituellement utilisée en réseau pour évaluer la performance des transmissions selon des critères quantitatifs tels que le délai, la gigue, le taux d'erreur, etc. Ces critères ne concernent que des propriétés non-fonctionnelles, c'est-à-dire qui ne sont pas en relation avec l'application.

A présent qu'Internet est à la portée de tout un chacun, les applications multimédias se sont multipliées et par conséquent la notion de qualité de service est de plus en plus utilisée à d'autres fins que l'évaluation des réseaux. Les définitions que l'on retrouve le plus souvent dans la littérature s'accordent à dire qu'actuellement il n'est plus possible d'évaluer la qualité de service d'une application en tenant compte uniquement des critères réseau et matériels [54][27].

[7] décrit alors la qualité de service comme l'ensemble des caractéristiques quantitatives et qualitatives d'un système multimédia distribué qui permet d'atteindre la fonctionnalité requise pour une application. Il apparaît clairement que l'intégration et la prise en compte du point de vue de l'utilisateur est nécessaire. Cependant elle n'est pas suffisante pour l'évaluation de la qualité de service lorsqu'il s'agit de l'utilisation de périphériques contraints. En effet les travaux menés actuellement sur la qualité de service des réseaux de capteurs montrent que celle-ci est évaluée en termes de précision des données mesurées et de durée de vie du réseau. Il est d'ailleurs montré que la précision des données est fonction du niveau d'énergie des noeuds [38]. Afin d'obtenir des résultats pertinents et efficaces, il est d'intérêt d'optimiser la consommation d'énergie et de préserver au maximum la durée de vie des noeuds. Dans de précédents travaux [42], nous avions défini la qualité de service comme étant « l'adéquation entre le service souhaité par l'utilisateur et le service qui lui est fourni ». Cette définition permet de prendre en compte à la fois la définition classique de la qualité de service réseau mais aussi les aspects fonctionnels comme l'ergonomie et la personnalisation du service.

Alors que la plupart des travaux se concentrent uniquement sur un ou deux aspects de la qualité de service, réseau et utilisateur, nous souhaitons intervenir à tous les niveaux, représentés dans la figure 2.3. Au niveau de l'utilisateur, nous souhaitons garantir le respect des souhaits de l'utilisateur. Au niveau de l'application, nous souhaitons garantir le respect des contraintes d'utilisation définies dans les spécifications. Ces deux niveaux concernent l'utilisation de l'application, le premier décrit ce que l'on voudrait faire avec cette application alors que le deuxième définit ce qu'il est possible et ce qu'il n'est pas possible de faire. L'utilisateur et l'environnement de l'application influencent la qualité de service liée à l'utilisation. L'infrastructure regroupe les périphériques, les composants logiciels et le réseau. Au niveau de l'infrastructure, nous souhaitons garantir à la fois la continuité de service et une durée de vie la plus longue possible de l'application malgré les variations des ressources matérielles, logicielles et réseaux et malgré la mobilité des périphériques.

Le principal objectif reste d'assurer la continuité des services de l'application. C'est lui qui va guider la plate-forme lors des reconfigurations.

2.4.2 Qualité de Service et Contexte

La définition de [19] (section 2.2) signifie que les évolutions du contexte se traduisent par des variations de la qualité du service rendu par l'application. La gestion de la QdS est une réponse possible aux évolutions du contexte. Dans nos travaux, nous proposons des outils de gestion de QdS afin d'adapter les applications de façon à ce qu'elles puissent fournir la meilleure qualité possible.

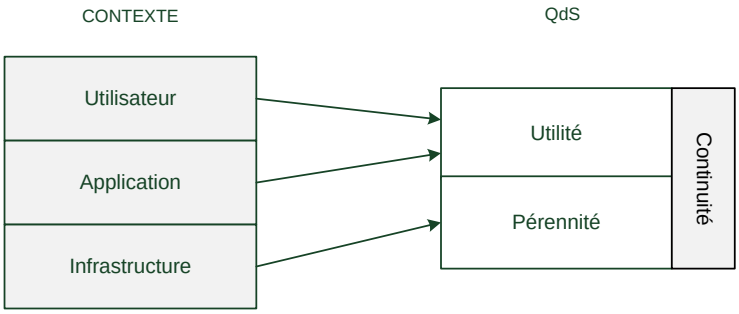

Figure 2.3 – Niveaux de qualité de service

Lorsque nous parlons de qualité de service d'une application, il est évident que le principal objectif est de garantir la continuité du service malgré les défaillances matérielles, logicielles et réseau. La continuité de service est essentiellement influencée par l'infrastructure de l'application. En effet, l'utilisation de périphériques mobiles et par conséquent des réseaux sans-fil implique des perturbations telles que les variations de débit ou l'indisponibilité des données. Nous devons également faire face aux problèmes d'hétérogénéité dus à l'utilisation de plusieurs variétés de périphériques, ainsi qu'aux problèmes de pannes matérielles comme le déchargement de la batterie des périphériques.

Face à ces différents besoins de qualité de service, plusieurs notions ont été proposées afin de mesurer les variations de contexte des environnements mobiles. Selon [3], les environnements mobiles sont caractérisés par une fluctuation et une limitation des ressources couplées à une variation des souhaits des utilisateurs. Il devient alors nécessaire d'adapter les systèmes afin d'assurer de façon continue et autonome la **sûreté** de l'application. Sous le terme de **sûreté**, [3] désigne le maintien de la facilité d'utilisation, l'utilité et la fiabilité de l'application face aux changements du contexte. De la même manière, [80] introduit la notion d'**utilité** comme un indicateur de satisfaction associé à une exigence de ressource que l'application doit fournir. Il indique la préférence du mode de fonctionnement de l'application en terme de QdS tel que "normal", "dégradé". Enfin, avec l'utilisation grandissante des appareils mobiles et notamment les capteurs pour la mesure du contexte, la préoccupation de conservation de l'énergie est de plus en plus présente. La durée de vie des appareils fait partie des critères de QdS au même titre que la continuité ou le délai [71]. Pour [53], la durée de vie d'un appareil est un critère de choix pour le rôle à jouer dans le réseau : traitement, relais, etc.

A partir de ces constatations, nous admettons que la continuité de service fait partie de la QdS globale de l'application à laquelle nous ajoutons deux types de QdS : la *QdS Utilité* et la *QdS Pérennité*.

Selon les objectifs évoqués dans la section 1.3, nous nous intéressons à deux thématiques. La première concerne le fait que nous devons faire en sorte que l'application fonctionne le mieux possible. Il s'agit ici de fournir un service qui réponde d'une part le mieux possible aux souhaits de l'utilisateur et d'autre part qui respecte les obligations et les restrictions dues aux spécifications de l'application. La deuxième thématique s'intéresse à la durée de vie de l'application. Elle doit fonctionner le plus longtemps possible. Ceci

est dû au fait que nous utilisons des périphériques contraints. Il s'agit ici de fournir des méthodes et des outils permettant de préserver les ressources et notamment les ressources énergétiques des périphériques.

Afin de répondre à ces deux thématiques, nous proposons de gérer la qualité de service selon deux niveaux. Le premier niveau concerne la *QdS Utilité* qui correspond au bon fonctionnement de l'application conformément aux spécifications et aux souhaits de l'utilisateur. Le second niveau concerne la *QdS Pérennité* qui correspond à la maximisation de la durée de vie de l'application.

Pour chacun des deux niveaux de QdS, nous devons fournir les outils et des critères d'évaluation. Ces critères seront basés sur les informations d'évolution de la QdS : le contexte. Nous allons décrire les deux niveaux de QdS et montrer les interactions qu'il existe entre les informations contextuelles et les QdS.

Utilité Les divers travaux traitant de la qualité de service se focalisent essentiellement sur la continuité de service. L'objectif est de faire fonctionner l'application en dépit des dysfonctionnements du matériel et du réseau. Cependant, le plus souvent, faire fonctionner l'application se résume à trouver des solutions basées sur des protocoles réseaux afin de continuer à acheminer des données d'un point A à un point B. La partie fonctionnelle de l'application est rarement traitée dans le sens où implicitement, il est considéré que l'application rend le service souhaité par l'utilisateur.

Néanmoins, nous considérons qu'il est important de prendre en compte la dimension métier de l'application. En effet, bien qu'il paraisse évident que celle-ci soit destinée à un utilisateur et doive répondre à ses besoins, elle est encadrée par des règles de fonctionnement et doit répondre à un cahier des charges. Nous devons garantir la continuité de service de l'application mais également vérifier, lors des changements de contexte, si l'application propose toujours un service adéquat et respectant les conditions d'utilisation. Ce niveau de QdS concerne la conformité d'utilisation de l'application, le respect des contraintes définies par ses spécifications et par les souhaits de l'utilisateur.

Pérennité Dans le domaine des périphériques contraints, le contrôle de la consommation des ressources est un objectif imporant. Ce type de périphérique a la particularité d'être autonome en énergie. Lorsqu'il n'en dispose plus, il devient inutilisable. L'utilisation de ressources comme le processeur ou le dispositif de transmission sans-fil ont un impact non négligeable sur la consommation d'énergie. Plus il y a de données à transmettre et à traiter, plus il y a d'énergie consommée. Des études menées sur l'utilisation de la radio des capteurs sans-fil ont montré que la transmission de données sur le réseau consomme entre dix et cent fois plus d'énergie que le calcul. Nous avons donc tout intérêt à minimiser les transmissions de données et maximiser les calculs. Lorsqu'un périphérique vient à ne plus fonctionner, tous les services jusqu'alors exécutés sur celui-ci sont déconnectés et ne peuvent plus être utilisés, compromettant fortement la continuité de l'application.

En effet, la continuité des services repose sur la durée de vie de l'application et par extension repose sur la durée de vie des périphériques. La garantie d'un service de qualité passe aussi par la garantie d'un service pérenne. C'est pourquoi nous choisissons de prendre en compte la dimension de pérennité de l'application dans l'évaluation de sa qualité de

service globale.

2.5 Synthèse

Au terme de cette section, nous pouvons constater que le contexte et la qualité de service sont indéniablement liés. Toute modification du contexte peut être interprêtée comme une modification de la qualité du service rendu. Ainsi les informations contextuelles apportent des éléments essentiels sur l'évolution de la qualité de service globale de l'application. Lorsqu'une modification du contexte survient, cela veut signifie que le contexte a été modifié et que par conséquent le service proposé n'est peut-être plus adapté dans sa dimension Utilisation ou dans sa dimension Pérennité ou dans sa globalité. Nous définissons une QdS globale dans laquelle nous incluons la continuité de service puisque elle est le principal objectif à garantir lorsqu'il s'agit de qualité. En effet les premiers travaux sur la QdS consistaient essentiellement à proposer des mécanismes permettant de maintenir le fonctionnement d'un service avant d'évoluer vers la prise en compte de l'utilisateur et de l'environnement physique. Le type QdS continuité est alors englobé par la QdS globale. Ensuite nous distinguons deux types de QdS : Utilisation et Pérennité. Ces deux types sont la conséquence de la prise en compte de l'utilisateur et de l'environnement de l'application d'une part et de l'infrastructure d'autre part. Il est important de noter que ces modifications de contexte ne doivent pas seulement être vues comme l'introduction de contraintes implémentatives à prendre en compte. En effet elles peuvent également correspondre à des disparitions de contraintes qui permettent d'améliorer le service global. Gérer le contexte c'est donc tout autant savoir dégrader le service offert pour l'adapter aux contraintes que l'améliorer lorsque le contexte le permet. Les informations contextuelles sont donc les déclencheurs du processus d'évaluation de la QdS. Elles vont être une base pour la définition des critères d'évaluation de QdS. Dans la figure 2.3, nous avons représenté les différents niveaux auxquels nous souhaitons agir en termes de qualité de service. Nous avons identifié trois niveaux influençant la QdS : l'utilisateur, l'application et l'infrastructure. La QdS étant influencée par les évolutions du contexte, selon les types de contexte que nous avons définis dans la section 2.2, chacun des niveaux correspond respectivement au *contexte utilisateur*, au *contexte d'utilisation* et au *contexte d'exécution*.

La figure 2.4 illustre les liens d'interaction entre les différentes types de contexte et les deux types de QdS définis dans la section 2.4.2.

Le *contexte utilisateur* est défini comme l'ensemble des entités caractérisant les souhaits de l'utilisateur. Ce sont toutes ses exigences relatives à ses envies d'utilisation des services. Une modification des souhaits de l'utilisateur a donc des conséquences sur la conformité du service rendu. Une telle conformité est gérée par la dimension Utilisation de la QdS. Le *contexte utilisateur* a donc des répercussions sur la *QdS Utilité*. Une modification des souhaits d'utilisation entraine une modification de la *QdS Utilité*.

Le *contexte d'utilisation* est vu comme l'ensemble des entités intervenant dans les prérequis des obligations et restrictions sur le fonctionnement des services proposés par l'application. De la même manière que pour le *contexte utilisateur*, une modification du

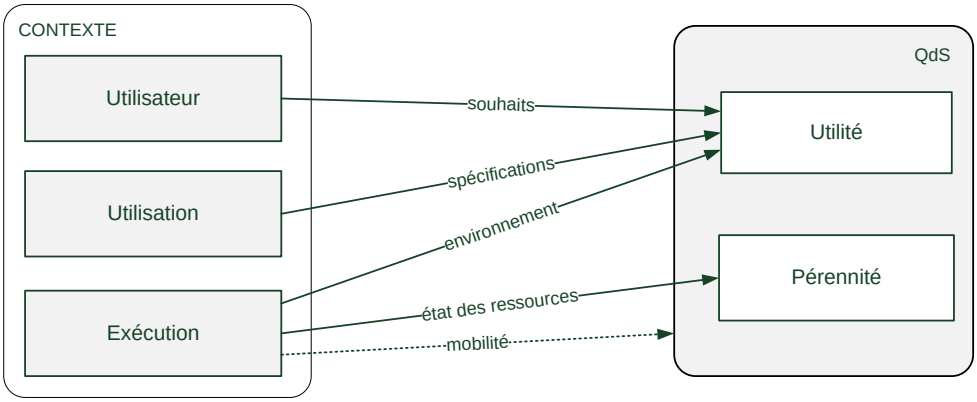

FIGURE 2.4 – Interactions entre contexte et QdS

contexte d'utilisation entraine une modification des contraintes requises pour le bon fonctionnement de l'application. Le respect de ces contraintes est géré par la *QdS Utilité*. Tout comme le *contexte utilisateur*, le *contexte d'utilisation* a des répercussions sur la *QdS Utilité*.

Enfin le *contexte d'exécution* est défini comme l'ensemble des propriétés permettant de caractériser le support de l'application. Ce sont les ressources matérielles, logicielles et réseau des périphériques. L'évolution des ressources au cours de l'exécution a essentiellement un impact sur la durée de vie du service et par extension de l'application toute entière. L'utilisation des différents composants physiques d'un périphérique a des conséquences sur la consommation d'énergie. Un défaut d'énergie correspond à la perte d'un périphérique et à la déconnexion de tous les services qu'il supporte. La consommation des ressources est gérée par la *QdS Pérennité*. Un changement de l'état des ressources entraine des modifications sur la durée de vie de l'application.

La mobilité affecte la QdS de l'application dans son ensemble. D'une part, lorsqu'un composant n'atteint plus le périphérique qui détient les composants qui précèdent et qui suivent ceux qu'il supporte, cela provoque l'arrêt du service auquel ces composants participaient. La continuité du service est alors perturbée, le service n'est plus rendu, l'application n'atteint plus ses exigences d'utilisation et les souhaits de l'utilisateur ne sont plus respectés, la *QdS Utilité* se dégrade. Également, un périphérique qui s'éloigne trop de ses prédécesseurs et de ses suivants, doit produire davantage d'effort pour maintenir un signal d'une puissance suffisante pour continuer de recevoir et transmettre des informations. Le maintien d'un signal de transmission consomme une quantité conséquente d'énergie. Assurer le service de cette manière pourrait avoir des conséquences négatives sur la continuité de service étant donnée l'autonomie en énergie des périphériques, un périphérique dans une telle situation pourrait voir sa durée de vie diminuer rapidement. Par conséquent, la *QdS Pérennité* se dégrade également.

2.6 Les architectures logicielles pour les systèmes sensibles au contexte

La particularité des systèmes sensibles au contexte est qu'ils réagissent aux différents changement du contexte afin de fournir à l'utilisateur des services adaptés à la situation. On distingue cinq types d'adaptation.

La première est ***l'adaptation de contenu***. En fonction des situations, les données sont modifiées pour ne présenter à l'utilisateur que celles qui sont pertinentes à sa situation.

La deuxième est ***l'adaptation de présentation***. Ce type d'adaptation touche le domaine des interfaces graphiques homme-machine. En fonction du statut hiérarchique de l'utilisateur, l'interface de l'application présentera ou non une information et proposera ou non une fonctionnalité.

Le troisième type est ***l'adaptation de comportement*** et l'adaptation des fonctionnalités fournies par un composant ou un service. Par exemple le paramétrage d'un composant correspond à une adaptation de comportement. Ce type d'adaptation peut parfois amener le développeur à briser l'abstraction boîte noire des composants ce qui limite la réutilisation de ces composants et empêche l'utilisation des composants sur étagère, COTS.

Le quatrième type d'adaptation est ***l'adaptation structurelle***. Elle vise à modifier la composition de l'application et/ou les connexions entre les différents composants permettant d'obtenir une application dont le comportement reste inchangé. C'est l'adaptation la plus utilisée actuellement dans le domaine des applications distribuées basées composants et c'est le type d'adaptation que nous visons dans cette thèse.

Enfin, la dernière adaptation est ***l'adaptation de déploiement***. Elle vise à proposer des déploiements qui prennent en compte les propriétés des périphériques supportant l'application. C'est une adaptation de plus en plus utilisée pour faire face aux problèmes engendrés par les limitations matérielles des périphériques mobiles et contraints massivement utilisés de nos jours.

Les trois premiers types d'adaptation, *adaptation de contenu, adaptation de présentation* et *adaptation de fonctionnalité* sont essentiellement tournées vers l'utilisateur. Le contenu et les fonctionnalités sont adaptés en fonction de ses préférences et la présentation est adaptée en fonction de son statut par exemple. Les adaptations de structure et de déploiement conviennent particulièrement aux contraintes matérielles et réseaux. Les fonctionnalités restent inchangées malgré les changements de contexte. Dans cette thèse, nous centrons nos recherches sur les contraintes de l'application et des périphériques plutôt que sur les contraintes de l'utilisateur.

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

2.6.1 Intergiciel pour l'adaptation contextuelle d'applications

Ce paragraphe décrit Aura, WComp et MUSIC, trois intergiciels qui ont choisi de répondre à la problématique de la sensibilité au contexte par une adaptation structurelle

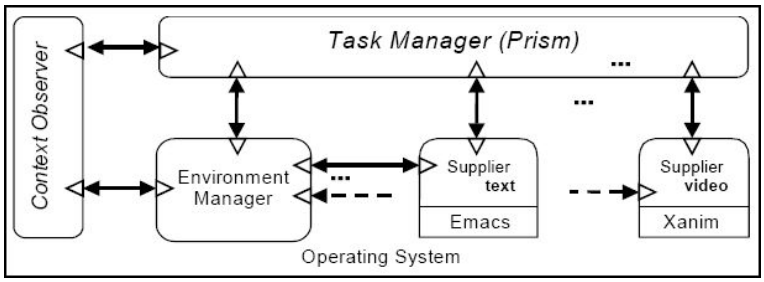

FIGURE 2.5 – Architecture générale d'Aura

des applications.

2.6.1.1 Aura

Aura [72] est un intergiciel sensible au contexte, centré autour de l'utilisateur, qui permet de concevoir des applications mobiles. Sa particularité réside dans le fait qu'il permet la migration de tâches selon le contexte. Ainsi, lorsqu'un utilisateur se déplace d'une localisation géographique vers une autre, et lorsque les ressources s'en vont et reviennent, le système s'adapte et offre à l'utilisateur un accès optimal, auto-ajusté et continu aux données et aux traitements.

Un des scénarios d'utilisation d'Aura est celui d'un utilisateur, Fred, qui travaille sur l'organisation d'une conférence depuis son domicile. Il parcourt le web à la recherche d'informations sur les hôtels qui pourraient accueillir les participants. Certaines de ces pages proposent de visualiser une vidéo pour une visite virtuelle des chambres de l'hôtel. Fred en a téléchargé quelques unes. Il prend également des notes sur un traitement de texte concernant les budgets. Fred sort de chez lui pour se rendre à son bureau. Pour que Fred puisse continuer à travailler sur l'organisation de la conférence, Aura configure cette tâche à son bureau afin qu'il puisse reprendre son travail dès qu'il arrive : il ouvre un navigateur avec les pages récemment visitées, les vidéos téléchargées mises en pause au même moment qu'à son départ et un traitement de texte contenant tout le travail déjà réalisé. Puisqu'il existe un grand écran dans le bureau de Fred, il est préféré à son ordinateur pour visualiser les vidéos et naviguer sur le web, libérant l'écran de l'ordinateur pour le traitement de texte.

Dans ce scénario, Aura prend automatiquement en charge la configuration et la reconfiguration des tâches de façon transparente face à un changement de l'environnement, ici un déplacement d'un endroit vers un autre. Pour proposer un tel système, Aura repose sur quatre sous-systèmes (figure 2.5) : Task Manager, Context Observer, Environment Manager et Supplier. Le Task Manager incarne le concept d'*aura* personnelle de l'utilisateur. L'aura sert d'interface entre l'utilisateur et l'environnement (ressources matérielles et logicielles). L'aura permet de prendre en compte les préférences de l'utilisateur concernant la configuration, les fonctionnalités et la qualité des services fournis ainsi que de minimiser la distraction de l'utilisateur face aux changements suivants :

- changement de localisation et nouvel environnement
- modification de l'environnement
- changement de tâche (service)

– changement de contexte

Le *Context Observer* fournit les informations sur le contexte physique et fait part à l'*Environment Manager* et au *Task Manager* des évènements importants de changement du contexte. L'*Environment Manager* représente la passerelle vers tout les supports qui composent l'environnement de l'utilisateur. Enfin les *Suppliers* sont les composants qui fournissent les services qui composents les tâches (services) utilisateurs tels que le traitement de texte, la lecture vidéo, etc. Chaque fois que l'utilisateur entre dans un nouvel environnement, les composants d'Aura communiquent entre eux afin de déterminer les migrations à effectuer permettant aux tâches de s'adapter en essayant de faire correspondre les préférences de l'utilisateur et les ressources et capacités qu'offrent l'environnement. La Task Manager est le composant décisionnaire de la reconfiguration. Pour effectuer son choix, il se base sur une description centrée sur les préférences de l'utilisateur des configurations possibles. Ces configurations sont détaillées à grain fin : compositions de suppliers préférées, niveau de qualité de service demandé, etc. Le choix est ensuite guidé par une fonction d'utilité définie par une distance entre le bénéfice, la satisfaction de l'utilisateur, et le coût d'une reconfiguration.

L'approche proposée par Aura situe l'utilisateur au centre de toute décision de reconfiguration. En effet, l'adaptation est dirigée en fonction de ses préférences au préalablement définies :
– préférences de configurations pour un service
– préférences des ressources logicielles
– préférences de QdS pour chaque service

Grâce aux préférences de QdS, Aura intègre les contraintes de ressources matérielles dans ses politiques d'adaptation.

Bien que ces préférences soient décrites indépendamment du contexte, elles définissent des configurations a priori à grain très fin qui peuvent parfois conduire à des arrêts de service en raison du grand nombre de conditions à vérifier.

De plus, la dynamicité du système est réduite puisqu'il n'est pas libre de composer le service. Un tel système ne permet pas de garantir la continuité du service quelles que soient les conditions de contexte.

2.6.1.2 WComp

WComp [75][14][49] est une approche basée composants légers pour concevoir des services web composites. D'une part, elle fournit un cadre de travail permettant de construire des applications sous forme de graphe de services web basés sur le concept de Container. D'autre part, elle fournit un intergiciel basé sur le concept d'Aspects d'Assemblage permettant d'adapter les services web.

Le cadre de travail de WComp repose sur le paradigme SLCA, Service Lightweight Component Architecture [33]. Ce paradigme regroupe les principes du paradigme service web basé évènement et du paradigme composant. Le premier apporte l'interopérabilité face à l'hétérogénéité des dispositifs qui composent un système ubiquitaire. L'utilisation de services tels que UPnP [35] et DPWS [69] facilitent la communication des services.

La réutilisation des services permet l'extensibilité, et la communication basée évènements garantit une forte réactivité du système. Enfin un autre avantage du paradigme services web est qu'il permet la mobilité des applications. Le paradigme composant apporte la dynamicité et la flexibilité notamment au niveau de la structure qu'il est possible d'adapter.

On distingue deux types de services : les services composites et les services basiques. Alors que les services basiques se suffisent à eux-mêmes, les services composites nécessitent la collaboration d'autres services. Dans WComp, chaque service composite encapsule un Container WComp qui gère dynamiquement un assemblage de composants légers WComp. Ce container implémente une API permettant de contrôler l'assemblage au moyen d'actions élémentaires : l'ajout et la suppression de composants et l'ajout et la suppression de connexions entre ces composants. Une application est alors composée de plusieurs services web réalisés chacun par un assemblage de composants WComp.

Le modèle de composants WComp basé sur le modèle JavaBeans permet d'appréhender la composition pour fournir des services de haut niveau. La communication basée évènement et la flexibilité de la structure sont des atouts pour les systèmes pervasifs tout en conservant la propriété boite noire des composants.

Le modèle d'adaptation des applications utilise le concept d'Aspects d'Assemblage [74][15]. Ce concept permet de tisser des schémas d'adaptation indépendants et transversaux qui se fusionnent en suivant des règles logiques en cas de conflit. Les adaptations sont un ensemble d'aspects d'assemblages qui ont été conçus à partir des connaissances a priori, mais partielles, des services sur lesquels ils s'appliquent. Cette approche permet de modifier la structure d'un assemblage dans deux cas : un assemblage qui représente localement une application ou un assemblage encapsulé dans un service composite. Un aspect d'assemblage s'appuie sur le principe de la programmation orientée aspect, AOP [39][11]. Il définit un point de coupe, un greffon et un tisseur. Un point de coupe permet de spécifier l'endroit de l'application où est inséré un greffon. C'est une succession de filtres sur les composants logiciels de l'application qui permet de sélectionner la liste des composants où le greffon sera intégré. Un greffon est une fabrique de sous-assemblages. Il est défini par rapport à un point de coupe et décrit un comportement qui est exécuté à l'endroit du point de coupe. Les spécifications de comportement sont ensuites traduites en un ensemble de modifications structurelles élémentaires par le tisseur : ajout et suppression de composants et ajout et suppression de connexions.

Pour chaque situation de contexte qui peut être observée est défini un aspect. Par exemple il est possible de définir différents aspects selon le niveau de batterie. Lorsque le niveau de batterie change ou si l'utilisateur souhaite modifier la gestion de l'utilisation de la batterie comme il est possible de faire sur les ordinateurs portables ou les PDA, l'aspect d'assemblage correspondant définit les différentes modifications à effectuer sur les composants et les connexions et les envoie au container qui contrôle l'assemblage du service web concerné.

L'approche proposée par WComp garantit une grande réactivité du système face aux changements de contexte et permet une reconfiguration aisée des applications grâce aux modifications élémentaires. Cependant il nécessite d'avoir prévu toutes les situations de changement de contexte et de définir un ou plusieurs aspects d'assemblage pour chaque situation. Le coût des adaptation telles qu'elles sont décrites dans [14] n'est pas pris en

compte. Or, l'utilisation de périphériques contraints nécessite de calculer la consommation des ressources du périphérique par chaque composant ainsi que la consommation des ressources réseau.

2.6.1.3 MUSIC

Le projet MUSIC [64][65][59] fait partie des travaux de référence dans l'expérimentation des processus d'adaptation des applications mobiles. MUSIC est un intergiciel permettant la reconfiguration d'applications mobiles et sensibles au contexte. Le processus d'adaptation défini dans MUSIC repose sur les principes de l'adaptation par planification, *planning-based adaptation*.

> L'adaptation par planification réfère à la capacité de reconfiguration d'une application en réponse aux changements de contexte en exploitant les connaissances de sa composition et des méta-données de QdS associées à chacun des services la constituant [26].

Afin que l'application possède toutes les informations qu'impose l'adaptation par planification, MUSIC propose une méthode pour la conception des applications.

La première étape consiste à identifier les cas d'utilisation de l'application. Les cas d'utilisation sont un moyen de mettre en avant l'environnement et toutes les parties fonctionnelles de l'application qui sont influencées par des changements de contexte. Cela permet d'identifier tous les évènements auxquels la plate-forme doit réagir.

La deuxième étape consiste à modéliser les parties fonctionnelles de l'application. Les applications sont constituées d'assemblages de composants qui peuvent avoir plusieurs implémentations possibles. Selon ces différentes implémentations, le concepteur définit toutes les *variations* possibles pour réaliser l'application. Pour chaque composant, il décrit toutes les caractéristiques et les propriétés (*property predictor*) nécessaires au processus d'adaptation comme la consommation de batterie, la consommation de mémoire, s'il requiert une ressource particulière comme un écran, etc. Des contraintes de placement et d'architecture sont également définies. Elles permettent d'imposer le déploiement d'un composant sur un type de nœud et l'utilisation d'un composant particulier pour réaliser une fonctionnalité. Chaque *variation* possède également ses *property predictor*. Grâce à ces propriétés, le concepteur peut définir une fonction d'utilité à l'application, la clé du processus d'adaptation.

La troisième étape consiste à proposer des modèles de déploiement. Chaque *variation* est associée à un ou plusieurs modèles de déploiement qui représentent la distribution des différents composants sur des types de nœuds.

Les propriétés sont relatives au contexte de l'application qu'il convient de décrire. Le concepteur doit définir l'environnement d'exécution de l'application en termes de contexte, de ressources et de services disponibles. Pour cela, il se base sur une ontologie qu'il peut étendre selon ses besoins. MUSIC propose un modèle de contexte où chaque information est représentée par un élément de contexte dont les principaux attributs sont son nom, sa représentation, sa valeur, sa dimension et ses méta-données. Un modèle similaire est proposé pour la modélisation des ressources. Un gestionnaire de contexte est consitué de composants spécifiques de capture et de raisonnement distribués sur les différents périphériques

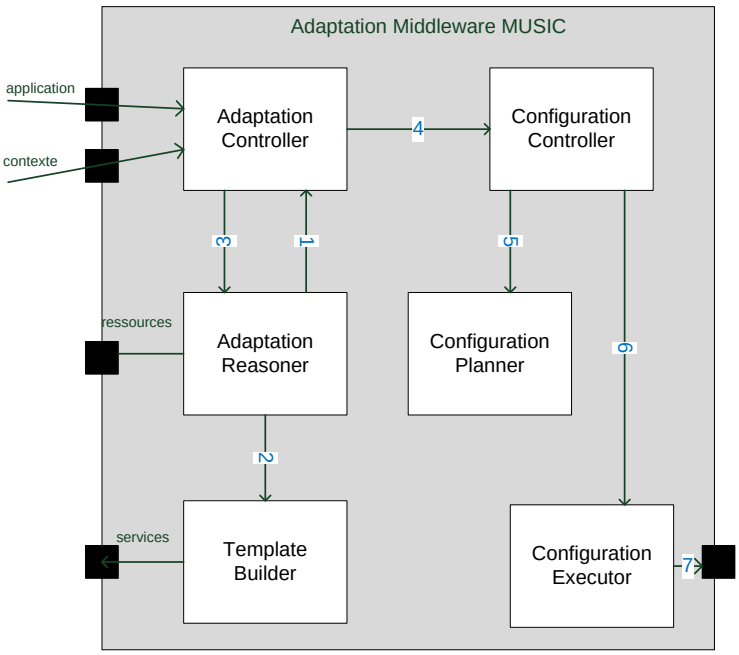

FIGURE 2.6 – Structure du gestionnaire d'adaptation de MUSIC

de l'application permettant ainsi d'avoir une vue à la fois locale et globale du contexte de l'application.

Le processus d'adaptation est réalisé par un gestionnaire d'adaptation qui repose sur la collaboration de six services comme le montre la figure 2.6. Il fonctionne de la façon suivante. L'*Adaptation Controller* est le service responsable de la réception des évènements de changement de contexte et de l'invocation du processus d'adaptation, l'*Adaptation Reasoner*. Ce service doit obligatoirement être déployé sur tous les périphériques. Il est déclenché chaque fois qu'une application est lancée ou arrêtée et chaque fois que le gestionnaire de contexte rapporte un changement de contexte.

L'*Adaptation Reasoner* est responsable du choix de la configuration et de la prise de décision. Il dépend du *Template Builder* qui fournit toutes les *variations* possibles de l'application. Ce dernier peut utiliser des heuristiques de choix afin de réduire le nombre de *variations* à étudier. Pour chaque variation, l'*Adaptation reasoner* calcule son utilité en fonction du contexte fourni et des propriétés des composants et choisit la configuration qui offre la plus grande utilité. La variation choisie est alors envoyée à l'*Adaptation Controller* qui la relaie au *Configuration Planner* qui décrit toutes les modifications nécessaires au déploiement de la variation. Puis ces modifications sont transmises au *Configuration Executor* qui les exécute.

L'approche proposée par MUSIC est une approche centrée sur le périphérique. Un domaine d'adaptation est défini autour du périphérique pour définir une zone où l'*adaptateur*

peut piocher des composants pour réaliser les adaptations. Pour faciliter le déploiement, MUSIC propose de concevoir des modèles de déploiement pour chaque configuration. Ceci permet de minimiser la complexité de l'application, de limiter les recherches et de fournir un résultat rapide. Cependant le côté statique de cette démarche limite la dynamique des systèmes pervasifs et mobiles. Enfin, bien que le caractère contraint des périphériques soit pris en compte dans les fonctions d'utilité, notamment lors de définition des coûts des composants, il manque la définition des coûts des connexions dans le cas d'un déploiement distribué.

2.6.2 Déploiement contextuel d'applications

L'adaptation structurelle est une réponse efficace aux changements de contexte. Elle modifie la composition d'une application pour lui permettre de continuer de fonctionner malgré les aléas de l'environnement. Cependant, une telle adaptation ne prend pas toujours en compte la particularité des dispositifs supportant l'application. L'adaptation de déploiement est un complément nécessaire lorsqu'il s'agit de traiter des périphériques contraints. Dans ce paragraphe nous décrivont CADeComp et AxSel, deux intergiciels pour le déploiement contextuel d'applications.

2.6.2.1 CADeComp

CADeComp [8] est un intergiciel pour le déploiement sensible au contexte des applications basées composants. Cet intergiciel étend les services de déploiement existants en y intégrant les capacités d'adaptation nécessaires au domaine des applications mobiles et des périphériques contraints. Il propose un déploiement automatique à la volée et sensible au contexte : une application est installée au moment de son accès et désinstallée juste après la fin de son utilisation.

Les applications sont considérées comme une collection de composants distribués sur le réseau et reliés entre eux via des ports. Le déploiement est défini selon cinq paramètres : l'architecture de l'application, le placement des instances composants, le choix de leur implémentation, les propriétés des composants et leurs dépendances. CADeComp repose sur un modèle de données permettant de décrire le contexte qui agit sur le déploiement et de définir des contrats de déploiement qui associent à chaque situation de contexte toutes les variations possibles des paramètres de déploiement. Le contexte modélise essentiellement les caractéristiques des instances des composants. Ces informations sont collectées lors de la spécification et du développement par le producteur du composant. Il permet de spécifier des contraintes sur le placement des composants ainsi que sur les connexions, obligatoires ou optionnelles.

Toutes ces informations permettent à CADeComp de contruire un plan de déploiement qui résulte du traitement des données de contexte et de la prise de décision d'adaptation. Ce plan de déploiement décrit l'architecture de l'application à déployer, ses instances de composants, leur implémentation, leur placement, leur dépendances et les connexions à mettre en place. Le déploiement est déterminé au travers de contrats. Un contrat est un ensemble de règles qui indiquent les variations des propriétés d'un composant, de ses dépendances et de son implémentation selon une situation de contexte précise.

CADeComp propose une méthode permettant d'aider le producteur de composants à fournir toutes les méta-données nécessaires à la définition des règles d'adaptation indépendamment de toute plate-forme. Il propose également un service de déploiement constitué d'un ensemble de composants adaptatifs placés au-dessus des services de déploiement existants tel que CORBA.

CADeComp est une approche intéressante car elle propose un mécanisme d'adaptation qui modifie à la fois la structure et le déploiement de l'application tout en prenant en compte explicitement les propriétés des composants tel que les contraintes de placement et le caractère optionnel des connexions. Cependant, son approche, basée sur les architectures dirigées par les modèles, impose de prévoir les adaptations au contexte et limite le caractère spontané des systèmes pervasifs.

2.6.2.2 AxSel

AxSel [30] est un intergiciel pour le déploiement autonome et contextuel des applications orientées services. Il utilise une approche basée sur les graphes pour proposer un processus de choix de déploiement en fonction des informations de contexte récoltées. Il introduit le concept de service composant et le définit comme suit :

Définition 2.1 (Ben Hamida) *Un service composant est un concept hybride qui réunit le service et le composant. Un composant est une unité logicielle encapsulant des fonctionnalités. Il peut être déployé et exécuté. Il possède une description et des interfaces. Une interface correspond à un service. Les services permettent l'import et l'export des fonctionnalités. Un composant est sujet à composition avec d'autres composants à travers des services importés et exportés. Un composant et ses services ne sont pas dédiés à un utilisateur spécifique.*

Le fonctionnement d'AxSel réside alors en trois éléments : un modèle d'application, un modèle de contexte et des heuristiques de déploiement.

Le *modèle d'application* définit des applications orientées services composants permettant une vision globale des dépendances entre les services composants. Cette vision à grain fin de l'application permet aux auteurs d'intégrer à chaque nœud des informations relatives à l'exécution des services et au déploiement des composants. Cette démarche facilite le processus de choix de déploiement. De plus elle apporte la flexibilité nécessaire à la reconfiguration des applications en permettant l'ajout et la suppression d'éléments du graphe. Une autre particularité d'AxSel est que les auteurs intègrent directement au niveau des graphes la notion de multi-fournisseurs, là où d'autres approches définissent des configurations ou des *plans*.

Le *modèle de contexte dynamique* est simple et basé sur des informations contextuelles de l'environnement à l'exécution. Pour la capture, il propose un mécanismes d'écouteurs de contexte. Des écouteurs sont définis pour chaque ressource à surveiller et sont basés sur un mécanisme publish/subscribe. Ils sont gérés par un service de gestion du contexte qui relaie le tout au service de décision. Ce dernier effectue ensuite un raisonnement sur le contexte en confrontant le contexte requis au contexte fourni ce qui résulte en une décision de reconfiguration. Ces reconfigurations sont calculées par des heuristiques de déploiement.

Ce sont des *heuristiques* multi-critères qui reposent sur des mécanismes de gestion des dépendances, de contextualisation du déploiement et d'adaptation à la volée. Après avoir extrait les dépendances de l'application, AxSel évalue la faisabilité du déploiement en se basant sur les contraintes. Il en résulte une liste de composants à charger ou à décharger sur le terminal.

AxSel aborde la problématique du déploiement contextuel souvent ignoré dans les approches de reconfiguration des systèmes pervasifs. Sa petite taille, 65Ko, permet de l'intégrer sur la majorité des périphériques sous réserve de disposer d'une machine virtuelle Java adéquate pour la version actuellement développée. L'utilisation de l'approche à service masque malheureusement l'aspect distribution de l'application et les coûts énergétiques qu'engendrent les liaisons réseau pour les périphériques contraints.

2.6.3 Architectures logicielles pour la qualité de service

Lorsqu'il s'agit de gérer la qualité de service dans les environnements mobiles, la réponse la plus répandue est celle du protocole de routage. Les protocoles de routage permettant de garantir une découverte et une mise en place de routes efficaces en termes de continuité et de bande passante. Plusieurs travaux se sont attachés à développer des algorithmes permettant de trouver la meilleure bande passante entre deux points comme [86] [81] [46]. De plus, ces protocoles sont pour la plupart dérivés de protocoles connus dans le domaine des MANETs comme étant faible consommateur d'énergie tels que AODV [45] et DSDV [61].

Cependant, bien que ces protocoles améliorent la QdS des réseaux, seuls, ils ne suffisent pas à répondre aux nouveaux intérêts des utilisateurs pour des applications de plus en plus riches (audio, vidéo) et personnalisées. [85] met en exergue que le routage seul ne garantit pas la qualité d'une application. Il faut à présent prendre en compte le type des applications pour les ajuster selon qu'il s'agisse de données temps-réel ou non en modifiant la taille des buffers de communications par exemple. L'application peut aussi influencer le rôle d'un périphérique : relais, capture, traitement, etc.

A partir de ce constat, de nombreux travaux ont proposé des architectures logicielles pour gérer la QdS de ces systèmes contraints et mobiles. De telles approches permettent d'aborder plus globalement la problématique de la prise en compte de l'application dans la mesure de la QdS et de proposer de nouvelles adaptations, notamment structurelles, qui, associées aux protocoles de routage, améliorent de façon conséquente la QdS des systèmes. Parmi ces travaux, nous présentons Qinna, QuA et son extension QuAmobile et Kalinahia.

2.6.3.1 Qinna

Qinna [77] [76] est une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles. Qinna propose d'utiliser les composants pour construire des applications destinées aux systèmes embarqués mobiles et d'y intégrer la gestion de la QdS. Pour cela Qinna se base sur sept principes dont les contrats de QdS,

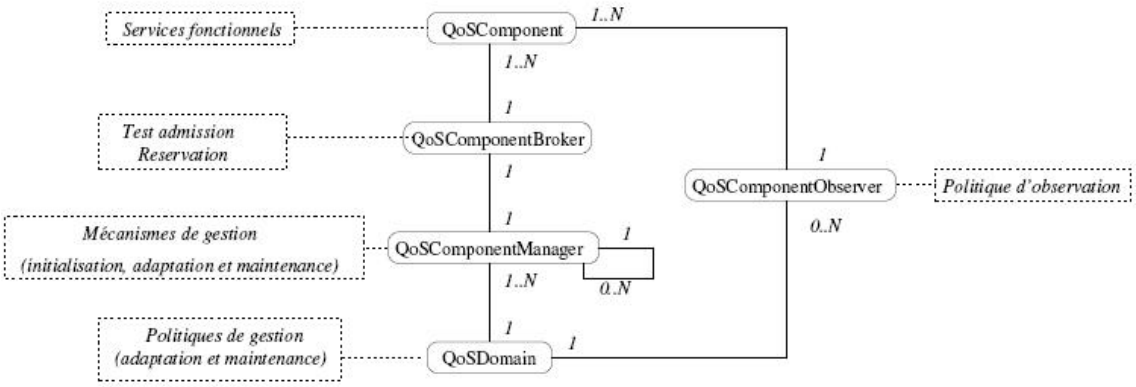

FIGURE 2.7 – Architecture de Qinna

VIDEO				
QdS objectif	Contrainte locale	QdS requises		
		DECODEUR	THREAD	MEM
BON	GRAND	25 i/s	20%	30 ko
MAUVAIS	PETIT	10 i/s	20%	30 ko

FIGURE 2.8 – Exemple d'une table de mapping pour l'utilisation du composant Video

les conteneurs de QdS et l'identification des composants à l'exécution. Son architecture repose sur cinq composants représentés dans la figure 2.7.

Les QoSComponents s'occupent de la partie fonctionnelle du système alors que les autres implémentent les préoccupations de QdS à différents niveaux. Le QoSDomain est une entité de haut niveau d'abstraction qui encapsule les autres QoSComponents. Il est l'unique point d'entrée de demande de gestion de QdS. Le QoSComponentObserver, le QoSComponentManager et le QoSComponentBroker collaborent afin de mettre en place des politiques d'adaptation.

Les adaptations de contrats peuvent aboutir à deux actions : la dégradation ou l'amélioration d'un contrat, respectivement la baisse ou l'augmentation du niveau de QdS. La mise en place d'un contrat repose sur la collaboration des QoSComponentManager et des QoSComponentBroker de chaque composant entrant dans la spécification de contrat. Cette collaboration est dirigée par une table de mapping qui décrit les dépendances entre les composants et leurs niveaux de QdS requis par rapport au niveau de QdS du contrat (figure 2.8).

Qinna permet une gestion efficace de la QdS des composants et notamment une gestion fine des ressources physiques telles que la charge CPU et la mémoire. En revanche, les adaptations que cette architecture propose sont limitées, on ne peut qu'augmenter ou baisser la QdS en ajustant les composants mais la question de la restructuration de l'application n'est pas abordée. En effet, grâce à la reconfiguration structurelle, il est tout à fait possible de maintenir un niveau de qualité de service grâce à des actions simples comme le remplacement, l'ajout ou la suppression de composants par exemple.

2.6.3.2 QuA

QuA [23] est une architecture pour la gestion de la qualité de service dans les applications distribuées à base de composants. Elle vise particulièrement les applications orientées multimédia. QuA repose sur deux concepts : les composants spécifiques de gestion de QdS et le *service planner*. Ces deux concepts permettent respectivement à QuA de viser deux objectifs : tout d'abord pouvoir composer dynamiquement les applications et ensuite pouvoir gérer la QdS des composants de l'application.

Les composants spécifiques de gestion de QdS renferment toutes les spécifications de QdS de l'utilisateur et les spécifications des ressources requises et fournies par l'application. Le tout est réalisé par des fonctions d'utilité qui calculent la qualité relative de la composition par rapport à la qualité parfaite. Chaque application est centrée autour d'un *service planner*. Le *service planner* prend en charge toutes les opérations servant à fournir les informations nécessaires à la composition de l'application. Ainsi il fournit les dépendances de tous les composants et leurs paramètres de configuration. Il fournit également toutes les caractéristiques de QdS de l'application et le modèle de QdS permettant le calcul des fonctions d'utilité. Enfin il fournit une correspondance de QdS, identifiant le niveau de QdS de chaque composant. Le *service planner* peut alors choisir les composants de l'application et les déployer. Une fois que l'application est déployée, les composants spécifiques surveillent le niveau de QdS de l'application et réitèrent le processus à chaque changement.

QuA montre l'intérêt de décrire les caractéristiques des composants d'une application lorsqu'on souhaite gérer sa QdS. Le *service planner* évalue toutes les configurations possibles d'un service selon les critères de QdS de chaque composant décrit. En revanche, la problématique du déploiement n'est pas abordée.

Les auteurs proposent également une extension de QuA pour les applications mobiles appelée QuAMobile [6]. QuAMobile, tout comme QuA, repose sur un service planner ainsi que sur un modèle de données combinant contexte et ressource assurant à l'intergiciel une description complète des informations permettant d'appliquer les configurations les plus adaptées à l'environnement. Les applications y sont modélisées selon des types de services. Lors d'une reconfiguration, l'intergiciel doit alors choisir parmi les alternatives possibles pour chaque type de service en fonction des résultats d'une fonction d'utilité. L'approche de QuAMobile est une approche intéressante par l'utilisation d'un modèle de données mixte et générique, basé sur le profil UML pour la QdS de l'OMG cependant, elle semble ne concerner que des adaptations locales contrairement à QuA.

2.6.3.3 Kalinahia

Kalinahia [43] [44] est une architecture logicielle permettant la prise en compte de la qualité de service dans les applications multimédia réparties. Cette approche basée composant vise deux objectifs : aider la conception des applications par une méthode de conception des applications intégrant la QdS et proposer une plate-forme d'exécution et d'adaptation de ces applications.

La méthode de conception proposée est basée sur des graphes. Un graphe de flots de contrôle permet de représenter les spécifications fonctionnelles fournies par le concepteur.

C'est un graphe de précédence à gros grain qui sert de base au graphe fonctionnel. Sur ces arêtes sont notées des conditions de parcours du graphe pour identifier les différentes décompositions utilisables. Le graphe fonctionnel représente le modèle fonctionnel de l'application. Il raffine le graphe de flots de contrôle et définit, à grain fin, les différentes décompositions fonctionnelles de chaque fonctionnalité fournie par l'application. A partir de ce graphe sont construits les graphes de configuration. Un graphe de configuration décrit la composition et l'implantation d'une configuration de l'application. Il précise les composants utilisés, leur localisation et leurs connexions. A partir de ce graphe, les auteurs proposent d'intégrer les informations de QdS pour construire le graphe d'évaluation. Sur chaque nœud et chaque arête sont notées les influences des composants et des connexions sur la QdS comme par exemple le temps de traitement, le débit, la taille d'une image, la cadence d'une vidéo, etc.

Toutes ces informations sont traitées par une fonction d'évaluation contenue dans la plate-forme d'adaptation. Elle est définie selon deux critères : le critère intrinsèque et le critère contextuel. Le premier reflète l'adéquation de la configuration avec les préférences de l'utilisateur tandis que le second reflète l'adéquation du service aux contraintes d'exécution.

L'adaptation des applications est réalisée par une plate-forme distribuée qui se base sur le principe de configuration représenté dans la figure 2.9.

Lorsque la plate-forme reçoit un évènement de reconfiguration, elle opère tout d'abord un tri parmi les ensembles de configurations selon des heuristiques de choix afin de ne sélectionner que les configurations pertinentes à étudier. Puis elle évalue la QdS de chaque configuration au moyen de la fonction d'évaluation, choisit la configuration qui offre la meilleure QdS, et la déploie.

La perception du contexte est réalisée par les conteneurs qui encapsulent les composants et les connexions de l'application. Cet ensemble de conteneurs distribués permet de disposer d'une vue d'ensemble de l'application et du contexte.

Les reconfigurations sont répercutées sur l'application de façon transparente par des modifications élémentaires qui sont l'ajout, la modification et la suppression de composants et de connexions. Ces reconfigurations suivent les principes de proximité et de plasticité qui permettent de sélectionner les configurations dont l'impact sur la perception par l'utilisateur du service rendu sera le plus faible possible.

Kalinahia est une approche de reconfiguration des applications distribuées qui vise particulièrement les applications multimédias. Elle propose une méthode de conception permettant d'intégrer la QdS et une plate-forme d'adaptation distribuée sensible au contexte. Le processus d'adaptation réside en une fonction d'évaluation de la QdS qui vise à trouver une configuration qui offre le meilleur compromis entre le respect des préférences des utilisateurs et le respect des contraintes d'exécution. Cette approche offre une base de travail intéressante pour la gestion de la QdS. De plus, contrairement à d'autres approches qui fournissent des adaptations prédéfinies, Kalinahia permet, grâce à des heuristiques de choix, de construire une configuration à partir des dépendances décrites dans les graphes. Cette approche n'est cependant pas suffisante lorsqu'il s'agit d'utiliser des périphériques contraints. Ni la méthode de conception, ni la plate-forme ne proposent de prendre en compte les contraintes matérielles de ressources dans le déploiement de l'application.

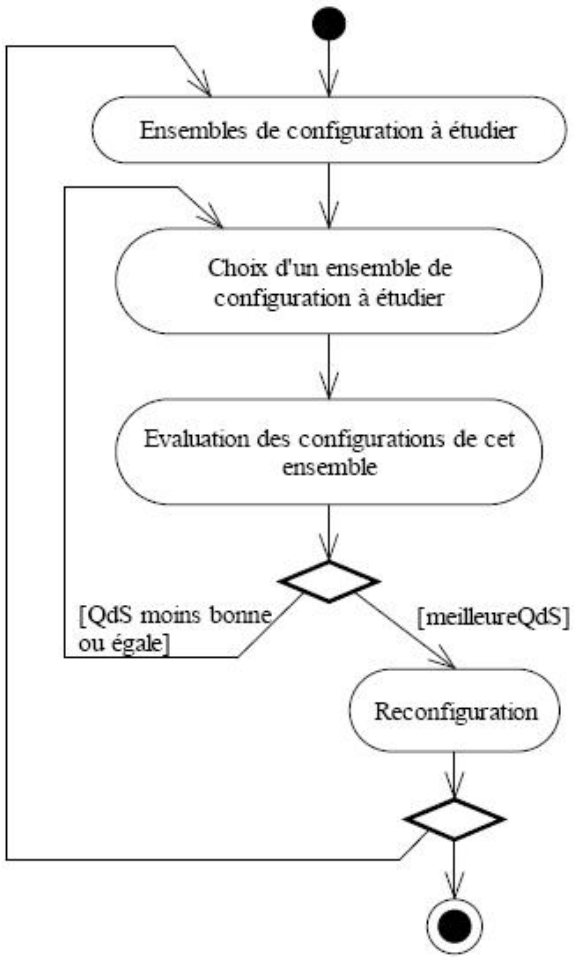

Figure 2.9 – Principe de choix d'une configuration par Kalinahia

2.6.4 Synthèse

Le tableau 1 synthétise les différentes approches selon des propriétés communes. Ce tableau permet de mettre en évidence les points de convergence et de divergence des différentes approches notamment dans le choix de l'évaluation des applications, le choix des reconfigurations et la mise en œuvre du déploiement.

Les approches que nous avons exposées dans ce paragraphe montrent différentes façons de considérer l'adaptation structurelle des applications : canevas logiciel, intergiciel, plate-forme d'exécution. Nous retiendrons de ces différentes approches l'importance de la flexibilité permettant d'agir sur la structure de l'application. Le paradigme composant paraît être une solution intéressante pour assurer cette propriété.

Un composant est défini comme une entité logicielle de composition qui répond à un modèle. Il peut être décrit en vue de son utilisation par un tiers pour la composition. La composition se fait sans modification du composant et l'accès au composant se fait au travers d'interfaces. Enfin, il possède des dépendances de contexte et son déploiement se

fait de manière indépendante [73][51][32].

Une application est alors représentée par un assemblage de composants qu'il est possible de modifier par des opérations élémentaires telles que l'ajout et la suppression de composants et de connexions entre les composants [75][77][8].

Les approches convergent également sur le fait de modéliser le contexte d'exécution, c'est à dire les caractéristiques des composants et des ressources, utilisées pour le choix des reconfigurations. Cette identification permet pour certains de réduire le nombre de configuration à étudier en vue d'une reconfiguration [64][65][59]. [64] propose dans ce sens une méthode de conception permettant d'aider le concepteur de l'application à identifier tous les composants et toutes les informations de contexte à définir pour la reconfiguration des applications. Nous retiendrons cette démarche pour la conception de nos applications.

En revanche, les approches divergent dans le choix des reconfigurations, le choix de déploiement et le déploiement lui-même. Le choix des reconfigurations peut être prédéfini ou construit dynamiquement. Prédéfini signifie que des configurations sont associées à une situation de contexte donnée. Lorsqu'une situation de contexte est détectée, le système ne peut pas choisir d'autres configurations que celles spécifiées. Construit signifie que les configurations sont construites à l'exécution et déclenchées par un changement de contexte. Des configurations peuvent être prédéfinies mais ne sont pas associées à un contexte donné. La limitation des choix de reconfiguration est nécessaire puisque le problème de l'optimisation de l'adaptation est un problème connu comme étant NP-complet. Cependant, utiliser des choix de reconfiguration prédéfinis limite fortement la réponse de l'application face aux changements du contexte. En effet, une telle limitation (parfois un seul choix possible) peut entrainer la rupture du service si toutes les conditions d'identification du contexte ne sont pas réunies. Par exemple, Aura peut suspendre l'exécution d'une tâche en attendant que toutes les conditions requises soient réunies. Dans cette thèse, nous traitons de la qualité de service des applications dont la continuité est la principale propriété à fournir quelle que soit la situation.

Pour pouvoir répondre à tous les changements de contexte, nous souhaitons accorder à la plate-forme de reconfiguration le plus de choix possibles pour pouvoir réagir. C'est pourquoi, pour une meilleure dynaminicité, nous choisissons à la volée les reconfigurations.

Les différentes approches divergent également sur le processus d'évaluation des reconfigurations. Parmi les différents processus nous retrouvons les fonctions d'utilité, les heuristiques de choix, les contrats et le tissage d'aspect. Les fonctions d'utilité et les heuristiques sont les processus les plus utilisés. Elles permettent une évaluation multicritères et dynamique. Le tissage d'aspect permet également une évaluation multicritère mais son utilisation requiert la définition de correspondances d'opérateurs qui limite la dynamicité. Enfin les contrats forment un cadre statique dont le résultat ne peut être que satisfait ou non. Etant donné que nous nous situons dans une finalité d'offrir la meilleure qualité de service pour une application, l'approche par fonction d'utilité ou par heuristique parait la plus adaptée.

Enfin nous constatons que la prise en compte du contexte dans le déploiement des reconfigurations est peu considérée. Seul [30] propose une prise en compte du contexte du périphérique en vue d'un déploiement. Cependant, celle-ci ne considère que l'impact des composants. Or les transmissions réseau sont très coûteuses énergétiquement et, dans le

domaine des périphériques contraints, la prise en compte de leur impact lors d'un choix de déploiement est critique.

Dans le paragraphe qui suit, nous explorons différentes approches de gestion du contexte.

Table 1 – Synthèse des approches de reconfiguration des applications distribuées

	Unité de composition	**Choix de déploiement**	**Déploiement**	**Reconfiguration**	**Modèle de contexte**	**Type de contexte**
Aura	Tâche		Non contextuel	Prédéfinie	*aura*	Utilisateur, Utilisation et Exécution
WComp	Services et assemblages de composants	Tissages d'Aspects d'Assemblage	Non contextuel	Construite	Aspect d'Assemblage	Utilisateur, Environnement physique et Exécution
MUSIC	Composant	Fonction d'utilité	Non contextuel	Prédéfinie	Ontologie	Utilisateur, Exécution (ressources)
CADeComp	Composant	Contrat et règles	Non contextuel	Prédéfinie		Environnement physique, Fonctionnel
AxSeL	Service composant	Heuristique multi critères	Contextuel	Construite	Simple	Utilisateur, Exécution
Qinna	Composants	Contrats	Non contextuel	Prédéfinie par mapping	Simple	Utilisateur, Exécution, Fonctionnel
Qua	Composants	Fonction d'utilité	Non contextuel	Construite		
Kalinahia	Composants	Fonction d'évaluation et heuristiques	Non contextuel	Construite	Simple	Utilisateur, Exécution

2.7 La gestion du contexte dans les applications pervasives

Les informations issues de l'environnement physique et matériel créent un Contexte pour l'interaction entre les humains et les services informatiques. Le Contexte est constitué de toute information qui caractérise une situation en relation avec les interactions entre les utilisations, les applications et l'environnement physique. Pour rendre les applications sensibles au contexte, il est nécessaire de fournir des outils permettant de le capturer, le comprendre et le traiter pour pouvoir influer sur le comportement des applications. Trois champs d'action sont définis :

- le support informatique
- les périphériques et le réseau
- l'utilisateur et l'environnement physique

2.7.1 Context Toolkit

Le Context Toolkit [66] fait partie des premiers travaux dans le domaine de la gestion du contexte dans les systèmes pervasifs. [66] s'inspire du domaine des IHM pour proposer un canevas logiciel d'aide à la conception. En effet, de la même manière que nous disposons de "GUI-builders" pour concevoir des interfaces graphiques, Context Toolkit propose des outils de conception logicielle pour les applications sensibles au contexte.

Le canevas regroupe cinq fonctionnalités représentées dans la figure 2.10. Tout d'abord la capture est réalisée par les *Context widgets*. Ces objets fournissent à l'application un accès au contexte tout en cachant la complexité de l'acquisition de ce dernier. Ensuite il fournit un *Interpreter*. L'*Interpreter* implémente l'abstraction d'une interprétation du contexte. Il permet de relever le niveau d'abstraction d'une information de contexte. Par exemple au lieu de fournir une information de température sous la forme "40°", l'*Interpreter* fournit l'information de température "chaud". Il est capable d'admettre en entrée une ou plusieurs informations contextuelles pour en produire une nouvelle de plus haut niveau. L'*Aggregator* assure la médiation des informations avec l'application. Il permet à l'application d'envoyer une unique requête pour consulter plusieurs informations de contexte. L'*Aggregator* se charge de collecter les informations demandées depuis les *Context widget*. Les *Services* sont responsables de contrôler l'action de l'application sur le contexte. Enfin le *Discoverer* est chargé de maintenir à jour l'annuaire des objets disponibles et de décrire leurs propriétés comme le langage, le protocole de communication, le nom de la machine sur laquelle ils sont disponibles, etc.

Les applications contruites par le Context Toolkit sont des applications qui interrogent le contexte, ce qui sous-entend qu'elles sont auto-adaptables ou qu'elles s'inscrivent dans une finalité d'adaptation de présentation et de contenu plutôt que d'adaptation structurelle. L'application est alors en lien étroit avec le contexte. Ce sont des applications sensibles au contexte.

Dans cette thèse, nous choisissons que les applications n'aient pas conscience du contexte qui les entoure. Lors d'un changement de contexte, ce ne sont pas elles qui raisonnent et décident de l'adaptation à effectuer. Nous souhaitons que la gestion du contexte et des adaptations reste totalement transparente aux applications.

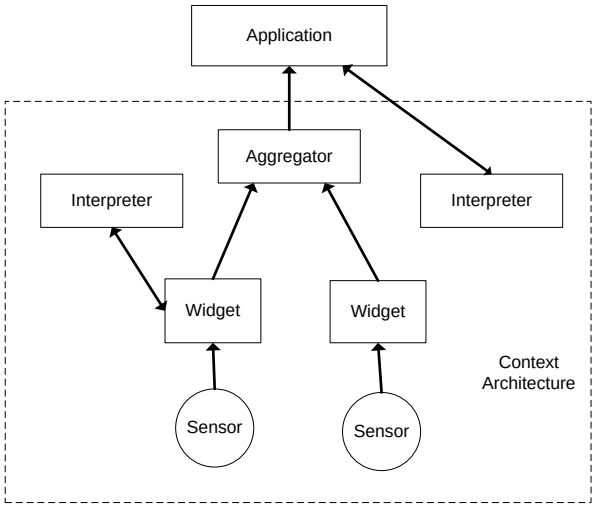

FIGURE 2.10 – Architecture du Context Toolkit

2.7.2 SECAS

Le projet SECAS [13] propose une plate-forme de conception et de déploiement d'applications sensibles au contexte d'utilisation afin de répondre à deux problématiques soulevées par le développement de telles applications : comment concevoir une architecture garantissant l'adaptation dynamique au contexte au cours de l'exécution et comment concevoir l'application elle-même pour qu'elle s'adapte au contexte ? Il propose également des moyens d'adaptation de tout le comportement d'une application au contexte d'utilisation. Les auteurs définissent le contexte comme l'ensemble des paramètres externes à l'application pouvant influer sur son comportement en définissant de nouvelles vues sur ses données et ses services. Ils précisent également que ces paramètres n'étant d'aucune utilité à l'utilisateur final, leur gestion doit lui être transparente. La gestion du contexte repose sur l'identification de situations contextuelles formées par ces paramètres et qui sont définies selon cinq axes, appelés facettes : le mode de communication, l'utilisateur, le terminal, la localisation et l'environnement.

L'architecture de SECAS est constituée d'un module de gestion du contexte et d'un module d'adaptation représentés dans la figure 2.11.

Le module de gestion du contexte s'inspire de celui du Context Toolkit de [66] et se compose de quatre entités : le capteur de contexte, l'interpréteur, un historique de contexte pour le stockage et le gestionnaire de contexte qui assure la diffusion vers le module d'adaptation. Le modèle utilisé pour la description du contexte est basé sur des schémas XML (listing 2.1) qui respectent les grammaires standards telles que RDF et CC/PP [34]. Le module d'adaptation fournit trois services d'adaptation : l'adaptation de contenu, l'adaptation de présentation et l'adaptation de comportement.

```
<contexteProfile ID="U17T355008002783631">
  <contextFacet name="terminal">
    <contextParameter name = "type" type="static">cldc
```

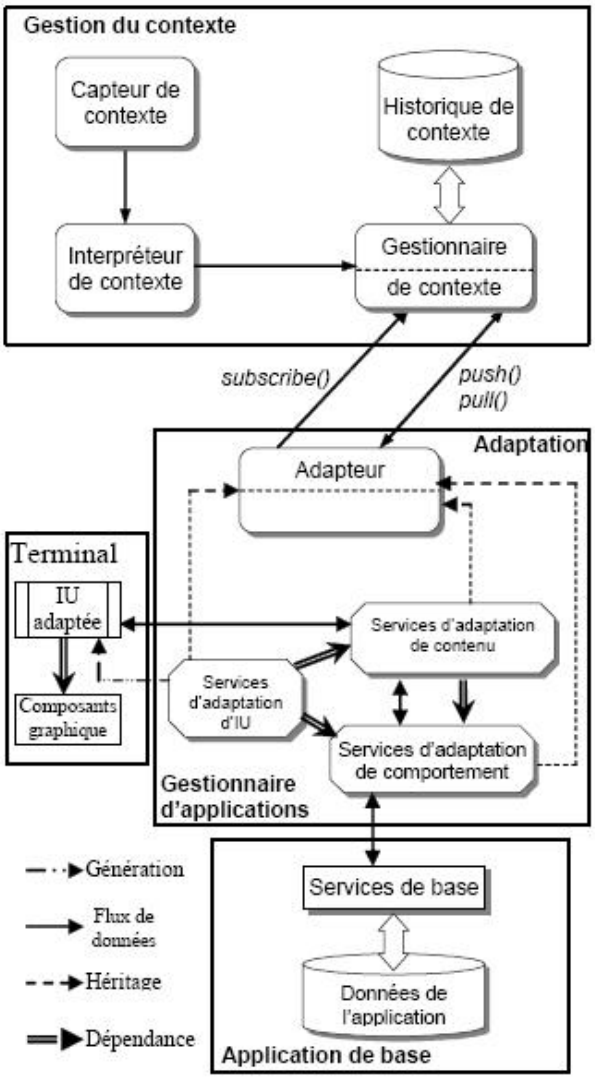

Figure 2.11 – Architecture générale de SECAS

```
</contextParameter>
<contextParameter name = "model" type="static">NOKIA6600
</contextParameter>
<contextParameter name = "serialNumber" type="static">35098987365
</contextParameter>
<contexteSubCategory name="hardwarePlatform">
  <contextParameter name = "totalMemory" type="static">2
  </contextParameter>
  <contextParameter name = "availableMemory" type="dynamic">1.3
  </contextParameter>
  <contextSubCategory name = "screesize">
    <contextParameter name = "width" type="static">208
    </contextParameter>
```

```
        <contextParameter name = "height" type="static">320
        </contextParameter>
      </contextSubCategory>
    </contextSubCategory>
    <contexteSubCategory name="softwarePlatform">
      <contextParameter name = "Audio" type="static">
      wav, midi, mp3, 3gp</contextParameter>
      <contextParameter name = "Images" type="static">png, bmp
      </contextParameter>
    </contextSubCategory>
    <contexteSubCategory name="api">
      <contextParameter name = "virtualMachine" type="static">
      Monty 1.0 VM</contextParameter>
    </contextSubCategory>
  </contextFacet>
  <contextFacet name="network">
    <contextParameter name = "connectionType" type="static">
    GPRS</contextParameter>
    <contextParameter name = "bandWidth" type="static">33
    </contextParameter>
  </contextFacet>
  <contextFacet name="userProfile">
  ...
  </contextFacet>
  <contextFacet name="location">
  ...
  </contextFacet>
```

Listing 2.1 – Exemple d'un profil de contexte dans SECAS

SECAS considère les applications comme un graphe de dépendances où chaque noeud représente un service. L'adaptation de comportement consiste à transformer le graphe en un autre graphe de dépendances de services adaptés. Pour cela, l'adaptateur opère un choix parmi les différentes versions offertes pour un service, guidé par les paramètres de contexte. L'adaptation de contenu consiste à modifier les propriétés des données présentées à l'utilisateur intéressé. L'adaptation de présentation, quant à elle, consiste à générer les interfaces graphiques suivant les caractéristiques du terminal. Pour cela, l'adaptateur se base sur la description des composants graphiques et des composants physiques.

L'architecture proposée par SECAS respecte les propriétés de séparation des préoccupations, de flexibilité, d'identification des composants et du contexte. La sensibilité au contexte dans SECAS ne considère que le contexte d'utilisation c'est à dire que les informations qui peuvent être capturées par des capteurs physiques. L'adaptation de comportement est basée sur le polymorphisme des services et n'offre pas la possiblité de modifier la composition des applications. De plus, bien que SECAS prenne en compte la caractéristique des périphériques contraints en proposant d'utiliser les services web et le protocole de communication XML-RPC pour éviter les problèmes de lenteur et de lourdeur de traitement, la problématique du déploiement des services de l'application n'est pas abordée.

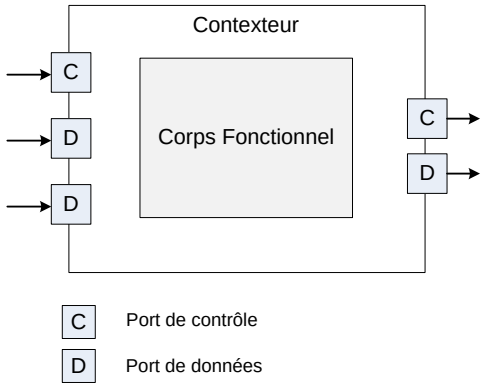

Figure 2.12 – Architecture d'un Contexteur

2.7.3 Le Contexteur

Le Contexteur [63] est une abstraction logicielle qui fournit la valeur d'une variable du contexte système. La finalité du Contexteur est de permettre à un système interactif de fournir des services contextualisés à un utilisateur donné engagé dans une activité à un instant t. Autrement dit, le Contexteur doit permettre de fournir un service adapté aux conditions de contexte.

Toutes les données de contexte du système sont acquises ou calculées par le Contexteur. Il a une structure semblable à celle d'un composant : des ports d'entrée et de sortie et un corps fonctionnel (figure 2.12).

Les ports sont de deux types : port de contrôle et port de données. Les ports de contrôle permettent de modifier le fonctionnement et les paramètres du contexteur. Les ports de données correspondent aux données que le contexteur traite et fournit. Chaque donnée est associé à une méta-donnée qui exprime sa qualité. Le corps fonctionnel implémente une fonction particulière de traitement de contexte dont la classification est semblable à celle définie dans [66] : le contexteur élémentaire qui encapsule un capteur physique, le contexteur à mémoire qui mémorise un historique de données, le contexteur à seuil qui teste le franchissement du seuil, le contexteur de traduction qui modifie la présentation, le contexteur de fusion qui produit une donnée à partir de plusieurs pour augmenter la qualité et enfin le contexteur d'abstraction qui produit des données de plus haut niveau d'abstraction.

Les différents contexteurs peuvent être assemblés de façon hiérarchique pour former des politiques de gestion du contexte et ne fournir à l'application que les données pertinentes dont elle a besoin.

Ces travaux sont un très bon complément de ceux de [66]. Bien que cette approche se limite au seul contexte système, elle permet une définition des politiques de gestion du contexte de façon simple et uniforme par l'utilisation d'un cavenas unique : le contexteur. La notion de méta-donnée est intéressante lorsque l'on veut tirer partie de tous les composants capables de traiter le contexte notamment dans les domaines comme les systèmes

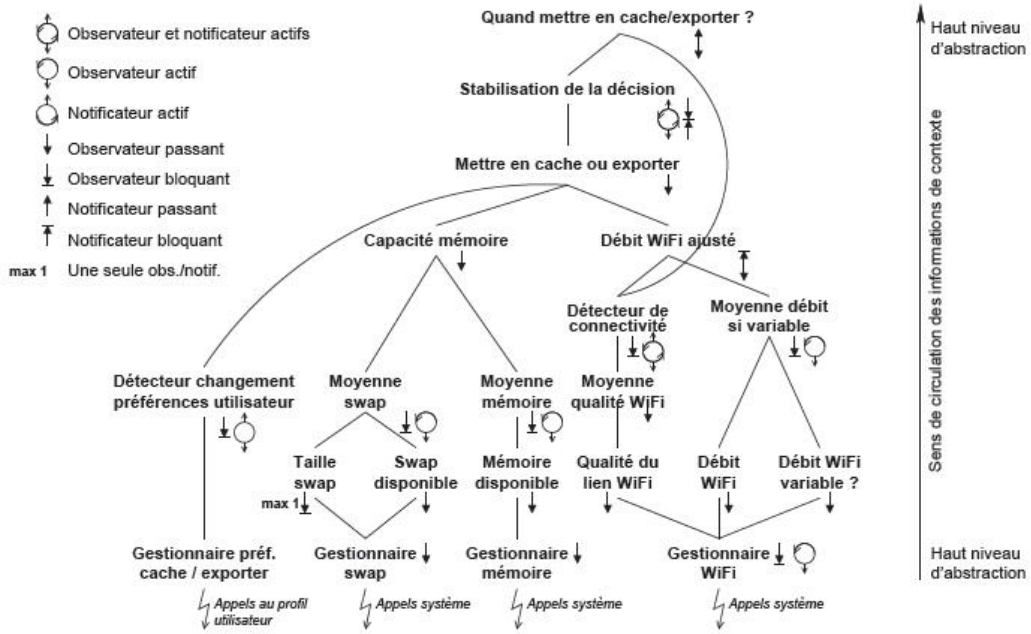

FIGURE 2.13 – Exemple d'une composition de nœuds de cocntexte par COSMOS

pervasifs, où l'environnement n'est pas connu a priori. De plus l'assemblage hiérarchique des contexteurs permet la flexibilité du gestionnaire de contexte et fournit la possibilité de construire des politiques de gestion du contexte à la volée. Enfin la séparation entre le contexte et l'application est plus nette et permet de l'intégrer facilement à un intergiciel d'adaptation des applications.

2.7.4 COSMOS

COSMOS [16] est un canevas logiciel basé composants pour la gestion des informations de contexte dans les applications sensibles au contexte. Les auteurs définissent trois propriétés auxquelles doit répondre un gestionnaire de contexte selon eux. Premièrement, il doit être centré sur l'application et l'utilisateur pour founir des informations qui soient faciles à interpréter. Deuxièmement il doit être construit à l'aide d'éléments composés plutôt que programmés ce qui facilite le développement et la réutilisation, ainsi que la flexibilité. Troisièmement il doit être performant en contrôlant et en minimisant l'utilisation des ressources pour son propre fonctionnement et éviter d'entraver celui de l'application. Pour cela les auteurs définissent une entité logicielle de composition : le nœud de contexte.

Un nœud de contexte est une information contextuelle modélisée par un composant logiciel. Ces nœuds de contexte peuvent être assemblés de façon hiérarchique afin de composer et constituer des politiques de gestion du contexte (figure 2.13).

Ils possèdent plusieurs propriétés : actif ou passif, observateur ou notifieur, passant ou bloquant. Ces propriétés permettent de construire des politiques de gestion du contexte de plus haut niveau d'abstraction que dans les architectures traditionnelles utilisant le simple

mécanisme publish/subscribe. Les traitements implémentés par les nœuds de contexte sont semblables à ceux proposés dans [66] : opérateur élémentaire pour la collecte, opérateur à mémoire pour l'agrégation, la fusion, la traduction de données, opérateur d'abstraction pour fournir une information de plus haut niveau, opérateur à seuil pour la surveillance ou encore évaluateur de profil utilisé notamment pour la surveillance de l'énergie, de l'espace mémoire ou d'une durée prévisible de connexion.

La réalisation des politiques de gestion du contexte de COSMOS réside dans quatre patrons de conception d'architecture : Composite, Patron de méthode, Poids-mouche, Singleton. Ces patrons assurent respectivement la composition générique des nœuds de contexte, la définition générique du comportement des nœuds, le partage des nœuds et la centralisation de la gestion des ressources système. L'originalité de l'utilisation de patrons de conception est d'utiliser les notions d'architecture et de composant pour traduire les politiques directement au niveau langage. Le travail du développeur est ainsi allégé et la génération automatique de nouvelles politiques de gestion du contexte à l'exécution devient possible.

La deuxième particularité de COSMOS réside dans sa proposition de gérer l'utilisation des ressources pour le fonctionnement de son gestionnaire. Pour cela chaque nœud de contexte doit enregistrer la tâche qu'il accomplit auprès d'un gestionnaire d'activités qui, selon son paramétrage, crée une activité par tâche ou une activité par nœud ou une activité par hiérarchie ou partie de hiérarchie.

COSMOS est un modèle de contexte qui propose d'encapsuler chaque information de contexte dans une entité appelée nœud de contexte. Il inclut un outil logiciel basé composant permettant de faciliter la construction de politiques de gestion du contexte dans les applications sensibles au contexte qui peut facilement se placer au dessus d'un intergiciel d'adaptation. Les nœuds de contexte sont construits sur le même modèle de composant ce qui facilite la réalisation des politiques par composition de nœuds. La diversité des fonctionnalités des nœuds de contexte apporte un avantage considérable dans la gestion du contexte. La composition des noeuds permet un traitement des informations et, par extension, un tri des informations dans chaque nœud pour n'aboutir en bout de chaîne qu'aux informations véritablement utiles à la décision de reconfiguration. Un tel système permet de soulager la plate-forme d'une grande quantité de calculs nécessaires à la prise de décision et, par conséquent, réduit la complexité de la plate-forme d'adaptation.

Les tests de mise en œuvre de COSMOS peuvent cependant être critiqués. Dans ces tests, COSMOS est implanté au moyen de composants FRACTAL et la machine utilisée est une machine relativement puissante de type ordinateur portable qui ne reflète pas la réalité des systèmes pervasifs. Il serait intéressant de tester si sa mise en œuvre est réaliste sur des périphériques tels que les téléphones portables, les PDA ou encore les capteurs.

2.7.5 Synthèse

Le tableau 2 synthétise les différentes approches étudiées et met en lumière leurs points de convergence et de divergence.

Le Context Toolkit [66] fait partie des premiers travaux sur la gestion du contexte dans les applications sensibles au contexte. Il propose un canevas logiciel pour aider le

Table 2 – Synthèse des approches de gestion du contexte

	Unité de composition	**Propriétés**	**Opérateur**	**Politiques**	**Type de contexte**	**Adaptation**
Context Toolkit	Objet	Notification	Capture, Interprétation, Agrégation	règles simples	Utilisateur, Environnement	Actionneurs
SECAS	Services		Capture, Interprétation, Historique	règles simples	Utilisateur, Terminal, Environnement, Localisation, Communication	
Contexteur	Contexteur	Observation, Notification	Capture, Mémoire, Interprétation, Fusion, Traduction, Seuil	Assemblages de contexteurs	Système	
COSMOS	Nœuds de contexte (composant)	Actif/Passif, Observation/Notification, Bloquant/Passant	Capture, Mémoire, Interprétation, Fusion, Traduction, Seuil	Patrons de conception	Environnement physique, Fonctionnel	

concepteur de l'application à modéliser le contexte en vue d'une utilisation facilitée par l'application. Il propose des éléments de base de la gestion du contexte comme la capture, l'interprétation et l'agrégation. Ces éléments forment les opérateurs implémentés par les gestionnaires de contexte. Leur liste a été étendue avec [63] [16] et propose des opérateurs de mémoire, de seuil, etc. Ceci permet une séparation de préoccupations et la réutilisation des opérateurs de contexte.

Les approches convergent également sur la propriété des éléments de contexte : ils peuvent observer ou notifier. Le premier attend qu'un tiers lui demande une information alors que l'autre diffuse l'information à chaque changement d'état. Nous retiendrons ces deux propriétés pour la conception de notre gestionnaire de contexte.

Parmi les politiques de contexte, nous retiendrons la construction des politiques par assemblages d'opérateurs de contexte. Cette solution nous paraît simple et facile d'implantation. Elle a l'avantage de ne fournir en bout de chaîne que l'information nécessaire au processus d'adaptation en vue d'une reconfiguration. De plus, ainsi distribuées, les politiques de contexte réduisent la complexité de la plate-forme et offrent une vue globale du contexte.

Enfin, la plupart des approches ne fournissent qu'une partie du contexte qui influence réellement l'application. La plupart oublie de prendre en compte les spécifications de l'application elle-même qui peuvent mener à des reconfigurations qui ne sont pas satisfaites. Nous devons alors proposer une classification du contexte qui permettent de prendre en compte le contexte dans sa totalité afin d'adapter les applications le plus finement possible.

2.8 Conclusion

Après cette analyse des différents approches pour la gestion des applications mobiles et sensibles au contexte, nous concluerons en présentant toutes les propriétés que doit posséder une architecture de reconfiguration des systèmes sensibles au contexte nous permettant de répondre à la problématique suivante : garantir aux applications la meilleure qualité de service malgré les changements de contexte. Pour cela nous avons défini les propriétés suivantes :

Vue globale Afin d'avoir une vue globale de l'application, la plate-forme doit être distribuée sur tous les périphériques de l'application.

Identification du contexte Afin de disposer de toutes les informations de contexte nécessaires au choix d'une reconfiguration, notre approche doit proposer une classification permettant de couvrir le contexte influençant le fonctionnement de l'application dans sa totalité. Notre approche doit également proposer une méthode de conception permettant de guider le concepteur dans la description des différentes entités entrant dans la composition du contexte. Ceci passe notamment par les spécifications de l'application, l'identification des composants et l'identification des périphériques.

Interopérabilité L'une des principales caractéristiques des systèmes que nous visons est la diversité des périphériques. Ces périphériques sont tous différents tant sur le plan matériel que logiciel. Nous proposons dans un premier temps une classification des périphériques. Parmi les classifications disponibles dans la littérature,

nous retenons les standards proposés par J2ME. Dans cette thèse, nous avons effectué nos recherches sur des capteurs Sun Spot [52] de Sun Microsystems. Ces capteurs sont dotés d'une machine virtuelle java, appelée Squawk, réalisée en conformité avec le standard de Java mobile, CLDC. Nous distinguons de cette manière trois types de périphériques : fixe, CDC et CLDC. Les périphériques fixes offrent des contraintes de ressources négligeables face aux deux autres types. Nous classons parmi les périphériques fixes les ordinateurs et les ordinateurs portables. Les périphériques CDC correspondent au standard Connection Device Configuration [36]. Ce sont les périphériques contraints tels que les smart-phones et certains PDA. Les périphériques CLDC correspondent au standard Connection Limited Device Configuration [37]. Ce sont les périphériques très contraints tels que les téléphones portables et les capteurs sans-fil.

Cette classification devra être prise en compte dans l'identification des composants et des périphériques afin de guider le choix des composants et le choix du déploiement d'une configuration. Cette classification devra également être prise en compte dans le déploiement de la plate-forme elle-même. En effet, un processus comme le processus de choix des composants ou de déploiement est un processus qui demande beaucoup de calcul et donc beaucoup d'énergie. Le déployer sur un périphérique très contraint peut être critique pour sa durée de vie.

Enfin les périphériques sont également différents dans leur mode de communication. Afin de pouvoir établir des collaborations entre périphériques hétérogènes, la plate-forme devra proposer un mécanisme permettant de masquer cette hétérogénéité.

Déploiement contextuel L'utilisation de périphériques contraints et très contraints nécessite de proposer des déploiements de configurations adaptés à leurs caractéristiques de ressources. Pour cela nous incluont dans la définition de la qualité de service la notion de durée de vie de l'application. En outre les considérations de mémoire et de charge CPU, nous prenons en compte la consommation en énergie des composants et des connexions. Cette prise en compte nous permet d'inclure dans le processus de choix des composants et de déploiement la notion de durée de vie.

Dynamicité Nous nous situons dans un contexte d'applications pervasives qui implique une grande réactivité au changement de contexte. Pour cela, la plate-forme de reconfiguration des applications doit pouvoir proposer des reconfigurations à la volée.

Flexibilité Enfin, pour pouvoir être reconfigurées, les applications doivent proposer une structure facilement modifiable. Pour cela nous proposons une réalisation des applications à base de composants et de connecteurs qui sont des entités logicielles facilement manipulables et qui permettent une gestion de l'application à grain fin. La plate-forme doit également proposer des outils de restructuration adaptés aux composants. Pour cela nous nous basons sur les mécanismes déjà proposés dans la littérature : l'ajout et la suppression de composants et de connexions. La flexibilité doit également être applicable à la plate-forme. En effet, en fonction du contexte, il doit pouvoir être possible d'adapter les politiques de reconfiguration. Ceci implique de fournir la possibilité de reconfigurer le processus de choix de reconfiguration de la plate-forme telle que le propose Music [64] en permettant un redéploiement de l'intergiciel, ou encore Qinna [77] en permettant la modification des contrats de QdS.

Chapitre 3

Méthode de conception

Sommaire

3.1 Introduction

L'approche d'adaptation des applications mobiles que nous proposons repose sur une réaction aux changements de contexte à trois niveaux tels que nous les avons définis dans

le chapitre précédent : *utilisateur*, *utilisation* et *exécution*. Afin de pouvoir interpréter le contexte nous avons besoin de le modéliser. Après avoir identifié les différentes sources de contexte et la nature des informations qu'elles produisent, nous proposons un modèle de contexte permettant d'offrir à la plate-forme un cadre de travail suffisant pour l'évaluation de la qualité de service de l'application.

La qualité de service quant à elle, est dirigée par la continuité de service ainsi que par les deux types de QdS que nous avons définis dans le chapitre précédent : *Utilité* et *Pérennité*. Pour son évaluation, nous proposons un modèle de qualité de service prenant en compte à la fois l'utilité de l'application et les caractéristiques de déploiement des composants sur les différents périphériques.

Ces modèles sont complétés par une démarche de développement permettant au concepteur d'identifier les informations de contexte particulières à l'application.

3.2 Modèle de contexte

Le contexte est la mesure de la QdS de l'application et par conséquent nous devons capturer toutes les informations et identifier toutes les situations afin de comparer cette mesure avec la QdS requise et pouvoir ainsi mettre en place les adaptations nécessaires. L'objectif est de pouvoir réagir aux changements de contexte en proposant des reconfigurations fournissantune structure de l'application la plus proche possible du service requis. Ceci signifie que nous souhaitons offrir à tout moment une application d'une qualité suffisante et en adéquation avec les besoins et les capacités matérielles et logicielles. Pour cela nous avons besoin d'avoir une vision globale de toutes les entités qui composent l'application et évoluent autour d'elle, à savoir une description complète et précise du contexte. D'après la classification que nous avons donnée dans le chapitre 2, nous distinguons trois types de contextes : le contexte d'utilisateur, le *contexte d'utilisation* et le *contexte d'exécution*. Ces contextes sont composés de deux types d'informations : les informations mesurables et les informations abstraites.

3.2.1 Les informations mesurables

Le contexte est en partie composé d'informations mesurables et quantifiables. Ce sont pour la majorité des grandeurs physiques représentées par des valeurs numériques comme par exemple la température, la luminosité, la taille d'un espace mémoire. Ces informations peuvent être capturées de deux façons.

3.2.1.1 Capture par l'application

L'application dispose de plusieurs moyens pour capturer des informations de contexte mesurables.

Les composants spécifiques Des composants spécifiques sont conçus spécialement afin de capturer des grandeurs physiques bien précises utiles à la détection de situations d'adaptation. Ce sont les informations relatives à l'environnement qui composent le *contexte d'utilisation*. C'est le cas des composants logiciels liés à des matériels tels que les capteurs qui permettent de relever la température, la luminosité ou le déplacement.

Les conteneurs de composants métiers et de connecteurs Nos applications sont conçues à base de composants logiciels interconnectés entre eux par des connecteurs. Ces composants et ces connecteurs sont développés à l'aide du modèle de composant OSAGAIA [10] et du modèle de connecteur KORRONTEA [9] que nous décrirons dans le chapitre 5.

Dans ce modèle, les composants métiers fournissent une fonctionnalité bien précise. Ils n'ont d'autre préoccupation que de traiter les données qu'ils reçoivent et transmettre leurs résultats. Ils sont encapsulés dans des conteneurs.

Les conteneurs permettent de ne pas mêler au code métier des composants des préoccupations non-fonctionnelles telles que les connexions avec les autres composants et la communication. Le conteneur sert d'interface entre les connecteurs et le composant qu'il renferme et s'occupe de réceptionner et diffuser les données. De la même façon les connecteurs sont des conteneurs encapsulant un composant métier dont le rôle est de faire circuler l'information localement ou par le réseau. Chacun de ces conteneurs pilote les entrées et les sorties de données au moyen de deux unités d'échange : l'unité d'entrée et l'unité de sortie. Les conteneurs de connecteurs sont dotés de buffers permettant de gérer le flux de données. Ils proposent des primitves permettant de détecter d'éventuels dysfonctionnement. Ainsi des informations telles que la saturation d'un buffer ou, au contraire, le fait que le buffer est vide, sont relevés par l'application puis rapportées à la plate-forme. Quand le buffer d'entrée est signalé comme étant saturé, ceci peut signifier que le composant ne traite pas les données assez rapidement. Un buffer vide peut, au contraire, indiquer qu'une augmentation de la QdS est envisageable.

Les cartes d'identité des composants La plate-forme peut également recueillir des informations sur les compétences techniques d'un composant. Chaque composant renferme dans une carte d'identité toutes ses propriétés techniques et notamment les propriétés sur la fonctionnalité qu'il fournit. Par exemple, pour un composant d'émission vidéo, nous trouvons le nombre d'image par seconde ou encore le nombre de couleurs de la vidéo. Les cartes d'identité seront décrites plus en détail dans la section 3.5.6.

L'application possède trois moyens de capturer les informations mesurables du contexte : les composants spécifiques, les conteneurs de composants métiers et de connecteurs et les cartes d'identité. Ces informations concernent essentiellement le contexte d'utilisation. En effet elles décrivent des informations sur l'environnement autour de l'application, sur les fonctionnalités des composants c'est à dire ce qu'ils sont capables de fournir et sur le fonctionnement de l'application. D'après la définition donnée dans le chapitre 2, il s'agit bien du *contexte d'utilisation* et *d'exécution*.

3.2.1.2 Capture par la plate-forme

La plate-forme dispose de moyens logiciels pour mesurer les ressources des périphériques. Elle peut alors récupérer les informations sur les capacités des périphériques telles que la capacité mémoire, le taux d'utilisation du CPU, la bande passante disponible et le niveau de batterie. Ces informations constituent ce que nous avons défini comme le *contexte d'exécution*.

Les informations mesurables de contexte sont représentées par des valeurs qui peuvent être comparées par des opérateurs mathématiques, et donc peuvent être encadrées par des seuils. Ces seuils permettent d'isoler des situations nécessitant une adaptation de l'application comme par exemple, lorsque le niveau de batterie d'un périphérique atteint le seuil de 10%, cela signifie qu'il va falloir délocaliser le plus possible de composants sur les autres périphériques de l'application afin de ne pas compromettre la continuité de l'application. Une telle décision est prise par la plate-forme supervisant l'application. Tout dépassement d'un seuil engendre un évènement signifiant que le contexte a évolué. La plate-forme va alors démarrer le processus d'évaluation de la qualité de service afin de trouver les adaptations adéquates et les mettre en œuvre.

3.2.2 Les informations abstraites

Jusqu'à présent nous avons pu remarqué que le contexte était composé d'un ensemble d'informations mesurables. Cependant ces informations ne représentent qu'une partie du contexte : le *contexte d'exécution* et le *contexte d'utilisation*. Le contexte d'utilisateur est, quant à lui, constitué des besoins de l'utilisateur, ce qu'il souhaite pouvoir faire avec l'application proposée. Par exemple, un utilisateur se trouve devant une œuvre au musée, il souhaite obtenir des informations sur son auteur, sa date de création, etc. L'application de visite sur son PDA lui propose un menu affichant les deux services qu'il peut utiliser : le service *Description* et le service *Guidage*. A l'aide de son stylet, l'utilisateur clique sur la widget *Description* pour signifier qu'il souhaite utiliser le service de *Description*. Nous devons alors traduire ce clic sur cette widget et transmettre l'information à la plate-forme qui décidera d'une adaptation de l'application à savoir une installation et/ou un démarrage du service demandé. L'information de demande du service *Description* n'est pas une information mesurable, elle n'est pas représentée par une valeur, elle n'est pas matérielle, elle est abstraite.

Le contexte est donc également constitué de telles informations abstraites. Contrairement aux informations mesurables, elles n'ont pas de représentation physique concrète. Pour la plupart, elles représentent un souhait, une situation ou une perception de l'utilisateur. Néanmoins ces informations font partie du contexte de l'application et la modification du contexte doit pouvoir être détectée afin d'adapter l'application en conséquence. Elles sont nécessaires pour la définition de certains évènements déclencheurs de reconfigurations comme la demande ou l'arrêt d'un service. De telles informations ne pouvant pas être capturées par des dispositifs comme les capteurs, nous devons proposer au concepteur de l'application un moyen de les décrire afin de les capturer et les interpréter. Nous réifions alors ces informations abtraites afin de les rapporter à une entité concrète. Dans le processus de modélisation du contexte, chaque information abstraite est alors réifiée par

une entité qui possède un attribut nom.

- *DemandeDescription* et *ArretDescription*
- *DemandeGuidage* et *ArretGuidage*
- *Traduction*
- *Conference*
- *Incendie*

Tout comme pour les informations mesurables, les informations abstraites sont capturées par la plate-forme.

3.2.2.1 Capture par l'application

L'application permet de capturer les interactions avec l'utilisateur par l'intermédiaire de l'interface homme-machine. Par exemple, nous associons un nom à chaque zone de l'interface où l'utilisateur peut agir par des clic de souris afin de pouvoir détecter les changements de ses souhaits et en avertir la plate-forme par un évènement. Pour cela nous établissons des règles d'association entre la zone cliquée et l'information produite par le clic :

$$clicDescription = \text{demande du service } Description \quad (1)$$

$$clicTraduction = \text{Demande de la fonctionnalité } Traduction \quad (2)$$

La règle 1 indique que lorsque l'utilisateur clique sur la zone de l'interface identifiée comme *clicDescription* cela signifie que l'utilisateur souhaite démarrer le service *Description*. La règle 2 indique quand à elle que la zone de l'interface nommée *clicTraduction* est associée à la demande de traduction du service en cours d'exécution. L'action associée consiste à transmettre cette information contextuelle à la plate-forme.

3.2.2.2 Capture par la plate-forme

La plate-forme quant à elle, capture les informations relatives à l'état de l'application, ses spécifications. Les spécifications de l'application décrivent des règles de fonctionnement sur son utilisation. Un exemple de règle est : lorsqu'un visiteur entre dans une salle du musée où se déroule une conférence, les services qu'il utilise ne doivent pas comporter de son. Afin de reconfigurer l'application du visiteur en conséquence, la plate-forme renferme des règles d'utilisation que la structure de l'application doit vérifier. Ces règles sont évaluées chaque fois qu'un évènement de contexte survient. Une conférence peut être détectée par le niveau sonore de la salle ou par la date et l'heure. Une conférence peut donc par exemple être détectée par l'ensemble d'informations formé par un niveau sonore et une localisation. On obtient l'association suivante :

$$niveau\,sonore > 70\,dB\ ET\ localisation = localisation\,visiteur \Rightarrow Conference$$

Afin d'adapter l'application du mieux possible nous avons besoin de capturer le contexte et de fournir les informations recueillies à la plate-forme dans une forme qui facilite le processus d'évaluation. Pour cela nous proposons d'utiliser un modèle de contexte.

De nombreux travaux proposent différentes façons de capturer et de modéliser le contexte. Il existe trois approches généralement utilisées pour sa modélisation et qui s'intéressent à différents niveaux de complexité.

La plus simple est un ensemble non structuré de couples attribut/valeur :

$$\{Utilisateur = \ll \text{visiteur1} \gg, Localisation = \ll \text{salle1} \gg\}$$

C'est l'approche qu'utilise par exemple [66] dans le context toolkit. Cette approche, bien que très simple à implémenter, ne permet pas d'exprimer d'informations à la sémantique riche telles que celles que nous souhaitons prendre en compte. Il n'est adapté que dans des cas où le contexte se réduit à un ensemble d'informations mesurables.

La deuxième approche est une approche proposée pour standardiser la représentation des informations dans le cas des applications Web telle que le Composite Capabilities/-Preferences Profil du W3C [18] [34]. Cette approche plus complexe et plus riche que la précédente est basée sur le Resource Description Framework, langage de description de métadonnées conçu par le W3C. Le profil CC/PP est défini comme une description des capacités des périphériques et des préférences des utilisateurs. Il est représenté selon deux niveaux. Un profil est composé d'un ou de plusieurs composants et chaque composant possède un ou plusieurs attributs. Chaque attribut est représenté par une valeur. Cette représentation permet d'identifier les composants du contexte, par exemple si ce sont des propriétés d'un périphérique, d'un utilisateur ou d'un composant logiciel.

La dernière approche est l'utilisation d'ontologies. Cette approche est la plus complète concernant le degré d'expression des informations et de leur sémantique. Elle permet notamment de gérer et d'éviter les conflits lorsqu'il s'agit d'identifier une situation bien précise. Cependant c'est aussi l'approche la plus complexe à mettre en œuvre et à implémenter dans un système. Son implémentation demande plus de temps et de ressources que les autres pour traiter les informations. Etant donné que nous utilisons des périphériques contraints et donc que les ressources sont limitées, nous devons les préserver afin de garantir la durée de vie la plus longue possible. Par conséquent nous éliminons cette aproche de gestion du contexte.

A la façon de RDF et tout comme [12], nous choisissons de modéliser le contexte de façon plus riche qu'un simple ensemble de couples attribut/valeur. En effet nous avons identifié deux types d'informations contextuelles :

- les informations mesurables. Ce sont les propriétés mesurables sur la figure 3.1
- les informations abstraites qui demandent un degré d'expression et une sémantique plus élevés. Ce sont les propriétés abstraites sur la figure 3.1

Nous définissons le contexte d'une application par un profil de contexte. Ce profil est composé de plusieurs éléments de contexte. Ces éléments sont utiles pour déterminer d'où proviennent les informations de contexte, s'il s'agit des informations d'un périphérique, d'un composant ou de l'utilisateur ou de l'application. Chaque élément de contexte est constitué d'un ensemble de propriétés mesurables et de propriétés abstraites. Une propriété mesurable est définie par un nom, une valeur, un type indiquant si elle est statique ou dynamique, une unité, une valeur maximale et une valeur minimale. Les propriétés mesurables permettent de décrire dans le détail toutes les caractéristiques physiques et

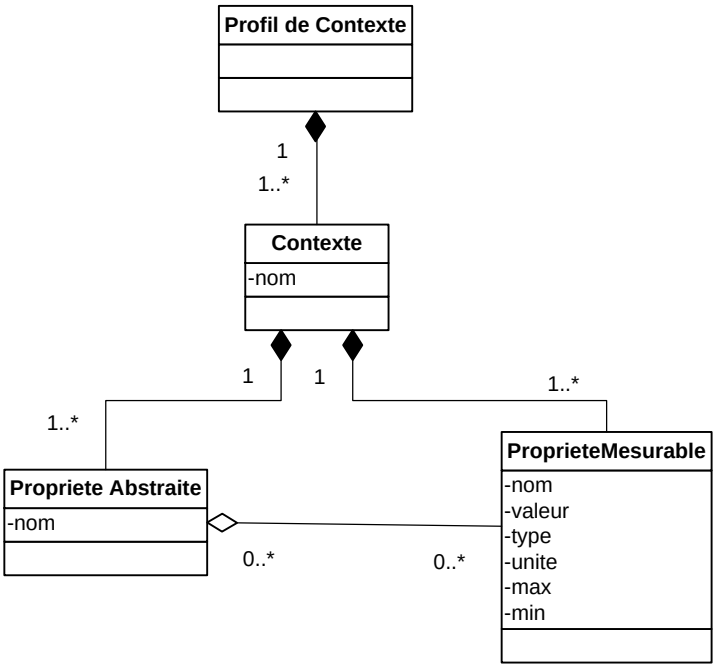

Figure 3.1 – Modèle des informations contextuelles

logicielles des périphériques et des composants logiciels. Ainsi pour les périphériques, nous décrivons toutes leurs propriétés telles que le niveau de batterie ou le niveau de charge CPU maximum par exemple ou encore le système d'exploitation utilisé. Ces informations sur l'état de fonctionnement du périphérique seront enrichies par les informations sur ses capacités techniques théoriques.

Concernant les composants logiciels, nous avons deux types de composants.

Premièrement, nous avons les composants logiciels spécifiques à la capture du contexte : les composants de contexte. Ces composants sont déployés sur des périphériques spécifiques comme les capteurs par exemple, pour capturer certaines informations du contexte utiles à l'identification de situation nécessitant une adaptation. Les composants les plus typiques sont les composants de capture d'informations environnementales telles que la température, l'humidité, le déplacement, la fumée, le son, etc. Les valeurs maximale et minimale forment un intervalle dans lequel la propriété doit se situer dans le cas contraire, cela déclenchera un évènement d'évaluation de la QdS. Contrairement aux autres propriétés, ces valeurs de seuil sont dépendantes de l'application. Selon le type d'application, les seuils peuvent être différents. Par exemple, pour une application de surveillance, nous allons capturer tous les sons situés au dessus du seuil du bruit de fond de la pièce lorsqu'elle est vide. Le seuil maximum est établi à 15 dB. Pour une application telle que l'application de visite du musée, nous capturons le niveau sonore de façon à détecter une conférence à savoir un niveau sonore situé au dessus de 70 dB.

Deuxièmement, nous avons les composants logiciels de l'application. Nous nous intéressons à leur partie métier d'une part, c'est à dire leurs capacités fonctionnelles qui permettent de produire les données métier. Par exemple, pour un composant qui traite des données vidéo, il faut définir le nombre de couleurs et le nombre d'images par seconde ou encore la résolution de l'image. D'autre part, nous nous intéressons à leurs caractéristiques techniques tout comme pour les périphériques afin de connaitre leur taux de consommation de CPU, la mémoire requise pour qu'ils fonctionnent ou leur débit d'émission.

Une propriété mesurable peut être statique ou dynamique. Une propriété statique est une propriété qui ne change pas au cours de l'exécution. Dans le cas des propriétés statiques, certains attributs peuvent ne pas comporter de valeur comme c'est le cas pour le nombre de couleurs. Un nombre de couleurs possède une valeur, en revanche n'a ni d'unité ni de valeurs seuils. Les propriétés statiques définissent les capacités théoriques de l'infrastructure alors que les propriétés dynamiques reflètent l'état à un instant donné des ressources disponibles.

Une propriété abstraite est identifié par un nom. Elle correspond à des informations abstraites, que l'on ne peut pas identifier par une valeur. Ce sont des données principalement produites par les interactions avec l'utilisateur et avec l'environnement qu'il faut pouvoir interpréter. C'est la représentation d'une situation ou d'un souhait non matériablisable mais dont l'apparition ou la disparition demande une adaptation du fonctionnement de l'application. Elle peut être composée d'un ensemble de propriétés comme c'est le cas pour la conférence par exemple.

La figure 3.2 représente les différents éléments composant le contexte avec leurs attributs. Dans la partie haute de la figure nous retrouvons les éléments fournissant des informations mesurables, les propriétés. Ce sont les périphériques, les composants logiciels de capture du contexte et les composants métiers de l'application. D'après les différents contextes que nous avons définis dans le chapitre 2, les propriétés mesurables relèvent de deux types : Les propriétés des périphériques ainsi que les propriétés non-fonctionnelles des composants logiciels font partie du *contexte d'exécution*. En revanche, les propriétés fonctionnelles des composants de l'application et les propriétés mesurées par les composants de capture du contexte font partie du *contexte d'utilisation*. Dans la partie basse sont représentés les éléments fournissant des informations abstraites, les propriétés abstraites. Ce sont les composants d'interface utilisateur et la vue de l'application par la plate-forme. Les informations fournies par l'interface utilisateur concernent le *contexte utilisateur* alors que la plate-forme renferme les spécifications liées à l'utilisation de l'application et donc le *contexte d'utilisation*. Bien qu'il existe une frontière sémantique entre les deux types de représentation des composants de contexte, propriétés mesurables et propriétés abstraites sont liées. Nous pouvons voir que les propriétés abstraites peuvent être détectées de deux façons.

Une propriété abstraite peut résulter d'une propriété mesurable ou d'une combinaison de propriétés mesurables. Par exemple, la propriété abstraite d'incendie peut être la combinaison de la propriété température et de la propriété fumée. La combinaison d'une valeur au dessus du seuil défini pour la propriété température et d'une valeur au dessus du seuil défini pour la propriété fumée va produire la propriété abstraite incendie qui va produire l'évènement contextuel alarme.

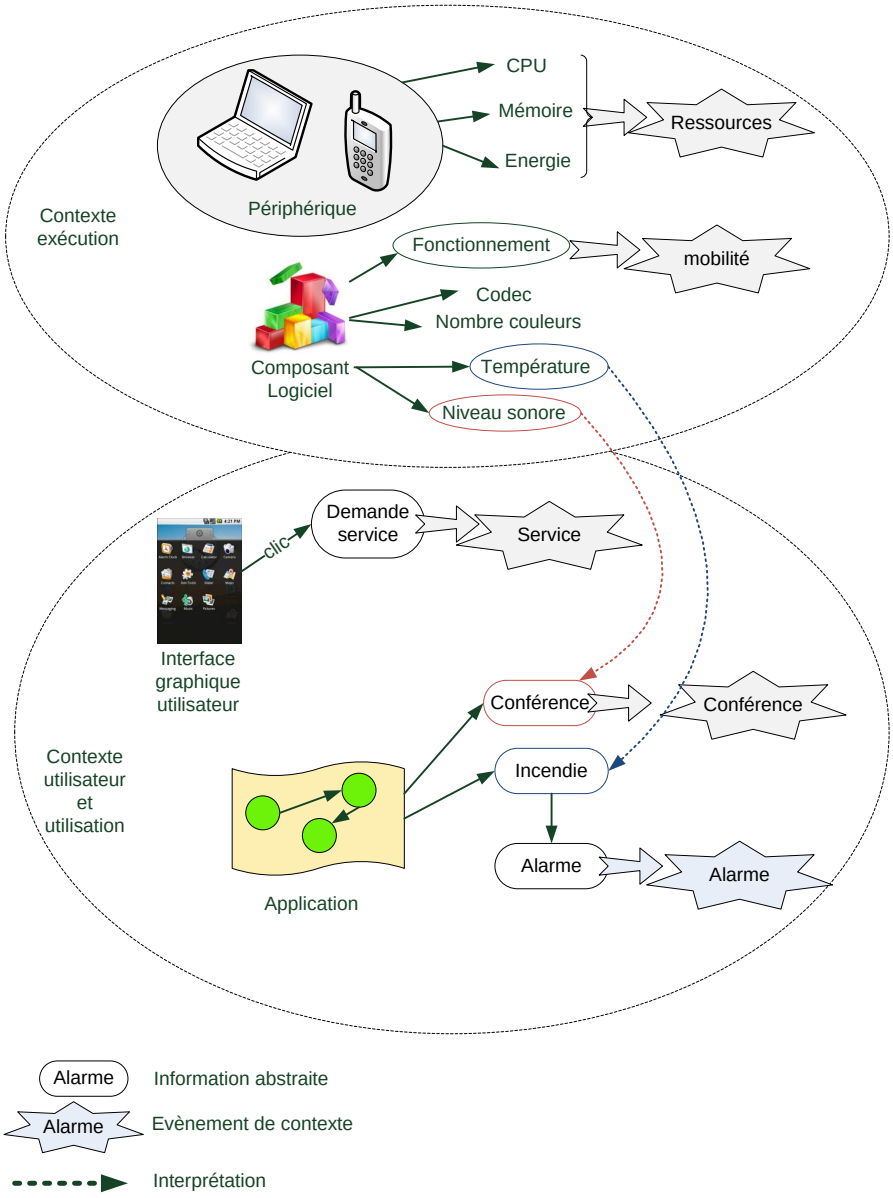

Figure 3.2 – Composition du contexte et interactions

Une propriété abstraite peut résulter d'une autre propriété abstraite. Par exemple nous pouvons enrichir le contexte d'une propriété abstraite d'alarme. Et effet quel que soit l'incident qui peut survenir dans le musée, la conséquence est que tous les visiteurs sont dirigés vers la sortie. Dans notre exemple, la propriété abstraite incendie peut être associé à la propriété abstraite d'alarme qui déclenche un évènement alarme. Une propriété abstraite seule peut également engendrer un évènement. Lorsque l'utilisateur clique sur l'interface de l'application pour demander un service, ce clic correspond à la propriété abstraite de demande d'un service et déclenche l'envoi d'un évènement demande de service

à la plate-forme.

Les liens entre les propriétés abstraites et les propriétés mesurables sont des interprétations ou des abstractions telles que le définit [22][12][63]. Il existe plusieurs façons de définir une abstraction d'une propriété mesurable ou d'un ensemble de propriétés mesurables. La première est l'utilisation de règles actives telles que les règles ECA inspirées par le domaine des bases de données actives [24] [21][70]. Ces règles obéissent à un mécanisme en trois étapes : Evènement, Condition, Action.

Evènement : spécifie le type d'évènement qui active la règle

Condition : agit comme une garde

Action : définit la reconfiguration à appliquer

[20] propose un outil de développement d'applications auto-adaptables basées composants dont les politiques d'adaptation se basent sur les règles ECA. Le listing 3.1 montre un exemple de définition d'une règle ECA pour la reconfiguration d'un composant.

```
when memory = low
if component.hasInterface("passivation")
do component.passivate
```

Listing 3.1 – Exemple d'une règle ECA pour la reconfiguration d'un composant

[20] propose notamment de définir des évènements composites en utilisant un ensemble d'opérateurs pour détecter les séquences, les alternatives ou les conjonctions d'évènements. Ainsi, ce mécanisme permet de réagir à un état de l'application ou son contexte mais également à son évolution.

La deuxième façon de réaliser l'abstraction de propriétés mesurables est la programmation par aspects [39] [11]. Un aspect est défini selon trois concepts : le point de coupe, le greffon et le tissage. Le *greffon* permet de définir à quel moment l'adaptation doit être appliquée et quelle adaptation doit être appliquée. Il correspond à l'évènement et à l'action des règles ECA. Le *point de coupe* définit l'endroit dans l'application où l'adaptation doit être appliquée. Enfin, le *tissage* définit comment est appliquée l'adaptation. La programmation par aspects encapsule à la fois l'abstraction du contexte, l'action à effectuer ainsi que l'endroit où l'effectuer.

Ces approches sont très complètes cependant, associer les ensembles d'évènements et les actions dans une même entité pose la question de la synchronisation des évènements et de leur pertinence relativement à la fréquence des mesures et au temps de transfert réseau notamment lorsque l'application utilise des systèmes contraints tels que les capteurs. Pour éviter ces problèmes, [16] et [63] proposent de réaliser des politiques de gestion du contexte par composition. Les évènements, et donc les composants qui les déclenchent, sont reliés entre eux selon une hiérarchie. En haut de la hiérarchie se trouve l'action de reconfiguration. Cette approche permet non seulement de séparer la détection du changement de contexte de l'action mais également de distribuer la gestion du contexte. Cette distribution de la gestion du contexte est une approche qui correspond au domaine de l'informatique ambiante et des applications mobiles compte tenu de la forte distribution des ressources et de l'étendue du contexte.

Notre politique de gestion du contexte repose sur les principes des règles ECA et de la distribution telle que le propose [16]. Nous définissons des règles de correspondance entre

un évènement et un autre évènement et entre un évènement et une action de reconfiguration. Par exemple : {niveau sonore >= 70 → DebutConference}

{Conference → −0.3, son}

Lorsque le niveau sonore atteint 70dB, l'évènement *Début Conférence* est déclenché. Lorsque la plate-forme reçoit l'évènement *Début Conférence*, elle baisse la qualité des services qui proposent du son (voir section 3.3.1.1).

Pour résumer, notre modèle de contexte est composé de deux entités correspondant à deux types de données :

- les propriétés mesurables qui sont produites par les contextes d'exécution et d'utilisation.
- les propriétés abstraites qui sont produites par les contextes utilisateur et utilisation.

Ces informations proviennent de différentes sources regroupées dans le tableau 3 :

Origine	Type d'information de contexte	Type de contexte
Composant spécifique à la capture du contexte	Mesurable (environnement)	Utilisation
Composant spécifique à la capture du contexte (Interface graphique)	Abstraite (actions)	Utilisateur
Composant métier	Mesurable (débit, couleurs, mémoire requise, etc.)	Exécution
Composant métier	Abstraite (son, image, etc.)	Utilisation
plate-forme	Mesurable (ressources, réseau)	Exécution
plate-forme	Abstraite (état de l'application)	Utilisation

Table 3 – Origine des informations contextuelles

Ces données répondent aux deux définitions suivantes :

- un évènement contextuel est déclenché par une ou un ensemble de propriétés mesurables.
- un évènement contextuel est déclenché par une propriété abstraite.

Ainsi décrit, le contexte va servir de base pour l'évaluation de la QdS dont nous allons décrire le modèle dans la section suivante.

3.3 Modèle de Qualité de Service

Le paragraphe précédent présentait le modèle de contexte que nous avons défini afin de modéliser toutes les informations utiles à l'adaptation d'une application.

Ce modèle est une première étape dans la conception d'applications adaptables et correspond à la première couche de l'architecture proposée par [74]. Cette architecture propose de gérer la sensibilité au contexte en trois couches.

La première est l'acquisition des informations contextuelles. Pour cela nous avons défini un modèle permettant de décrire ces informations et de fournir à la couche supérieure les données utiles à l'adaptation. Nous avons également identifié toutes les entités produisant ou fournissant les informations de contexte. Les composants spécifiques de l'application produisent les informations de *contexte utilisateur* et utilisation. Les conteneurs des composants métiers et des connecteurs permettent de recueillir des informations du *contexte d'exécution* telles que le débit au niveau de chaque liaison par exemple. La plate-forme recueille les informations du *contexte d'exécution* au moyen des cartes d'identité des périphériques et des composants métiers ainsi que par les composants de contexte situés sur chaque périphérique (voir section 3.5). Toutes ces informations sont ensuite dirigées vers le service *Superviseur* de la plate-forme puis vers le service *Générateur de Reconfiguration* (voir section 4.5).

La deuxième couche correspond à l'interprétation de ces données. À ce niveau, les données issues de la première couche sont encore brutes dans le sens où elles sont seulement décrites. Une information seule est inutile. Par exemple, une température mesurée à un instant t d'une valeur de 30° n'a pas de sens telle quelle. Il faut l'évaluer en la comparant à une valeur de référence telle qu'un seuil. Si la valeur de la température est au dessus du seuil, alors il faut reconfigurer. Suite à l'évaluation, une décision d'adaptation est transmise à la troisième couche.

La troisième couche fournit les mécanismes permettant d'adapter l'application au contexte. Il existe plusieurs moyens d'adapter une application telle que la reconfiguration de son architecture, la configuration de ses composants ou encore le redéploiement des composants.

Dans ce paragraphe, nous nous intéressons à l'interprétation des informations contextuelles. Dans le chapitre 2, nous avons retenu la définition de [19] qui dit que les évolutions du contexte peuvent se traduire par des variations de la qualité du service rendu par l'application. Rappelons que dans nos travaux, nous avons choisi de traiter particulièrement deux types de qualité de service en sus de la l'assurance de la continuité de service : la *QdS Utilité* et la *QdS Pérennité*.

Dans la suite de ce paragraphe, nous allons décrire pour chaque QdS les caractéristiques que nous prenons en compte dans la mesure de la QdS globale d'une application reconfigurable pour périphériques mobiles et contraints.

3.3.1 Caractéristiques de Qualité de Service

Dans cette section nous nous intéressons à l'évaluation des informations contextuelles et à la mesure de la QdS de l'application. La QdS est influencée par le contexte de l'application.

La figure 3.3 rappelle les interactions entre les trois contextes et les deux types de QdS que nous avons défini dans le chapitre 2.

La *QdS Utilité* est influencée majoritairement par l'utilisateur qui émet des choix de fonctionnalités et par le contexte utilisation qui renferme toutes les conditions d'utilisation de l'application, exprimées en règles. Le *contexte d'exécution* influence également la *QdS Utilité* par le biais des composants de l'application mesurant des grandeurs physiques

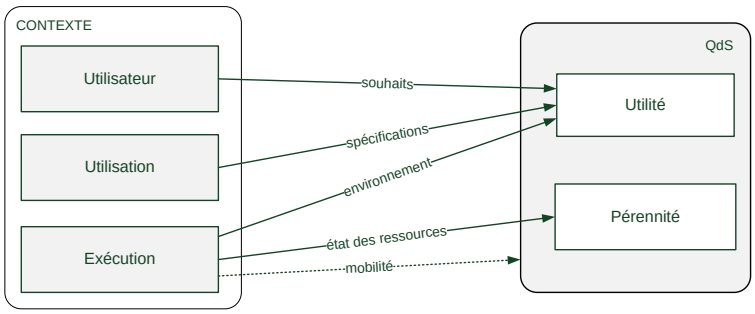

Figure 3.3 – Schéma d'interactions entre les contextes et les QdS

de l'environnement telles que la luminosité ou la température. Combinées aux conditions d'utilisation, ces informations modifient la QdS.

La *QdS Pérennité* quant à elle est essentiellement influencée par le *contexte d'exécution* et plus particulièrment l'état des ressources des périphériques. Chaque changement au niveau des ressources entraîne une modification de la durée de vie restante de l'application.

3.3.1.1 QdS Utilité

La *QdS Utilité* mesure les caractéristiques dépendantes des contextes utilisateur et utilisation. Elle permet de mesurer la conformité d'une configuration par rapport aux conditions courantes d'utilisation. Elle mesure également la correspondance des fonctionnalités par rapport aux choix de l'utilisateur.

D'autres travaux relatifs à l'adaptation d'applications au contexte prennent en compte l'utilisateur. [13] propose d'adapter l'application selon plusieurs dimensions de contexte telles que le réseau, l'utilisateur, le périphérique, la localisation et l'environnement. Pour chaque dimension est défini un profil. Ce profil définit des paramètres permettant de caractériser des situations qui demandent une adaptation du comportement de l'application. Parmi ces dimensions, nous nous intéressons à l'utilisateur. Dans [13], les paramètres définis dans le profil utilisateur peuvent être ses préférences comme la langue de l'application, le type de média (image, audio, texte), etc. Un tel niveau de détail permet de proposer à l'utilisateur une application qui lui plait le plus possible.

Cependant, dans nos travaux, nous nous confrontons à des conditions d'utilisation qui imposent des modifications de l'application et qui sont prioritaires devant les préférences de l'utilisateur. De plus, aujourd'hui, lorsqu'une personne achète un périphérique mobile, elle le choisit pour ses caractériques : grand écran, couleur, tactile, pouvant envoyer, recevoir et lire des vidéos, des images, etc. Nous supposons que quelle que soit l'application, cette personne souhaite utiliser les caractéristiques de son périphérique au maximum de leurs capacités. En effet, il parait peu probable qu'un utilisateur possédant un appareil avec un grand écran couleur préfère recevoir des images en noir et blanc. C'est pourquoi dans nos travaux, contrairement à [13], nous ne demandons pas à l'utilisateur ses préférences. Par défaut, nous choisissons de lui proposer quoi qu'il arrive, la meilleure qualité possible de l'application par rapport aux capacités du périphérique. Dans ces travaux, le *contexte*

utilisateur se réduit uniquement au choix d'une fonctionnalité proposée par l'application par l'intermédiaire d'une interface homme-machine. L'utilisateur peut seulement émettre des choix sur la demande ou l'arrêt d'une fonctionnalité. Néanmoins de tels choix imposent l'ajout, le démarrage, l'arrêt ou le retrait d'un service, c'est-à-dire qu'ils modifient la structure de l'application et donc impliquent des reconfigurations.

De la même façon, en fonction d'évènements caractérisant une condition d'utilisation, il faudra vérifier que les spécifications de l'application sont bien respectées. Par exemple, lorsque l'évènement « *DébutConférence* »apparaît, si un utilisateur se trouve dans la salle du musée où se déroule la conférence, nous proposons d'éviter la transmission de son. A partir de ce constat, si l'application actuellement en éxécution utilise des composants dont le rôle est de traiter des données audio, il est préférable de les retirer et pour cela il faut reconfigurer l'application.

La note de *QdS Utilité* traduit la proximité de la qualité de service de la configuration courante de l'application par rapport à la qualité de service idéale. Elle traduit le degré de conformité de la configuration avec :

- la qualité intrinsèque de l'application
- les spécifications de l'application.

La qualité intrinsèque est la qualité liée au type de l'application, son domaine d'application et le but fonctionnel qu'elle doit atteindre. Cette qualité est définie par le concepteur de l'application. Parmi toutes les configurations possibles d'une application, il décide en fonction du but final, quelle est la configuration offrant la meilleure qualité, puis celle qui offre une bonne qualité mais moins que la première, et ce jusqu'à la dernière configuration qui offre la qualité la moins bonne.

Nous mesurons la qualité de service en pourcentage. La qualité de service Utilisation idéale d'une configuration prend la note de 1, c'est-à-dire qu'elle correspond à 100% aux conditions d'utilisation et propose bien la fonctionnalité sélectionnée par l'utilisateur. La note vaut 0 quand le service rendu ne correspond ni aux spécifications, ni aux souhaits de l'utilisateur.

Dans la méthode de conception, la première étape consiste à identifier tous les services que doit offrir l'application. Elle concerne uniquement les objectifs fonctionnels de l'application. Dans la deuxième étape, le concepteur décrit tous les cas d'utilisation de l'application. C'est à ce stade qu'il décrit les situations d'utilisation et met à jour les conditions qui impliquent des adaptations. A partir de cette étape et pour les quatre suivantes, nous prenons en compte la qualité de service en définissant dans le détail les critères et les outils pour la mesurer. La troisième étape consiste à regrouper des configurations selon leur rôle. Ces groupes permettent de classer les différentes configurations d'un service selon des critères discriminants pour le type d'application visé. Dans l'exemple de la visite du musée, nous avons choisi comme critères le son, l'image et le texte. Ainsi lorsque l'évènement « *DébutConférence* »est levé, le choix d'une configuration sera orienté en priorité vers des configurations de rôle image et texte. Cependant nous ne souhaitons pas exclure un groupe. Dans le cas où aucune configuration sans son ne peut être déployée, cela signifierait que le service deviendrait indisponible. Or l'objectif est d'assurer le service quel que soit le contexte, même s'il est de mauvaise qualité. C'est pourquoi nous préférons dans ce cas mal noter une configuration de rôle son qui ainsi ne sera choisie qu'en dernier

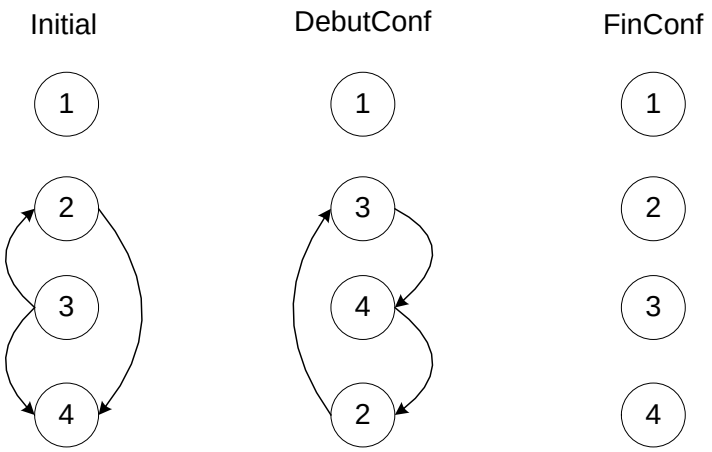

Figure 3.4 – Variation du classement des configurations selon leur note de *QdS Utilité*

recours pour assurer le service. La quatrième étape de la méthode consiste à proposer un premier classement de toutes les configurations confondues selon un critère intrinsèque. Ce classement est établi par le concepteur lui-même en fonction du type de l'application. Il décide, selon lui, quelle est la configuration offrant la meilleure qualité de service pour cette application jusqu'à la configuration de pire qualité. A ce stade les configurations ont donc chacune un rôle et une note.

Puisque nous ne voulons pas exclure des configurations sous peine de ne plus assurer le service, une solution est de modifier les rangs du classement initial en fonction des évènements de contexte. Pour cela il faut pouvoir donner des poids aux différentes configurations en fonction de leur rôle. La cinquième étape est dédiée à la définition des évènements. Chaque évènement est associé à un rôle dont le poids sera plus ou moins fort et fera monter ou descendre les configurations dans le classement.

Si nous résumons, un évènement doit être associé à un rôle et à un poids. Les évènements contextuels défini à l'étape 5 (section 3.5.5 doivent alors être typés.

$$< Evenement\ E,\ Role\ R,\ Operateur\ Op,\ Poids\ P >$$

Pour l'évènenement E, la plate-forme doit appliquer le poids P avec l'opérateur Op sur les notes de *QdS Utilité* des configurations définies par le rôle R.

$$< \text{DébutConf},\ Son,\ -,\ 20 >$$

$$< FinConf,\ Son,\ +,\ 20 >$$

Pour l'évènement « *DébutConf* », la plate-forme baissera de 0,2 la note de QdS des configurations de rôle Son.

Une fois que le classement est établi, il faut évaluer chaque configuration et voir si l'une d'entre elles peut être déployée sur les périphériques disponibles. Pour celà il faut évaluer leur QdS au niveau Pérennité.

3.3.1.2 QdS Pérennité

La *QdS Pérennité* regroupe toutes les caractéristiques concernant le *contexte d'exécution*. Elle traduit l'influence de l'évolution des ressources matérielles, logicielles et réseau sur la durée de vie de l'application. Alors que pour la *QdS Utilité* nous raisonnions directement au niveau des services, ici nous devons d'abord raisonner au niveau des composants pour obtenir la note de QdS de chaque composant de la configuration et calculer ensuite la note du service entier. Lorsque la plate-forme va chercher une nouvelle configuration à déployer, elle va dans un premier temps chercher, pour chaque composant, un périphérique capable de le supporter. Pour cela, elle doit vérifier que le périphérique dispose des ressources requises par le composant. Puis, une fois que les composants sont placés, elle vérifie que le réseau dispose des débits requis par chaque liaison entre périphériques distants. En effet la vérification des ressources réseau ne peut se faire que lorsque le déploiement est réalisé. C'est pourquoi l'évaluation de la *QdS Pérennité* est faite par deux notes : une note de *QdS Pérennité de consommation des ressources* et une note de *QdS Pérennité de consommation du réseau*.

Note de *QdS Pérennité de consommation des ressources* Les composants peuvent avoir plusieurs notes de QdS selon le périphérique sur lequel ils sont déployés. Cette note est en réalité composée de trois notes en pourcentage : une note de consommation de CPU, une note de consommation mémoire et une note de consommation d'énergie. Chacune des trois notes doit être établie selon le même protocole. Tout d'abord chaque composant doit être exécuté sur la même configuration, pendant la même durée et à la même fréquence. Par rapport à cette configuration, on obtient une série de mesures pour l'occupation CPU, l'occupation mémoire et la dépense d'énergie. Ensuite, pour chaque ressource est établie une moyenne qui est ramenée à un pourcentage. Un composant a donc pour note un triplet composé :

- d'une note de consommation CPU, C
- d'une note de consommation de mémoire, M
- d'un note de consommation d'énergie, E

$$Composant(C, M, E)$$

Chaque composant est évalué par rapport à un périphérique. Un composant ne peut être exécuté sur un périphérique que si les ressources du périphérique le permettent c'est à dire que le périphérique dispose d'un taux d'occupation restant de CPU suffisant, d'un taux de mémoire restante suffisant et enfin d'un taux d'énergie restante suffisant. Nous devons donc comparer la consommation du composant par rapport aux ressources disponibles sur le périphérique. Pour cela nous mesurons sur le périphérique les mêmes valeurs que celles utilisées pour obtenir la note du composant. On obtient alors une note pour le périphérique qui est un triplet composé :

- d'une note de CPU disponible
- d'une note de mémoire disponible
- d'une note d'énergie disponible

$$Peripherique(C, M, E)$$

Lorsque le composant C(C_c, M_c, E_c) est déployé sur le périphérique H(C_h, M_h, E_h), sa note devient alors :

$$C\ sur\ H = \min(\max(0, C_h - C_c), \max(0, M_h - M_c), \max(0, E_h - E_c)) \quad (3)$$

Cependant, comment départager un composant qui consomme peu d'énergie mais occupe la quasi totalité du CPU et d'un composant qui consomme autant d'énergie que d'occupation CPU ?

La *QdS Pérennité* traduit en pourcentage, la consommation des ressources d'un périphérique par un composant. Comparons trois composants A, B et C :

$$A(0.5, 0.2, 0.2);\ B(0.2, 0.5, 0.2);\ C(0.2, 0.1, 0.6)$$

A et B consomment peu d'énergie par rapport à C mais beaucoup plus de CPU et de mémoire. Si on fait la moyenne des trois notes, ils obtiennent tous les notes de 0.3 et sont donc a priori tous les trois équivalents.

Déployons chacun d'entre eux individuellement sur un périphérique P(0.8, 0.6, 0.7) selon la formule 3 :

$$A_p(0.3, 0.4, 0.5);\ B_p(0.6, 0.1, 0.5);\ C_p(0.6, 0.5, 0.1).$$

Dans cet exemple, lorsque C est déployé sur P, il ne reste plus beaucoup d'énergie à P pour continuer de fonctionner. C est très énergivore, bien qu'il consomme peu de CPU et de mémoire, il ne parait pas raisonnable de le déployer sur P. A et B sont moins énergivores cependant B consomme quasiment toute la mémoire de P. La mémoire est saturée et plus aucun autre composant ne pourra être déployé sur P. Il en est de même lorsque le CPU est saturé, aucun composant supplémentaire ne peut être exécuté.

Le discriminant dans cet exemple est la ressource pour laquelle la note est la plus basse. Nous proposons alors que la note d'un composant C sur un périphérique H soit :

$$note(\text{C sur H}) = \max(0, \min(C_h - C_c, M_h - M_c, E_h - E_c)) \quad (4)$$

Lorsqu'on souhaite déployer plusieurs composants sur un même périphérique, il faut alors soustraire aux ressources du périphérique les ressources consommées par chacun des composants. a(C_a, M_a, E_a) et b(C_b, M_b, E_b) sont déployés sur H(C_h, M_h, E_h) :

$$\text{a et b sur H} = \max(0, (C_h - C_a - C_b,\ M_h - M_a - M_b,\ E_h - E_a - E_b))$$

Selon la formule 4, la note de l'assemblage (a,b) sur le périphérique H est alors :

$$\text{note(a, b sur H)} = \max(0, \min(C_h - C_a - C_b, M_h - M_a - M_b, E_h - E_a - E_b))$$

La note d'un déploiement D pour lequel les composants a et b sont placés sur H tandis que le composant c est placé sur P, est le min des deux notes obtenues.

$$\text{note(D)} = \text{note(a, b, c sur H, P)} = \min(\text{note}(a, bsurH),\ \text{note}(CsurP))$$

Cette façon d'évaluer la *QdS Pérennité* n'est surement pas la plus optimale, ni la plus efficace puisque à un assemblage qui consomme peu sur un périphérique et beaucoup sur l'autre, on préfère choisir un assemblage qui consomme moyennement sur deux périphériques. Plutôt que d'en épuiser un rapidement, on en épuise deux régulièrement. Avec cette méthode, on épuise peu à peu tous les périphériques plutot que de les épuiser un par un. Le but étant de maximiser la durée de vie de l'application en maximisant la durée de vie des périphériques, la méthode proposée ici est alors logique.

La note vaut 0 quand le service consomme toutes les ressources du périphérique et met en péril la durée de vie du périphérique et par conséquent la durée de vie de l'application toute entière. La note est égale à 1 lorsque le service consomme peu de ressources par rapport à celles disponibles sur le périphérique et contribue à la maximisation de la durée de vie de l'application. La note 0 est atteinte lorsque par exemple $(C_h - C_c) < 0$. Ceci signifie que le périphérique H ne peut pas supporter le composant C.

Ce premier processus d'évaluation de la *QdS Pérennité* permet d'évaluer la disponibilité des ressources des périphériques face aux exigences des composants en termes de consommation des ressources matérielles.

Ce type d'évaluation est proposé par la plupart des plate-formes d'adaptation d'applications mobiles notamment, MUSIC [64] et AxSeL [30]. Cependant peu de ces approches prennent en compte le coût des liaisons réseau entre les périphériques. Dans le contexte des périphériques contraints et d'une forte distribution des applications sur lesquels nous centrons nos travaux, nous considérons que la prise en compte du poids des liaisons réseau, qui se traduisent par une forte consommation d'énergie, est indispensable. De la même façon que nous évaluons le poids de la consommation des ressources matérielles, nous évaluons le poids de la distribution des composants sur les périphériques.

Note de *QdS Pérennité* de consommation du réseau Pour un déploiement choisi, nous connaissons le périphérique d'accueil de chacun des composants. Dans la configuration, nous avons les liens entre composants, donc nous connaissons les liaisons réseau à établir entre les composants et plus généralement entre les périphériques.

Un périphérique H_1 connait par sa carte d'identité le débit maximum théorique du réseau R_1 qui le lie à H_2 ($DMaxT_{H_1R_1}$). De même H_2 possède l'information $DMaxT_{H_2R_1}$.

Ces deux valeurs ne sont pas forcément les mêmes (par exemple en Wifi 802.11b et 802.11g). Nous conserverons :

$$DMaxT_{H_1H_2} = \min(DMaxT_{H_1R_1}; DMaxT_{H_2R_1})$$

Le périphérique H_1 connait également le débit moyen actuellement utilisé sur sa liaison réseau avec H_2 ($DMoy_{H_1H_2}$). De façon similaire, H_2 a la même information $DMoy_{H_2H_1}$. Cette information est fournie par les conteneurs de connecteurs présentés au chapitre 5 section 4.4.2. Ces deux valeurs $Dmoy_{H_1H_2}$ et $DMoy_{H_2H_1}$ sont identiques. Nous garderons comme débit disponible pour la liaison $H_1 - H_2$ sur R_1 :

$$BP_{H_1H_2} = DMaxT_{H_1H_2} - DMoy_{H_1H_2}$$

Pour chaque lien entre deux composants $C_1 - C_2$ (nous ne prenons en compte que les liaisons entre composants sur périphériques distincts), nous connaissons le débit de sortie

de C1 d'après sa carte d'identité. Comme nous savons sur quels périphériques sont C_1 et C_2, nous savons sur quelle liaison $H_i - H_j$ nous avons besoin de ce débit de sortie.

Il faut prendre tous les liens entre composants, calculer pour chaque liaison,

$$\frac{\max((BP_{H_iH_j} - (\text{Somme des débit de sortie des composants émettant sur cette liaison})), 0)}{DMaxT_{H_iH_j}}$$

Remarque. Le max est réalisé avec 0 au cas où la somme des débits demandés par le déploiement serait supérieure à la bande passante.

Maintenant que nous avons décrit les deux QdS et leur évaluation individuelle, nous allons décrire le déroulement de l'évaluation d'une application.

3.3.2 Représentation et évaluation de la QdS

Le modèle de qualité de service que nous avons défini permet une mesure à deux niveaux : matériel et réseau. Un tel modèle permet de fournir des applications offrant le meilleur compromis entre la qualité d'utilisation et la durée de vie, c'est à dire la meilleure façon d'utiliser une application tout en respectant les contraintes de durée de vie imposées par le contexte [64].

En fonction du contexte nous pouvons choisir de donner plus ou moins de poids à chacune des QdS. Par exemple, lorsque le niveau de batterie devient faible, la priorité est donnée à la pérennité de l'application. Dans ce cas là, il est possible que soit choisie une configuration qui ne respecte pas toutes les spécifications de l'application mais qui en revanche permette de la faire durer plus longtemps. Dans le cas d'une application de surveillance, lorsqu'un évènement de détection d'intrus survient, il est important que l'application fournisse une image de haute qualité. La *QdS Utilité* sera alors privilégiée par rapport à la *Pérennité*.

Dans le premier cas, un poids fort est donné à la *QdS Pérennité*. Ceci signifie que lors de l'évaluation des configurations, la plate-forme pourra descendre relativement bas dans le classement mais, par contre, les notes de pérennité devront rester proche de la note idéale. Dans le deuxième cas, c'est au contraire, un poids fort qui est donné à la *QdS Utilité*. Les configurations devront être choisies parmi le haut du classement même si des configurations de moins bonne qualité au niveau *Utilité* sont plus pérennes.

Les poids sont donc des seuils que l'évaluateur ne doit pas franchir dans un contexte précis. Etant donné que les configurations doivent fournir la meilleure qualité de service possible, ces seuils permettent de choisir des configurations dont les notes sont situées dans l'espace défini par la qualité idéale, et les seuil fixés pour les deux types de QdS (**figure 3.5**).

Les QdS étant mesurées par des notes normalisées entre 0 et 1, elles peuvent être représentées sur 2 axes.

L'évaluation se déroule de la façon suivante. Une configuration A à un instant t, est représentée par la note de *QdS Utilité*, $Ut(A)$ (figure **??**). Cette note définit un axe horizontal sur lequel sont placées les notes de consommation de ressources des différents

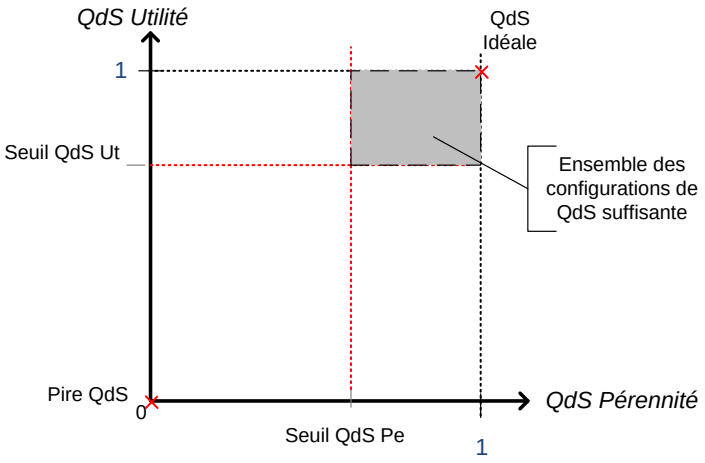

FIGURE 3.5 – Représentation de l'ensemble des configurations de QdS suffisante

déploiements possibles de la configuration A, $Pe_D(A)$. Si aucune des notes $Pe_D(A)$ n'est située dans l'espace des déploiements possibles, une autre configuration est évaluée. Dès qu'un déploiement a une note $Pe_D(A)$ qui appartient à l'espace des déploiements possibles, alors sa note de consommation est calculée. On obtient alors, pour chaque déploiement, deux notes :

- une note de consommation des ressources $Pe_D(A)$
- une note de consommation réseau $Pe_R(A)$

La note globale de *QdS Pérennité* est alors obtenue par la formule 5 :

$$\text{note QdS Pe de A} = \min\big(Pe_D(A), Pe_R(A)\big) \tag{5}$$

Ce choix d'évaluation permet d'équilibrer la consommation des ressources matérielles et des ressources réseau qui sont fortement corrélées dans le cas des périphériques contraints.

Selon le type d'évènement de contexte, l'action à mener n'est pas la même et l'évaluation précédant cette action est également différente.

Les évènements provenant du *contexte utilisateur* sont relatifs à des demandes au niveau des services proposés par l'application. L'utilisateur a la possibilité de demander le démarrage, l'arrêt, l'ajout ou le retrait d'une option sur un service en cours. Concrètement cela se traduit par des ajouts, suppressions, démarrages ou arrêts de composants, c'est à dire une modification de la structure de l'application. Ce type d'évènement ne modifie ni les notes de QdS et donc ne modifie pas le classement des configurations, ni les seuils de priorité des *QdS Utilité* et *Pérennité*. Il demande simplement une évaluation de l'application avec la modification souhaitée et le choix d'une nouvelle configuration.

Les évènements provenant du *contexte d'utilisation* sont relatifs à des changements de contexte influant les spécifications de l'application. Comme nous l'avons décrit dans la section 3.3.1.1, certains évènements permettent de modifier les notes de *QdS Utilité* et de modifier l'ordre des configurations dans le classement.

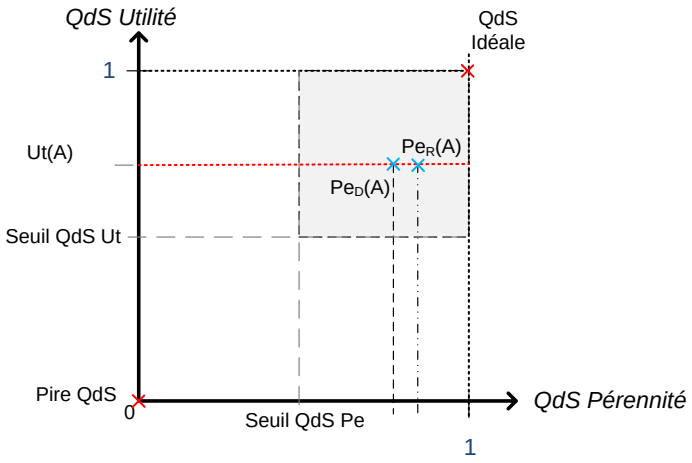

Figure 3.6 – Représentation de la QdS d'une configuration à un instant $t0$

3.4 Modèle des applications

Comme nous l'avons défini dans les objectifs au chapitre 2, nos travaux s'intéressent à la gestion de la qualité de service des applications c'est à dire la gestion de leurs propriétés non fonctionnelles. Une telle problématique nécessite une nette séparation des préoccupations entre la logique métier et la logique non fonctionelle. C'est pourquoi nous avons choisi d'utiliser une plate-forme de supervision en charge de la partie non fonctionnelle afin de piloter les applications. Elle a pour objectif de gérer la structure de l'application par des reconfigurations dynamiques afin que celle-ci fournisse un service de la meilleure qualité possible. Elle se base sur un mécanisme d'évaluation de la QdS à deux critères : Utilisation et Pérennité. Le premier permet d'évaluer la conformité par rapport aux spécifications d'utilisation de l'application alors que le deuxième évalue le degré de longévité de l'application. Le principal intérêt de cette plate-forme est qu'elle permet d'avoir une vue complète de la structure de l'application. En effet elle est répartie sur l'ensemble des périphériques participant à l'application et reçoit des états sur le fonctionnement et la structure de l'application de la part de ses composants ainsi que des informations sur l'état des périphériques participant à l'exécution de l'application. Elle envoie ensuite des commandes à l'application en réponse aux évènements contextuels afin de l'adapter et d'améliorer sa qualité de service. Ainsi, avec un tel reflet de l'application, nous pouvons proposer des solutions qui ne sont pas uniquement basées sur des critères techniques de faisabilité.

Nous allons maintenant décrire le modèle des applications pour lesquelles cette plate-forme est dédiée.

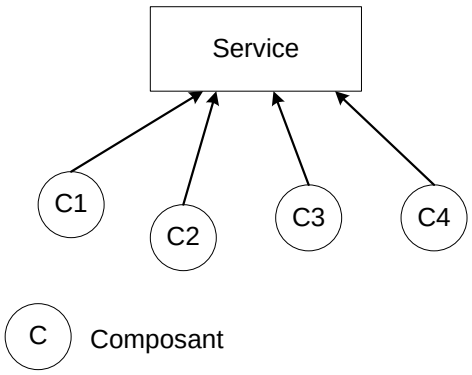

FIGURE 3.7 – Illustration de la définition d'un service selon la définition 3.1

3.4.1 Service

Ces dernières années nous avons vu les paradigmes de programmation évoluer et, avec eux, une nouvelle abstraction est apparue : le service. Un paradigme tel que le Service Oriented Architecture (SOA) préconise l'usage du service comme entité de construction d'une application. Largement utilisé par les services web, la programmation orientée service se base sur les principes de couplage faible, de publication, de découverte et de liaisons entre fournisseurs et consommateurs de services. Bien que de nombreux travaux s'accordent sur la pertinence d'une telle approche, il n'existe aucun consensus sur la définition d'un service. Nous proposons une première définition qui s'inspire des celles que les plus citées.

Définition 3.1 (Service) *Un service est une entité bien définie, autonome et indépendante de la plate-forme. Il désigne un comportement fourni par un composant à n'importe quel autre au travers d'un contrat. Le service est un mécanisme d'accès à des fonctionnalités. Il est fourni par n'importe quel composant ou fournisseur de services. L'accès à un service est réalisé par un couplage faible, à travers des interfaces et une politique de composition. [28][48][57]*

Il s'agit de la définition classique du service dans les architectures orientées services. Ces architectures sont basées sur un ensemble de services qui collaborent et proposent de considérer un service selon les caractéristiques suivantes :

- une large granularité : un service fournit une fonctionnalité qui est le résultat de la collaboration de plusieurs autres fonctions (figure 3.7).
- un couplage faible : les services communiquent entre eux via des opérations standard. Il n'y a pas d'appel direct à un service. Contrairement aux composants qui ont besoin de connaitre leurs liaisons avant l'exécution, les services n'ont pas besoin de se connaitre a priori. Pour collaborer, ils passent par des mécanismes de description, de publication et de découverte.

La deuxième définition donnée par [30][31] propose un nouveau concept alliant les paradigmes service et composant :

Définition 3.2 (ServiceComposant) *Une service composant est un concept hybride qui*

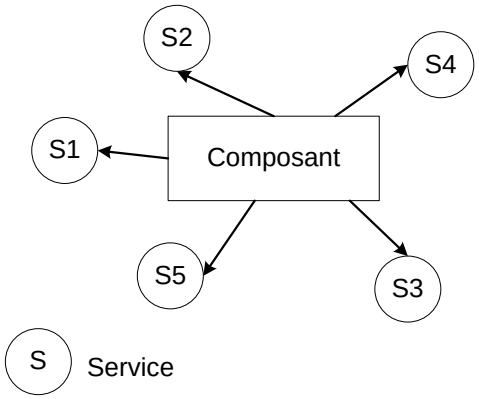

Figure 3.8 – Illustration de la définition d'un service selon la définition 3.2

réunit le composant et le service. Un composant est une entité logicielle encapsulant des fonctionnalités. Il peut être déployé et exécuté. Il possède une description et des interfaces. Une interface correspond à un service. Les services permettent l'import et l'export des fonctionnalités. Un composant est sujet à composition avec d'autres composants à travers des services importés et exportés. Un composant et ses services ne sont pas dédiés à un utilisateur spécifiques.

Cette définition correspond à celle que l'on retrouve dans OSGi [1] qui est une description d'une implémentation possible du paradigme SOA. Contrairement à la première définition qui propose qu'un service encapsule plusieurs fonctionnalités, ici un composant fournit plusieurs fonctionnalités dont chacune correspond à un service. La granularité est de fait plus faible que pour la précédente définition. En revanche, la notion de faible couplage reste identique. Les service sont liés entre eux par des standards et déchargent les composants de toutes les propriétés non fonctionnelles.

Dans nos applications, lorsque nous parlons de services, nous nous approchons de la définition des Web Services (définition 3.1) [57]. Nous définissons un service comme une entité de haut niveau d'abstraction, autonome et fournissant une fonctionnalité qui peut être réalisée par un composant ou un assemblage de composants. Contrairement à [31], un composant fournit une seule fonction et peut collaborer avec d'autres composants afin de participer à la réalisation d'un service. Un même composant peut ainsi intervenir dans la composition de plusieurs services. Si nous faisons un parallèle avec le paradigme objet, un service n'est qu'une interface d'une fonctionnalité d'un composant. Dans nos travaux, un service est l'abstraction d'une somme de composants qui peut, le cas échant, se réduire à un seul composant. De plus cet assemblage de composants peut changer en cours d'éxécution en fonction du contexte dans le but de continuer à fournir le service.

3.4.2 Composant

Nous venons de définir un service comme l'assemblage dynamique de plusieurs composants. Les composants peuvent être de deux natures : matériel ou logiciel. Un composant

matériel est par exemple, une caméra, un microphone ou un thermomètre. Ces composants acquièrent des données brutes compréhensibles uniquement par eux-même. Ces données sont inexploitables dans l'état par l'application. Afin de les traiter, nous avons besoin de les interpréter. Pour cela, est associé à chaque composant matériel, un composant logiciel qui permet de faire la traduction des données brutes et de produire des données exploitables par d'autres composants logiciels.

Ainsi dans ces travaux nous n'allons considérer que les composants logiciels. Nous définissons un composant comme une entité logicielle implémentant une fonction précise, de faible granularité. Un composant peut avoir la propriété d'être délocablisable ou non. Un composant peut être lié à un matériel comme c'est le cas des composants que nous venons de citer. Ainsi un composant qui nécessite des ressources qui ne sont pas disponibles sur un autre composant ne peut pas être délocalisé. Sans le matériel, il ne peut pas fonctionner et devient inutilisable par l'application. Ainsi, certains composants liés à un matériel ne sont pas délocalisables.

Dans le cadre de cette restriction de délocalisation intervient la notion de pertinence des données. Une donnée peut avoir beaucoup d'importance et puiser tout son sens pour l'appication sur un lieu A mais perdre tout intérêt si elle a été produite sur un lieu B. Par exemple, le périphérique supportant un composant logiciel de relevé de température sur le lieu A est saturé ou a atteint un seuil critique du niveau de batterie. Nous choisissons de soulager le périphérique en délocalisant le composant de relevé de température sur un autre périphérique doté d'un thermomètre au lieu B. Cependant, si le lieu B est éloigné du lieu A, nous avons toutes les raisons de penser que la température relevée sur le lieu B ne reflète pas la température relevée jusqu'alors sur le lieu A. La température devient non pertinente pour l'application et est inexploitable pour garantir des résultats cohérents. Dans de tels cas, la délocalisation n'est pas conseillée. De la même façon qu'un composant est lié à un périphérique pour des raisons de dépendances matérielles, il peut aussi être lié pour des raisons sémantiques. Un composant peut être délocalisable ou non en fonction de son lien avec un matériel et en fonction de sa pertinence par rapport à un lieu.

3.4.3 Structure des applications

Les objectifs de ces travaux sont de garantir la meilleure QdS à savoir garantir la continuité de service, le respect des exigences d'utilisation et une durée de vie la plus longue possible. C'est pourquoi nous centrons notre modèle d'application sur ses fonctionnalités (figure 3.9). Une application est composée d'un ou de plusieurs services. Au vu de la définition que nous avons retenue pour un service, ce dernier peut être réalisé par un composant ou un assemblage de composants. Il n'existe pas un unique assemblage par service. Selon les composants disponibles, plusieurs combinaisons d'assemblages de composants sont capables de réaliser le service. Les composants communiquent en se transmettant des données ou des flux de données. Ces transmissions sont assurées par l'intermédiaire de connecteurs. Cette distinction entre composants et connecteurs permet aux composants de s'affranchir des propriétés de communication. Ils se relient aux connecteurs par des mécanismes génériques et ne se préoccupent pas du transport des données. Une application est constituée d'un ensemble de services composés d'assemblages variables de composants reliés entre eux par des connecteurs.

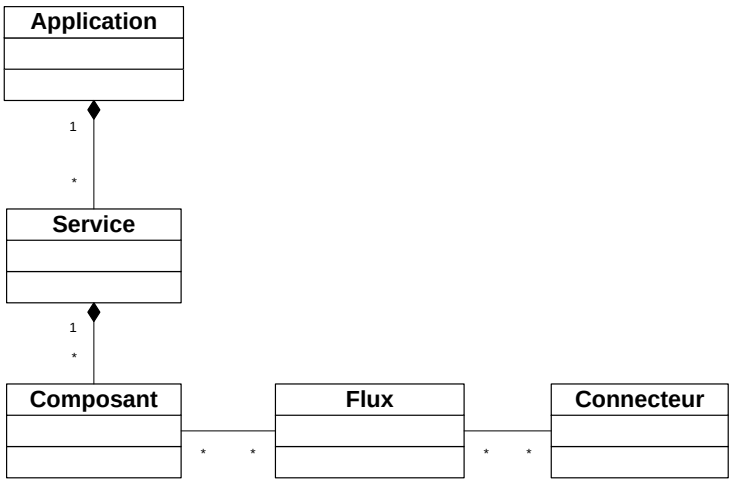

Figure 3.9 – Diagramme de classe d'une application

3.5 Méthode de conception

Maintenant que nous avons défini les modèles de contexte, de QdS et d'application, nous devons définir une démarche de conception afin de guider le concepteur de l'application dans le processus de description des informations nécessaires à la prise en compte de la QdS.

3.5.1 Etape 1 : Identification des services de l'application

Une application est destinée à des utilisateurs. Ces utilisateurs ont des envies, des souhaits relatifs à ce qu'ils vont pouvoir faire avec cette application. L'application doit leur fournir ces services. C'est pourquoi la première étape de cette démarche consiste à définir tous les services que l'application doit fournir. Cette description est faite en langage naturel à la façon d'un cahier des charges. Dans notre exemple d'application de visite d'un musée, nous identifions tout d'abord trois types d'utilisateurs. Le premier est le visiteur simple. Il souhaite visiter librement le musée au gré de ses envies et obtenir des informations sur les œuvres. Ensuite, il y a le visiteur qui souhaite être guidé afin de ne pas rater une oeuvre incontournable. Enfin, il y a les administrateurs du musée. Ces derniers souhaitent pouvoir gérer au mieux les visites dans le musée en termes de nombre de visiteurs par jour ou en temps réel par exemple. Pour chacun des utilisateurs, l'application fournit trois services :

- Un service de *Description*. Ce service propose à l'utilisateur de décrire l'oeuvre devant laquelle il est arrêté et donne le nom de l'auteur, la date et le lieu de création. Selon le contexte, ce service peut être proposé sous forme vidéo, vocale ou textuelle.
- Un service de *Guidage*. Ce service propose à l'utilisateur de le guider dans sa visite du musée. Il peut soit proposer un circuit prédéfini avec un temps, un plan et des oeuvres imposés, soit fournir un simple plan permettant à l'utilisateur de se repérer dans le musée lors d'une visite libre.

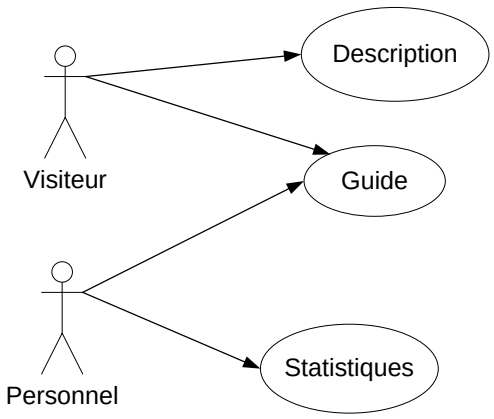

FIGURE 3.10 – Identification des services fournis par l'application de visite d'un musée

– Un service de *Statistiques*. Ce service propose aux administrateurs du musée une série de données concernant le nombre de visiteurs actuellement dans le musée, le nombre de visiteurs dans la journée ou encore la liste des oeuvres pour lesquelles a été demandée le plus souvent une description.

Une fois que les service ont été définis, nous devons spécifier les règles d'utilisation.

3.5.2 Etape 2 : Cas d'utilisation

Lors de l'utilisation de l'application, des problèmes, des changements dans l'environnement peuvent survenir qui sont indépendants de sa volonté et indépendants de son périphérique. Certaines situations demandent une adaptation du fonctionnement de l'application. Cette étape permet de mettre en évidence toutes les situations que peut rencontrer l'application et qui demandent des adaptations.

Chacune de ces situations peut être représentée par un cas d'utilisation. Ceci constitue la représentation des spécifications de l'application à savoir les obligations, les services qu'elle doit fournir pour chaque situation. A partir de ces cas d'utilisation, nous pouvons déduire les éventuelles restrictions quant à son utilisation. Dans le cas de notre application de visite d'un musée, nous identifions deux situations.

La première situation est celle de la conférence, illustrée dans la figure 3.11.

Tout d'abord, le concepteur exprime la situation en langage naturel :

Lorsqu'une conférence est en cours dans le musée, le service de Description ne doit plus proposer de vidéo ou d'audio.

En effet, lors d'une conférence, l'orateur a besoin de calme et les sons provenant des différents périphériques en cours d'utilisation peuvent entrainer des nuisances. De même, les visiteurs du musée peuvent être incomodés par le bruit engendré par la conférence et avoir des difficultés à suivre le discours donné par le service de *Description*. Dans une mesure de confort, le concepteur décide qu'il est préférable de bannir le son pendant la

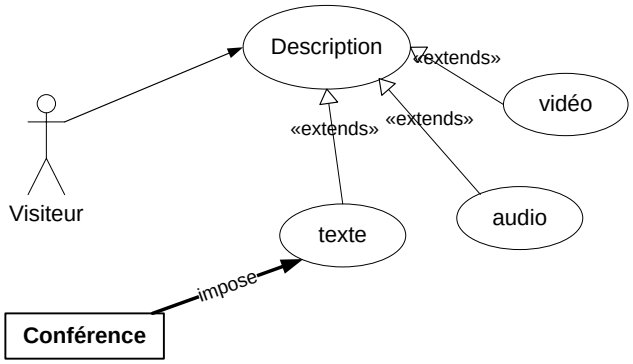

Figure 3.11 – Représentation du cas d'utilisation de la conférence

durée d'une conférence. Ce cas permet de mettre en évidence que lorsqu'un évènement de type conférence est détecté par la plate-forme, l'application ne doit plus proposer dans sa structure de composants qui diffusent des données vidéo ou des données audio. Une conférence est une information que nous ne pouvons pas mesurer puique ce n'est pas une grandeur physique. Elle fait partie des informations abstraites. En revanche une conférence est identifiable à partir d'un ensemble de propriétés. Une conférence est une situation où sur un lieu précis, une personne parle à une assemblée. Elle est donc identifiable par un niveau sonore dans un lieu donné. Tel que nous l'avons décrit dans la section 3.2.2, cet ensemble est représenté sous la forme d'une propriété abstraite. Pour détecter une conférence, nous créons donc une propriété abstraite conférence. La propriété abstraite ≪ conférence ≫est composée d'un niveau sonore en décibel et d'une localisation identifiée par le nom de la salle du musée. Le niveau sonore moyen engendré par une conférence est d'environ 70 dB. Par conséquent, la valeur maximum de la propriété niveau sonore est fixée à 70 dB. Quand, dans une salle donnée, le seuil est dépassé et que le visiteur se trouve dans la même salle, la propriété abstraite conférence déclenchera un évènement contextuel nommé ≪ *conférence* ≫. Lorsque la plate-forme recevra cet évènement (figure 3.12), elle déclenchera le processus d'évaluation de QdS afin de reconfigurer l'application dans le but de ne proposer qu'une version sans son du service de *Description* comme :

– un flux d'images sous-titré
– un texte.

La deuxième situation est celle de l'incendie, illustrée dans la figure 3.13.

Elle est exprimée par :

Lorsqu'un incendie est détecté dans le musée, arrêter le service de Description et/ou adapter le service de guide afin d'afficher le trajet jusqu'à la sortie (passer du mode visite guidée ou visite libre au mode évacuation).

Afin d'évacuer le musée efficacement en cas d'incendie, l'idée est de proposer à l'utilisateur un plan montrant le chemin vers la sortie de secours la plus proche. Ce cas permet de mettre en évidence que lorsqu'un évènement de type incendie est détecté par la plate-forme, l'application doit proposer le service *Guidage* avec un plan vers la sortie de secours.

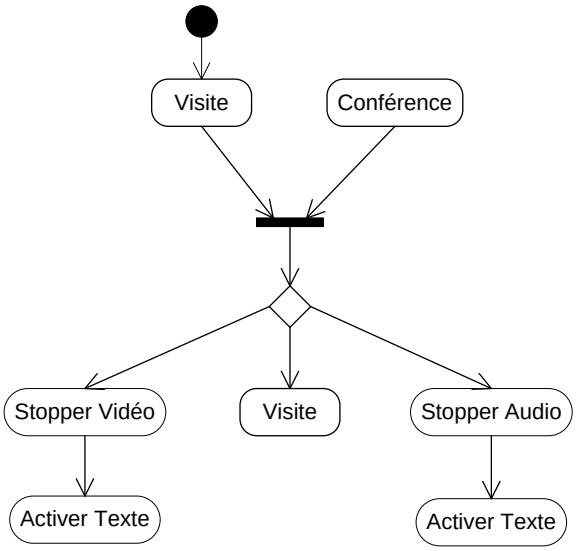

FIGURE 3.12 – Réaction suite à l'évènement *Conference*

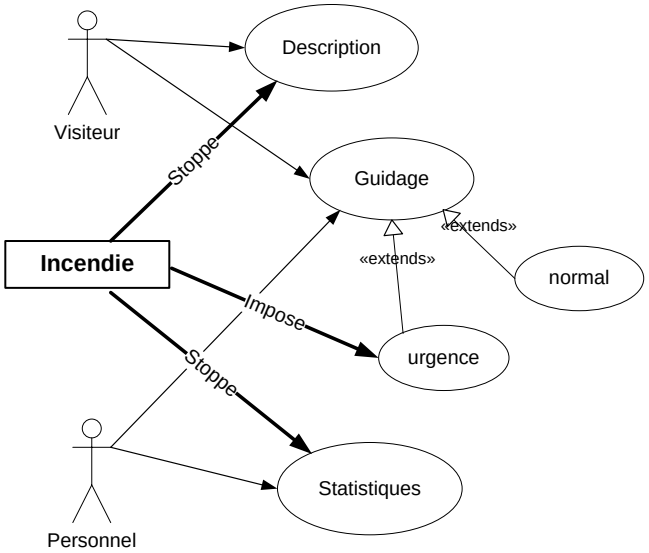

FIGURE 3.13 – Représentation du cas d'utilisation de l'incendie

Ceci implique donc que si le service de *Description* est en cours d'utilisation, la plate-forme doit le stopper et le remplacer par le service *Guidage*. Si le service de guide est en cours d'utilisation, la plate-forme doit adapter le contenu des données qu'il transmet à savoir, ne plus diffuser la visite guidée mais un plan de sortie. La plate-forme doit donc proposer une nouvelle structure de l'application qui propose le service *Guidage* dans sa version de guide vers la sortie. Tout comme le cas de la conférence, un incendie n'est pas mesurable,

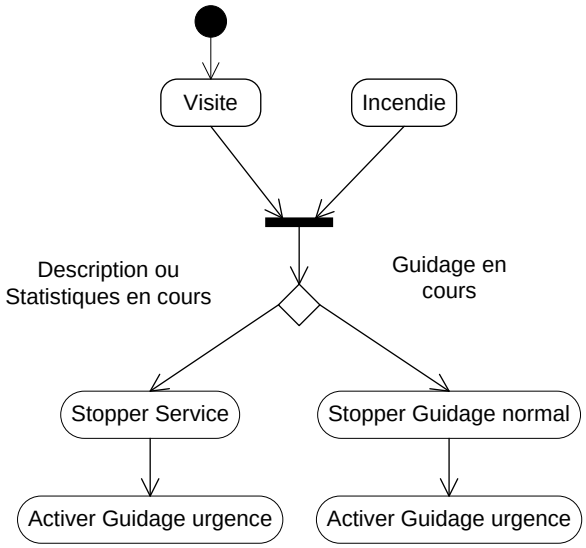

Figure 3.14 – Réaction suite à l'évènement *Incendie*

c'est une information abstraite. Un incendie peut être détecté de la manière suivante : il est le résultat d'une température élevée et de la présence de fumée. La propriété abstraite incendie est alors composée de deux propriétés, la propriété température et la propriété fumée. Lorsque la valeur des deux propriétés dépasse les seuils définis, l'ensemble provoque l'apparition de la propriété abstraite incendie qui déclenche l'envoi d'un évènement contextuel *Alarme*. On peut remarquer ici que, contrairement à l'exemple précédent où la propriété abstraite était localisée, la détection d'incendie peut se faire en un lieu quelconque du musée. La propriété abstraite *incendie* peut provenir de n'importe quel couple de capteurs de température et de fumée. Lorsque la plate-forme reçoit l'évènement *Alarme* (figure 3.14), elle déclenche le processus d'évaluation de l'application afin de détecter tout d'abord quels sont les services en cours d'exécution, les arrêter et proposer une solution constituée uniquement du service de *Guidage* dans sa version plan de sortie de secours.

3.5.3 Etape 3 : Décomposition des services

Nous avons défini les services fournis par l'application et décrit les situations d'utilisation qui demandent une adaptation de l'application afin de fournir un service conforme aux diverses exigences. Ces cas d'utilisation sont une première base pour la décomposition fonctionnelle des services.

Lorsqu'on décompose les services, selon le nombre de composants disponibles, le nombre d'assemblages possibles pour réaliser chaque service peut être élevé. Lors de la recherche de la configuration offrant la meilleure QdS, plus le nombre d'assemblages à tester est grand, plus le délai pour obtenir le meilleur assemblage est long. Or, dans le cas des périphériques contraints, le temps est un facteur critique puisque le niveau de batterie baisse au cours de l'exécution de l'application. Nous devons alors déterminer un moyen de réduire le temps d'évaluation des assemblages. Par exemple nous pouvons réduire le nombre d'assemblages

à tester.

Lorsqu'une reconfiguration est nécessaire, il est important de connaitre la fonctionnalité ou le composant qui fait défaut. En effet, dans notre premier cas, si une conférence est en cours dans le musée, le service de *Description* ne doit plus proposer d'audio. Il parait donc inutile de proposer à l'outil d'évaluation de QdS des composants ou des assemblages de composants dont la fonctionnalité est d'acquérir de traiter ou de restituer des données audio. Dans ce cas, nous nous situons au niveau des fonctionnalités des composants. La fonctionnalité son doit être évitée si cela est possible. Nous devons alors proposer en premier lieu à l'outil d'évaluation des assemblages dépourvus de composant proposant une fonction audio.

Pour cette première décomposition, il s'agit de proposer une distinction entre les différents assemblages constituant les services en familles de configurations afin de guider l'évaluation de QdS au niveau des fonctionnalités à fournir. Une telle décomposition permet d'éviter des évaluations inutiles et de réduire le délai d'obtention de l'assemblage de meilleure QdS. Nous définissons une famille de configuration comme l'ensemble des assemblages de composants répondant à une même fonctionnalité. Nous nous basons sur les cas d'utilisation pour déterminer les familles. Dans notre exemple, pour le service de *Description*, nous identifions trois familles de configuration : la famille image, la famille audio et la famille texte. Lorsque l'application sera dans la situation du premier cas, la plate-forme choisira d'évaluer des configurations parmi la famille texte puisque video et audio sont exclues selon les règles définies.

Une fois les familles de configurations identifiées, nous devons, pour chaque service, établir toutes les décompositions possibles capables de réaliser le service. Selon les composants disponibles, un service peut être composé de différentes façons. Plusieurs assemblages peuvent rendre le même service mais différent par leur qualité. Prenons l'exemple du service de *Description*. Ce service est disponible en trois versions : une version vidéo, une version audio et une version texte qui correspondent aux trois familles de configuration que nous venons d'identifier. Pour chacune des familles, nous décomposons le service. Un premier assemblage capable de réaliser le service de *Description* est un assemblage constitué d'un composant de restitution vidéo. Cet assemblage peut être complété d'un composant de compression, suvi d'un composant de décompression vidéo ou encore il peut être enrichi par un composant de conversion en noir et blanc ou par un composant de réduction de taille d'image ou de réduction du nombre de couleurs. Un autre assemblage capable de réaliser le service de *Description* est celui composé d'un composant de restitution audio. Tout comme pour la vidéo il peut être enrichi d'un composant de compression suivi d'un composant de décompression audio ou encore d'un composant de traduction. Enfin, nous pouvons également choisir d'utiliser un assemblage constitué d'un composant de restitution de texte auquel nous pouvons ajouter un composant de traduction. Pour chacun des assemblages, il est impératif de définir les éventuelles contraintes comme les contraintes de précédence. En effet, l'utilisation d'un composant de compression exige qu'il soit suivi par un composant de décompression.

3.5.4 Etape 4 : Classement des configurations

Nous avons décrit toutes les configurations possibles pour chacun des services. Ces configurations sont toutes équivalentes d'un point de vue fonctionnel, dans le sens où elles rendent toutes le même service. Cependant chacune des configurations propose des qualités différentes. Le choix d'une configuration ne doit pas se faire au hasard.

Dans le contexte spécifique des périphériques contraints, le temps est précieux puisque l'évolution des ressources est très rapide. Nous ne pouvons pas nous permettre de perdre du temps à tatonner parmi les configurations possibles afin de trouver la configuration la mieux adaptée. C'est pourquoi nous proposons de classer les configurations.

Ce classement a pour but de classer les configurations selon leur qualité. Ainsi il sera un guide pour la plate-forme lorsqu'elle exécutera le processus d'évaluation de QdS à la recherche de la solution la mieux adaptée. Le classement est effectué selon la qualité d'utilisation du service, c'est à dire selon la configuration qui offre la meilleure utilisation du service.

Le niveau de qualité des configurations diffère selon le type d'application conçue. En effet, si nous prenons l'exemple d'un service qui transmet des données vidéo, les exigences seront différentes selon qu'il s'agisse d'une application de surveillance ou d'une application de tourisme. La première privilégie un service qui transmet les images rapidement, en temps réel, alors que l'autre se concentrera plutot sur la taille de l'image ou le nombre de couleurs. De ce fait le classement ne peut pas être automatisé et est laissé au libre arbitre du concepteur.

En tête de liste du classement, nous trouverons la configuration proposant la meilleure qualité et en queue de liste sera positionnée la configuration fournissant la moins bonne qualité. Par ailleurs, comme nous indiqué en section 3.3.1.1, ce classement sera dynamiquemet modifié par des évènements contextuels de façon à respecter les contraintes d'utilisation de l'application. Cependant ce classement n'est pas suffisant pour faire un choix de configuration. A cette étape de la conception, nous n'avons pris en compte que la dimension Utilité de la QdS. Le classement peut comporter des équivalences de qualité entre certaines configurations. Le classement sera complété par la suite par la dimension Pérennité de la QdS. Néanmoins, la prise en compte de la durée de vie de l'application ne peut pas être faite au moment de la conception. Ce critère évolue au cours de l'exécution en fonction de la consommation des ressouces. Ce n'est qu'au moment de l'exécution que nous pourrons évaluer les configurations selon le critère de Pérennité.

3.5.5 Etape 5 : Identification des évènements de l'application

Afin de pouvoir adapter le comportement de l'application en fonction des évolutions du contexte, il faut pouvoir capturer ce contexte et le transmettre à la plate-forme de supervision. Lors de la définition du modèle de contexte, nous avons mis en évidence deux types d'informations contextuelles : les propriétés mesurables et les propriétés abstraites. Dans cette étape, nous nous intéressons aux propriétés abstraites. Les propriétés abstraites sont relatives aux contextes *utilisateur* et *utilisation*. Elles sont le résultat des interactions de l'utilisateur avec l'application et des influences de l'environnement physique sur l'application.

Par exemple, un utilisateur souhaite utiliser le service de *Description*. Il va interagir avec l'application afin de demander la mise en place du service. Cependant l'application n'a pas le pouvoir d'agir sur la création et la destruction des services puisque nous avons choisi de piloter les applications par une plate-forme de supervision. Dans ces conditions, l'application ne traite uniquement des données fonctionnelles et relaie tout autre type d'information à la plate-forme. L'application transmet donc l'information de demande de création du service *Description* à la plate-forme. Pour que la plate-forme perçoive ces informations et qu'elle puisse réagir en conséquence, elle doit avoir en sa possession une description complète de toutes les informations qu'elle sera amenée à traiter.

Dans cette étape, le concepteur décrit toutes les propriétés abstraites auxquelles l'application doit réagir concernant les contextes utilisateur et utilisation selon le modèle de contexte que nous avons défini. Décrivons dans un premier temps tous les propriétés abstraites que peut produire un utilisateur par l'intermédiaire de l'IHM dans notre exemple d'application de visite d'un musée. Chaque clic de souris sur une case bien précise de l'IHM produit les propriétés abstraites suivants :

- Demande d'un service. Lorsqu'un utilisateur souhaite utiliser un service, par exemple le service *Description*, il en fait la demande auprès du serveur de l'application.
- Arrêt d'un service.
- Traduction. Le service de *Description* peut être adapté selon la langue de l'utilisateur. Nous définissons alors une propriété abstraite *Traduction* afin de pouvoir prendre en compte les préférences de langue de l'utilisateur pour les services *Description* et *Guidage*.

Dans un deuxième temps, nous décrivons les propriétés abstraites produites par les changements de l'environnement physique de l'application. Ce sont celles que nous avons identifiées dans la deuxième étape de la méthode (section 3.5.2) lors de la description des différents cas d'utilisation de l'application. Pour capturer ces informations de *contexte d'utilisation*, nous avons besoin d'être constamment à l'écoute de l'environnement. Pour cela nous mettons en place des composants spécifiques appelés *composant de contexte* permettant de garantir le respect des contraintes répertoriées dans les cas d'utilisation. Ces composants sont distribués sur tous les périphériques supportant l'application lors du premier déploiement et restent actif jusqu'à l'arrêt de l'application. Dans le cas de notre exemple, nous avons identifié deux cas d'utilisation : le cas de la conférence et le cas de l'alarme incendie. Pour la conférence, nous fixons un seuil maximal d'une valeur de 70dB à la propriété niveau sonore. Nous associons ensuite cette propriété à la propriété abstraite *Conférence* de la façon suivant :

$$Si \text{ niveau sonore} \geqslant 70dB \Rightarrow Conférence$$

$$Si\ Conférence \Rightarrow \text{évènement}\{\text{Conférence}\}.$$

Cette règle définit que lorsque la valeur du niveau sonore atteint ou dépasse 70 dB, alors les conditions requises pour la détection de la propriété abstraite *Conférence* sont atteintes et le composant de contexte envoie un évènement contextuel nommé *Conférence* à la plate-forme.

De la même manière nous pouvons définir le propriété abstraite d'incendie par l'association d'une température et d'une quantité de fumée. Une température élevée ne signifie

pas forcément qu'il y a un incendie, cependant si en plus de cette forte température, de la fumée est détectée, alors il est probable qu'un incendie se soit déclaré. Nous associons ces deux propriétés à la propriété abstraite d'incendie de la façon suivante :

$$Si \text{ température} \geqslant 60^{\circ} ET \text{ fumée} = vrai \Rightarrow Incendie$$

$$Si \ Incendie \Rightarrow \text{évènement}\{\text{Alarme}\}$$

Cette régle définit que lorsque la valeur de la température atteint ou est supérieure à 60 degrés celsius et si de la fumée est détectée, alors les conditions requises pour la détection de la propriété abstraite d'incendie sont valides. La propriété abstraite d'incendie est produit, le composant de contexte envoie l'évènement *Alarme* à la plate-forme..

Ces règles permettent de définir toutes les propriétés que les composants de contexte doivent surveiller afin d'en informer la plate-forme et lui permettre de reconfigurer l'application si besoin.

3.5.6 Etape 6 : Cartes d'identité des composants et des périphériques

Dans la section 3.2, nous avons identifié les différentes sources d'informations contextuelles. Si nous résumons, le contexte est capturé soit par les composants logiciels, soit par la plate-forme installée sur tous les périphériques et qui peut ainsi contrôler leurs ressources. Afin de pouvoir évaluer la QdS, nous avons besoin de connaitre toutes les caractéristiques des composants et des périphériques ce qui est réalisé par des cartes d'identité.

Chaque carte d'identité est composée de trois parties : l'identification du composant, une partie statique et une partie dynamique. Un composant est identifié par :

- son nom
- sa fonction, est-ce un composant de restitution d'images, de son, un composant de compression de données, de mesure de luminosité, etc.
- son type :
  - Fixe
  - CDC c'est à dire que ses exigences lui permettent d'être exécuté sur un périphérique dont les limitations de ressources répondent au standard CDC de J2ME [36].
  - CLDC, c'est à dire que ses exigences lui permettent d'être exécuté sur un périphérique dont les limitations de ressources répondent au standard CLDC de J2ME [37].
- sa dépendance ou non à un périphérique

La partie statique regroupe les informations qui ne changent pas au cours du temps. Pour un composant, dans la partie statique nous retrouvons ses caractéristiques techniques à savoir :

- le nom du système d'exploitation qu'il utilise
- la part de CPU qu'il utilise pendant son exécution
- l'espace mémoire dont il a besoin pour être installé et exécuté
- le débit réseau moyen qu'il fournit en sortie.

Dans la partie dynamique, nous retrouvons toutes les informations liées à l'éxécution et au déploiement :

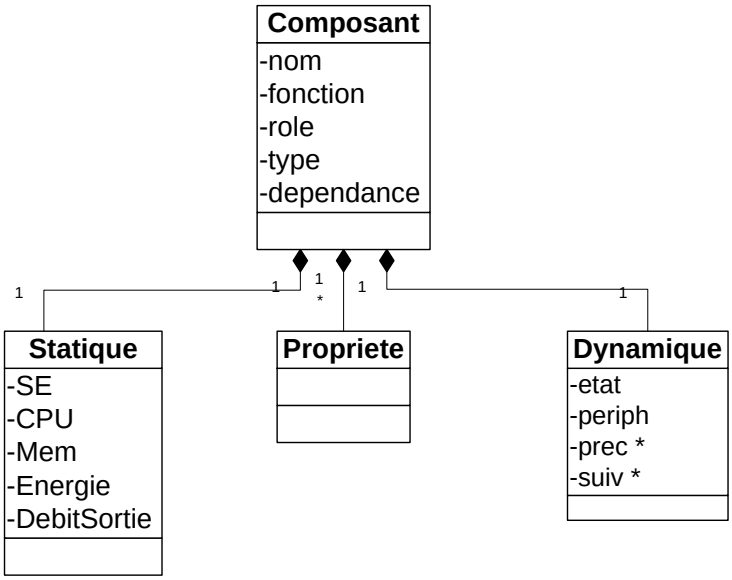

FIGURE 3.15 – Carte d'identité d'un composant

- son état, c'est à dire s'il est installé, en cours d'exécution ou arrêté
- le périphérique sur lequel il est installé
- les périphériques sur lesquels sont installés les composants qui le précèdent et qui le succédent dans la configuration.

Ces caractéristiques font partie du *contexte d'exécution* et sont nécessaires pour évaluer la *QdS Pérennité*. Les composants ont également une influence sur l'utilisation de l'application et donc sur le critère de *QdS Utilité*. La carte d'identité est également composée d'un ensemble de propriétés fonctionnelles liées à la partie métier des composants. Dans ces propriétés nous décrivons toutes les informations sur la fonction du composant telles que le codec utilisé pour un composant vidéo, le nombre d'images par seconde ou le nombre de couleurs, s'il mesure une température ou un niveau sonore.

Un périphérique est identifié par un nom, un type et un système d'exploitation. Le type peut prendre trois valeurs : Fixe, CDC ou CLDC. Etant donné que nous utilisons des périphériques mobiles, nous avons choisi de nous référer au standard industriel le plus répandu dans le domaine de la programmation d'applications pour les périphériques mobiles, Java 2 Mobile Edition. J2ME propose deux standards en matière de programmation d'applications et sont identifiés par les ressources disponibles par les périphériques. Sa partie dynamique décrit l'état de ses ressources :

- le niveau d'énergie restant
- l'espace mémoire disponible
- le taux de charge du CPU

Les données que nous venons de décrire font partie du *contexte d'exécution* et n'influencent que la *QdS Pérennité*. Or, certaines propriétés des périphériques peuvent influencer la *QdS Utilité* telles que la taille de l'écran par exemple. Nous décrivons alors toutes

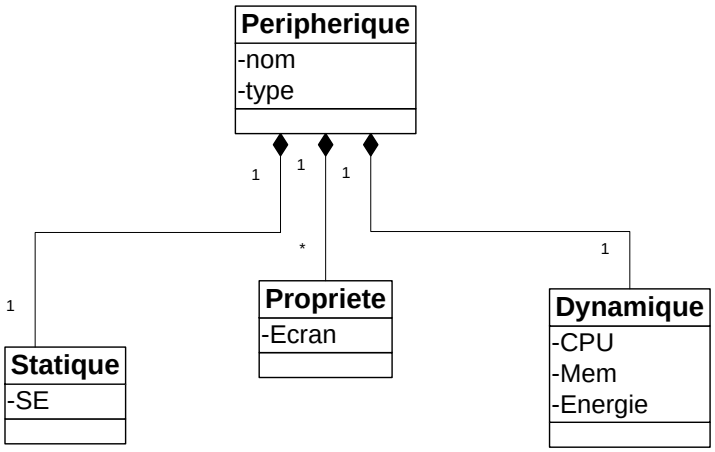

Figure 3.16 – Carte d'identité d'un périphérique

les propriétés fonctionnelles du périphérique.

3.6 Conclusion

Guidés par notre volonté de proposer une gestion de la qualité de service adaptée aux applications pervasives, nous présentons un modèle d'évaluation de la qualité de service centrée à la fois sur l'utilité et la longévité des applications permettant d'exploiter au mieux les possibilités offertes par les périphériques. L'utilité réfère à la définition donnée par les fonctions d'utilité [4][82] et à celle donnée par [42]. Elle représente l'adéquation du fonctionnement d'une application face aux attentes de l'utilisateur. La longévité regroupe la durée de vie de l'application et ce que nous appelons la continuité de service. La durée de vie signifie que l'application doit solliciter le moins possible les ressources matérielles et réseau. La continuité de service signifie que l'application doit fonctionner malgré les variations du contexte.

Les contributions de ce chapitre permettent de répondre à l'utilité et à la durée de vie de l'application au moyen d'un modèle d'évaluation de la QdS d'une part et d'un modèle de contexte couplé à une démarche de conception d'autre part. Bien que l'utilisateur ne se situe pas au centre de ces travaux, le modèle d'évaluation de la QdS procède néanmoins en deux temps : étude de la *QdS Utilité* puis étude de la *QdS Pérennité*. La particularité de ce modèle d'évaluation de la QdS est qu'il propose une évaluation de la durée de vie de l'application basée sur deux coûts :

- La coût de la consommation des ressources matérielles par les composants
- Le coût des liaisons réseau entre les composants

Cette double évaluation permet ainsi de prendre en compte dans sa totalité le caractère contraint des périphériques lors du déploiement des applications. De plus, grâce aux seuils

de qualité de service, le modèle d'évaluation de QdS permet à la plate-forme d'être elle-même sensible au contexte et de s'auto-adapter au cours des changements de contexte.

Pour pouvoir prendre en compte les différentes caractéristiques des périphériques et des composants, nous présentons un modèle de contexte proposant d'établir des cartes d'identité permettant d'identifier les propriétés fonctionnelles et non fonctionnelles de chaque entité. Le modèle de contexte propose également d'identifier des situations de contexte complexes grâce aux propriétés abstraites.

Enfin, nous présentons une méthode de conception permettant d'aider le concepteur à identifier toutes les informations de contexte nécessaire à la plate-forme pour reconfigurer les applications.

Chapitre 4

Kalimucho : plate-forme d'adaptation sensible au contexte pour la gestion de la qualité de service

Sommaire

4.1 Introduction

Dans le chapitre 3, nous avons défini un modèle de contexte afin de classifier les informations capturées, puis un modèle de QdS afin d'obtenir une mesure de l'état de l'application à partir du contexte.

Le chapitre 4 présente les outils associés à ces modèles permettant de capturer et d'interpréter l'information ainsi que les outils permettant d'adapter les applications.

Afin de maîtriser le comportement d'une application il faut pouvoir être capable de connaître toutes les informations sur son fonctionnement et sa structure et de les modifier lorsque c'est nécessaire. Il existe deux grandes catégories dans le domaine de la gestion du contexte et de l'adaptation d'application : l'auto-adaptation et la supervision.

Auto-adaptation Ce type de système est conçu pour pouvoir s'auto-adapter dynamiquement pour s'accomoder d'une variation des ressources, d'une défaillance système ou d'un changement des besoins de l'utilisateur. Auto-adaptation (self-adaptation) signifie que les adaptations sont gérées par l'application elle-même [41]. Elle doit évaluer son propre comportement, sa configuration ainsi que son déploiement. Si on considère le triplet capture, évaluation, décision, une application auto-adaptable capture son contexte, l'analyse et modifie son comportement. La figure 4.1 représente les interactions entre une application auto-adaptable et le contexte. Tout d'abord, l'application capture le contexte (flux n° 1) puis adapte son comportement (flux n° 3). Enfin, l'exécution de l'application modifie le contexte (flux n° 2).

Cependant, dans le cas des applications distribuées, l'application n'a d'interaction qu'avec le contexte local à son périphérique. Pour pouvoir interagir avec l'ensemble du contexte, il est nécessaire de mettre en place des mécanismes spécifiques pour tous les périphériques supportant l'application. Par conséquent, l'ajout de ces mécanismes augmente la complexité de l'application et rend sa maintenabilité plus difficile puisqu'elle mèle à la fois la gestion de la logique métier et la gestion des propriétés non-fonctionnelles. Plus encore, pour pouvoir évaluer sa configuration et son déploiement, l'application a besoin de recueillir les descriptions des composants - la ou les fonctionnalités qu'il propose et leurs QdS, les dépendances avec d'autres composants - la description de son déploiement actuel et des différentes autres configurations possibles. Elle doit également prévoir un planning pour mettre à jour les données de contexte et évaluer sa configuration. Enfin,

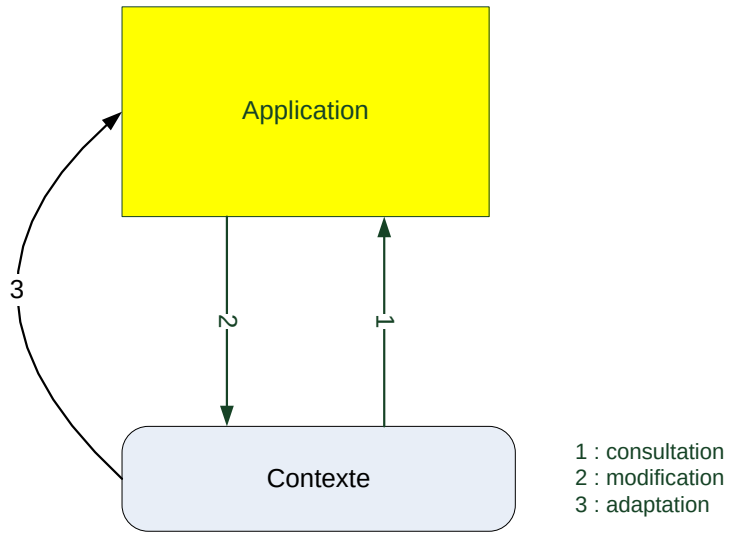

Figure 4.1 – Interactions dans les applications auto-adaptables

l'utilisation de ces applications en environnement ubiquitaire et hétérogène empêche la mise en place de solutions génériques ce qui ne permet pas d'exploiter tout le potentiel des périphériques [68]. C'est pourquoi, beaucoup de systèmes choisissent de résoudre ces problèmes par l'utilisation de plate-formes.

Supervision Dans les approches par supervision, une plate-forme d'exécution sert d'interface entre l'application et le contexte. Elle permet d'avoir accès à l'ensemble du contexte même distant. La figure 4.2 représente les interactions entre les trois entités application, contexte et plate-forme. L'application peut seulement capturer le contexte grâce à des mécanismes fournis par la plate-forme (flux n° 1). Elle peut aussi modifier le contexte ainsi que la plate-forme (flux n° 2). Enfin, l'application et la plate-forme s'adaptent au contexte (flux n° 3).

Pour pouvoir adapter l'application dans son ensemble, il est nécessaire de distribuer la plate-forme sur tous les périphériques supportant l'application. Une telle architecture permet de capturer le contexte local et de proposer des adaptations locales. De plus, la communication entre les plate-formes locales permet d'obtenir une vision globale du contexte et donc d'effectuer une mesure globale du contexte et des actions d'adaptation. Pour cela, chaque plate-forme doit effectuer trois tâches :

- Capturer le contexte.
- Analyser le contexte, c'est-à-dire évaluer si une reconfiguration est souhaitable.
- Adapter c'est-à-dire proposer un ensemble d'outils permettant de reconfigurer l'application selon les changements de contexte.

Afin de garantir une meilleure cohérence lors de l'adaptation du comportement des applications, nous avons choisi d'utiliser une plate-forme de supervision qui doit répondre à deux objectifs. Premièrement, elle doit permettre de capturer les trois types de contexte que

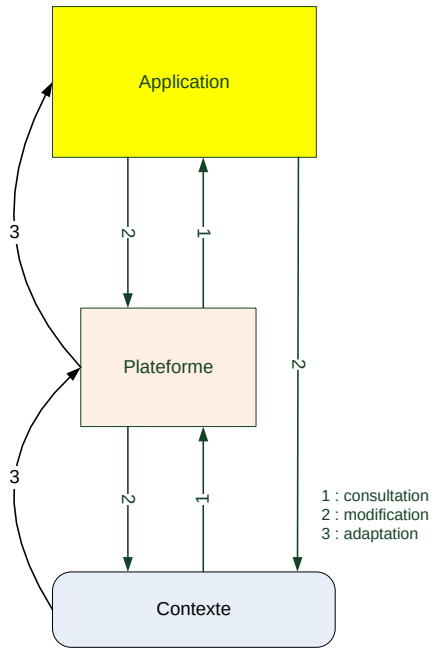

FIGURE 4.2 – Interactions dans les applications supervisées

nous avons définis à savoir le contexte utilisateur, le contexte d'utilisation et le contexte d'exécution. Deuxièmement, elle doit adapter l'application selon le contexte en garantissant une *continuité de service* ainsi qu'une *QdS Utilité* et une *QdS Pérennité* suffisantes. Pour atteindre ces objectifs nous proposons que la plate-forme s'appuie sur deux types d'adaptations : l'adaptation aux besoins et l'adaptation au matériel.

4.1.1 Adaptation aux besoins

Une application doit répondre à plusieurs types de besoin. Premièrement, il y a les besoins décidés lors de la conception. Ce sont les spécifications de l'application sur tous les services qu'elle doit fournir et la façon dont elle doit les fournir. Nous avons donc besoin d'un mécanisme permettant de définir ces spécifications lors de la conception de l'application ainsi qu'un mécanisme permettant de détecter, pendant l'exécution, l'adéquation de la structure en cours avec les spécifications et de réaliser des adaptations lorsque c'est nécessaire.

Deuxièmement, il y a les besoins lors de l'exécution. Ce sont les besoins de l'utilisateur. Tels que nous les avons présentés dans le chapitre 3, les besoins de l'utilisateur sont ici restreints. Nous ne traitons pas les préférences de l'utilisateur comme il est fait dans les applications multimédias par exemple :

- préférence de la couleur au noir et blanc
- préférénce d'une fréquence d'une mesure à la minute plutot que la moyenne toutes les 5 minutes

Dans le cadre de cette thèse, lorsque nous parlons des besoins de l'utilisateur nous parlons uniquement des fonctionnalités qu'il souhaite utiliser parmi celles que lui propose l'application. Dans l'exemple de l'application de visite du musée, il pourra demander à utiliser le service description ou le service guidage ou obtenir une traduction mais il ne pourra pas préciser s'il souhaite de la vidéo, du son ou du texte. Compte tenu du caractère autonome des périphériques concernant les ressources énergétiques, nous décidons qu'il est préférable de gérer les différentes façons de fournir une fonctionnalité à l'utilisateur par rapport à l'état du matériel.

4.1.2 Adaptation au matériel

Afin de gérer au mieux la durée de vie des applications, nous nous intéressons tout particulièrement aux ressources des périphériques supportant l'application ainsi qu'à leur mobilité.

Quand un utilisateur choisit un périphérique, il le choisit pour des propriétés particulières. Par exemple il va vouloir un grand écran couleur, une grande capacité mémoire ou au contraire quelque chose de basique qui lui permette uniquement de recevoir et envoyer des données. Quel que soit son choix, nous considérons qu'il souhaite utiliser son matériel au maximum de ses capacités.

A partir de ce principe, la plate-forme adapte le rendu de l'application par rapport aux capacités physiques du périphérique. Les configurations sont classées dans un premier temps selon un critère intrinsèque, c'est-à-dire indépendant de tout contexte (chapitre 3). Lorsque la plate-forme doit déployer un service, elle teste dans un premier temps la compatibilité matérielle d'une configuration avec les périphériques disponibles. Ainsi, elle choisira une configuration adaptée aux capacités physiques du périphérique ainsi qu'à l'état de ses ressources. Dans l'exemple du service description, la plate-forme vérifie les capacités du périphérique :

- s'il dispose d'un écran et d'un haut-parleur, alors elle choisira une configuration qui propose de la vidéo
- s'il dispose uniquement d'un écran, elle choisira une configuration proposant des images et du texte
- s'il dispose uniquement d'un haut-parleur, elle choisira une configuration ne proposant que du son

L'adaptation aux capacités physiques est une première étape lorsqu'il s'agit de matériel cependant il est également nécessaire de considérer la durée de vie de l'application. La durée de vie d'une application est fonction de l'état des ressources matérielles et réseau. C'est pourquoi il est nécessaire qu'avant chaque choix de configuration, l'état des ressources telles que le taux d'occupation du CPU, le taux d'occupation de la mémoire, le niveau de batterie et la charge du réseau soit mesurés afin d'éviter de mettre en place une configuration qui risquerait de compromettre la durée de vie des périphériques et par conséquent celle de l'application.

4.2 Objectifs

Face à ces deux besoins d'adaptation : adaptation aux besoins et adaptation au matériel, nous proposons deux approches complémentaires : une méthode de conception et une plate-forme de supervision. La méthode de conception que nous avons détaillée dans le chapitre précédent permet de décrire toutes les informations nécessaires à la mise en place de mécanismes de capture et d'analyse du contexte. Elle aide le concepteur à déterminer toutes les propriétés que la plate-forme doit surveiller et toutes les règles d'utilisation nécessaires au déclenchement des évènements de contexte. Elle aide également à déterminer les associations entre un évènement et l'action à mener afin que le comportement de l'application prenne en compte le changement de contexte.

Cette première phase du développement de l'application guidée par la méthode de conception est une phase de modélisation du contexte telle que le décrivent le modèle de contexte et le modèle d'évaluation de QdS [43][64][12][74].

Une fois que nous savons quels types d'information vont circuler entre l'application et la plate-forme, nous devons mettre en place des mécanismes permettant de recueillir ces informations, de les traiter et surtout de reconfigurer l'application. Afin de limiter les problèmes de cohérence et de séparer les préoccupations fonctionnelles et non-fonctionnelles, ces mécanismes doivent être effectués de manière transparente et globale à l'application. Nous proposons que l'ensemble de ces mécanismes soit assuré par une plate-forme de supervision : la plate-forme Kalimucho. Puisque la méthode de conception permet de modéliser l'ensemble des informations nécessaires à la gestion des deux types de QdS, Utilisation et Pérennité, la plate-forme Kalimucho que nous proposons doit également proposer un fonctionnement permettant de garantir ces deux QdS :

- La configuration choisie par Kalimucho doit être en adéquation avec les spécifications de l'application et les besoins de l'utilisateur.
- Le déploiement choisi par Kalimucho doit garantir la plus longue durée de vie possible à l'application

4.2.1 Gestion de la *QdS Pérennité*

La gestion de la *QdS Pérennité* consiste à garantir une durée de vie la plus longue possible à l'application.

Elle consiste tout autant à prendre en compte les propriétés matérielles et logicielles et la mobilité que de proposer des adaptations en adéquation avec la situation de contexte.

La plate-forme Kalimucho a donc besoin de mécanismes permettant d'avoir connaissance des propriétés matérielles des périphériques, plus précisemment l'état de leurs ressources et l'état de leurs connexions réseau et des propriétés logicielles des composants c'est-à-dire leurs taux de consommation des ressources physiques à l'exécution. Ces propriétés ont été respectivements modélisées dans les cartes d'identité des périphériques et des composants. Ces cartes d'identités devront être stockées sur la plate-forme afin qu'elle puisse y avoir accès à tout moment. Kalimucho doit ensuite décider quel type d'adaptation elle doit mener pour garantir la meilleure *QdS Pérennité*. Reprenons l'exemple du composant qui consomme trop de ressources par rapport à celles disponibles sur le périphérique.

Une première solution est de déplacer ce composant vers un autre périphérique qui lui, dispose de niveaux de ressources suffisants. Si aucun périphérique ne dispose de ressources suffisantes, il faut alors remplacer ce composant par un autre. Dans le premier cas, la plate-forme ne fait que déplacer un composant tout en gardant le même assemblage. C'est ce que nous appelons une migration. Dans le deuxième cas, il faut remplacer le composant, c'est-à-dire qu'il faut trouver un nouvel assemblage réalisant le même service mais dont la composition ne fait pas intevenir le composant en question. Cette opération correspond à déployer une nouvelle configuration de service. Il existe également des composants qui proposent plusieurs QdS. A chaque QdS correspond une note de *QdS Pérennité*. Il est alors possible de paramétrer le composant en fonction du contexte. Ces solutions conviennent également dans le cas de mobilité. Dans le premier cas, il est possible de déplacer le composant inaccessible vers un autre périphérique accessible et qui possède toutes les conditions de ressources. Sinon, il est possible de déployer une nouvelle configuration en prenant en compte la nouvelle topologie du réseau.

4.2.2 Gestion de la *QdS Utilité*

La gestion de la *QdS Utilité* consiste à fournir un service qui réponde aux spécifications de l'application et aux souhaits de l'utilisateur. C'est également fournir un service qui exploite au mieux les propriétés matérielles des périphériques telles qu'un écran, un nombre de couleurs, etc. Pour fournir un tel service, la plate-forme Kalimucho a besoin de mécanismes lui permettant d'accéder à trois types d'informations.

Elle doit pouvoir accéder aux spécifications de l'application. Les spécifications décrivent des situations particulières d'utilisation qui nécessitent des adaptations. Par exemple, à l'heure de fermeture du musée, tous les services en cours d'exécution doivent être arrêtés et remplacés par un service guidant le visiteur vers la sortie du musée.

Elle doit pouvoir accéder aux demandes des utilisateurs en termes d'utilisation de fonctionnalités. Par exemple, la langue par défaut dans l'utilisation du service de description des œuvres est le français. Le visiteur qui ne comprend pas cette langue souhaite une traduction de la description dans une langue de son choix. Depuis le service de l'application de visite du musée, il choisit d'utiliser la fonctionnalité traduction. La plate-forme doit alors prendre en compte cette demande et proposer une modification du service en cours fournissant la traduction.

Enfin, la plate-forme doit pouvoir accéder aux propriétés matérielles et logicielles des périphériques et des composants. Ces données enregistrées dans les cartes d'identité de ces derniers permettent d'éviter d'une part les incompatibilités entre composants et périphériques comme par un exemple un composant d'acquisition vidéo qui se trouverait sur un téléphone portable qui ne dispose pas de caméra. D'autre part cela permet d'éviter une sous-exploitation des périphériques comme diffuser la description des œuvres sous format texte alors que le périphérique de l'utilisateur est un smart-phone doté d'un lecteur vidéo et d'un écran large haute définition.

La plupart des adaptations nécessaires à la gestion de la *QdS Utilité* consistent à ajouter ou supprimer des composants et parfois même à déployer des services entiers comme dans le cas de la fermeture du musée.

4.2.3 Principes de fonctionnement

Pour répondre aux attentes décrites dans les paragraphes précédents, nous avons choisi de piloter nos applications par une plate-forme de supervision appelée Kalimucho. Kalimucho est constituée de cinq services collaborant (figure 4.3) et est distribuée sur tous les périphériques supportant l'application afin d'avoir une vision globale de cette dernière. Kalimucho est une plate-forme intrusive qui est capable de répondre aux changements de contexte en modifiant la structure de l'application à deux niveaux. Au niveau de la composition des services, elle est capable d'ajouter ou de supprimer des composants et de proposer des configurations de différentes qualités tout en réalisant le même service. Au niveau du déploiement des services, elle est capable de déplacer des composants et donc de modifier les connexions tout en gardant la même architecture.

L'ensemble de ces cinq services doit permettre à Kalimucho de garantir les quatre objectifs suivants :

1. Elle doit tout d'abord permettre de capturer le contexte. Les évènements peuvent provenir de l'application ou de la plate-forme elle-même. Elle doit disposer de mécanismes capables de capter ces évènements, de les interpréter et de prendre la décision adéquate pour l'adaption à apporter à l'application. Ces mécanismes correspondent à la surveillance du fonctionnement de l'application et sont assurés par le service *Superviseur*.
2. Lorsque la décision de reconfiguration est prise, la plate-forme doit alors pouvoir proposer un schéma de déploiement viable. Pour cela elle doit connaitre l'ensemble des composants logiciels et des périphériques disponibles et évaluer si la solution trouvée est conforme aux exigences de QdS. Cette évaluation est assurée par le service *Générateur de Reconfiguration*.
3. La reconfiguration implique des déplacements, ajouts et suppression de composants. Afin de garder une application efficace, il faut s'assurer de la viabilité des connexions réseau entre les périphériques supportant l'application. Kalimucho a donc besoin d'un service permettant de tenir à jour la topologie réseau de l'application. Cette opération est assurée par le service *Routage*.
4. Enfin, nos applications ont vocation à pouvoir être utilisée avec n'importe quel périphérique. Tous les périphériques ne disposent pas des mêmes ressources matérielles et logicielles. Nous devons donc gérer cette hétérogénéité entre les périphériques. Les composants doivent pouvoir fonctionner sur tout type de périphérique. Nous utilisons un modèle de composant où chaque composant est encapsulé dans un conteneur ad-hoc, adapté au périphérique [47], lui permettant de s'affranchir de toutes les propriétés non fonctionnelles. Les conteneurs ont une durée de vie limitée à l'utilisation du composant. Ils sont créés lorsque que le composant est installé et détruits lorsque que le composant est détruit. Le conteneur doit alors être adapté au périphérique sur lequel le composant est installé. La plate-forme a besoin d'un service capable de créer des conteneurs au cas par cas en fonction du composant qu'il englobe et du périphérique qui l'accueille. C'est le service *Usine à Conteneur*. De la même façon, les composants étant connectés par des connecteurs, les connecteurs ont aussi une durée de vie limitée à l'utilisation du composant. Tout comme les composants, les connecteurs sont encapsulés dans un conteneur. Chaque conteneur de connecteur est

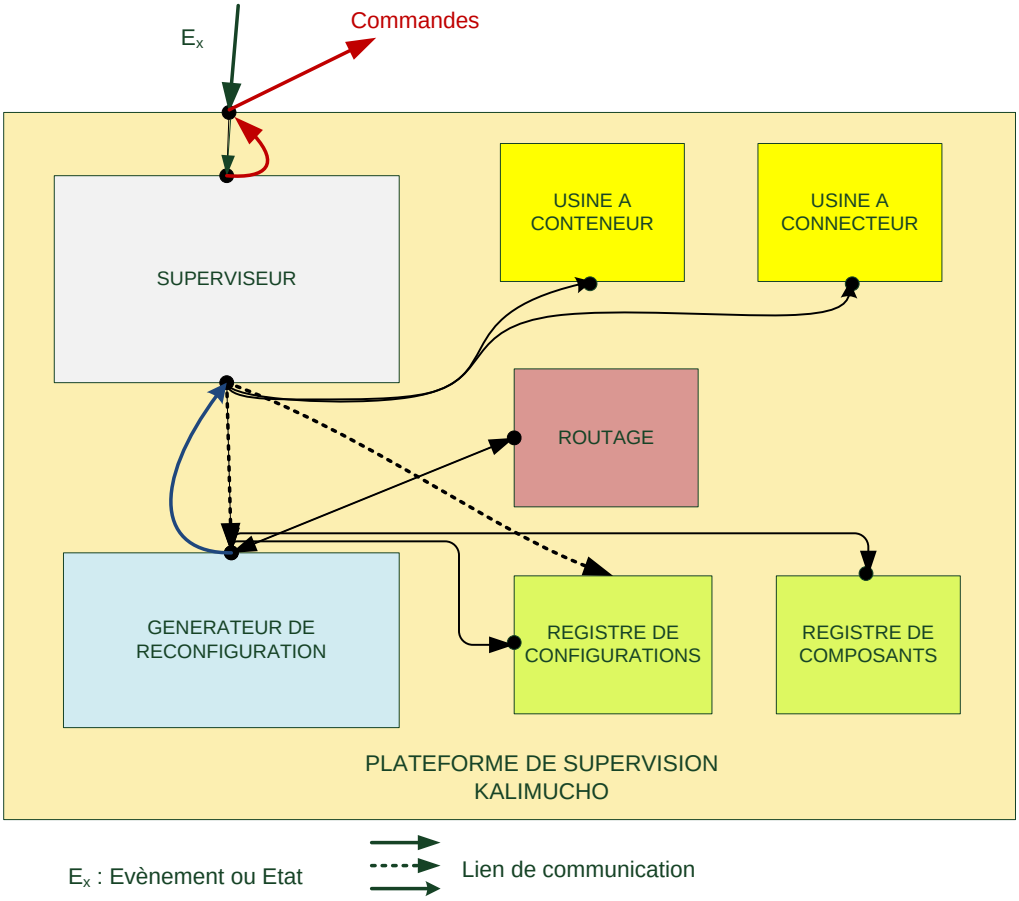

Figure 4.3 – Schéma de la plate-forme Kalimucho

créé lorsque la connexion est mise en place et détruit lorsqu'elle est supprimée. Tout comme les conteneurs, ils doivent être adaptés aux périphériques et sont fournis par le service *Usine à Connecteur*.

4.3 Les moyens d'action

Une application est constituée de services réalisés par des assemblages de composants. Adapter l'application consiste alors à modifier les composants, leurs connexions et la composition des services. Nous identifions trois moyens d'action permettant d'agir sur la structure d'une application.

La migration de service : consiste à modifier la distribution d'un service sans modifier ses composants. Les composants sont déplacés, avec leur état, d'un périphérique vers un autre sans aucun remplacement, ajout ou suppression de composants (figure 4.4).

Le déploiement d'un service : consiste à proposer une nouvelle composition et donc un nouvel assemblage pour réaliser un service (figure 4.5).

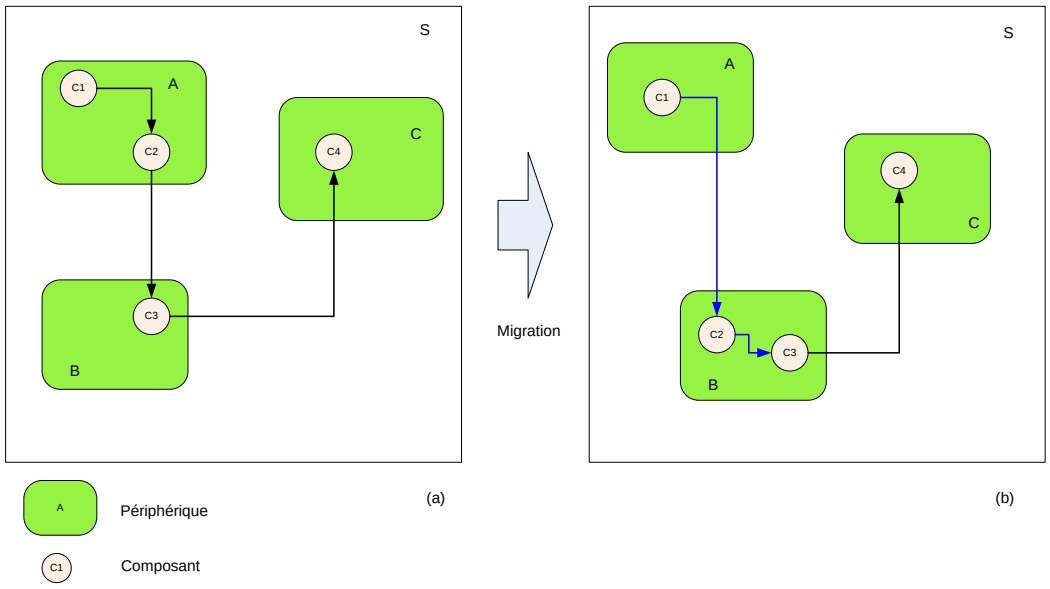

FIGURE 4.4 – Exemple d'une migration d'un service S

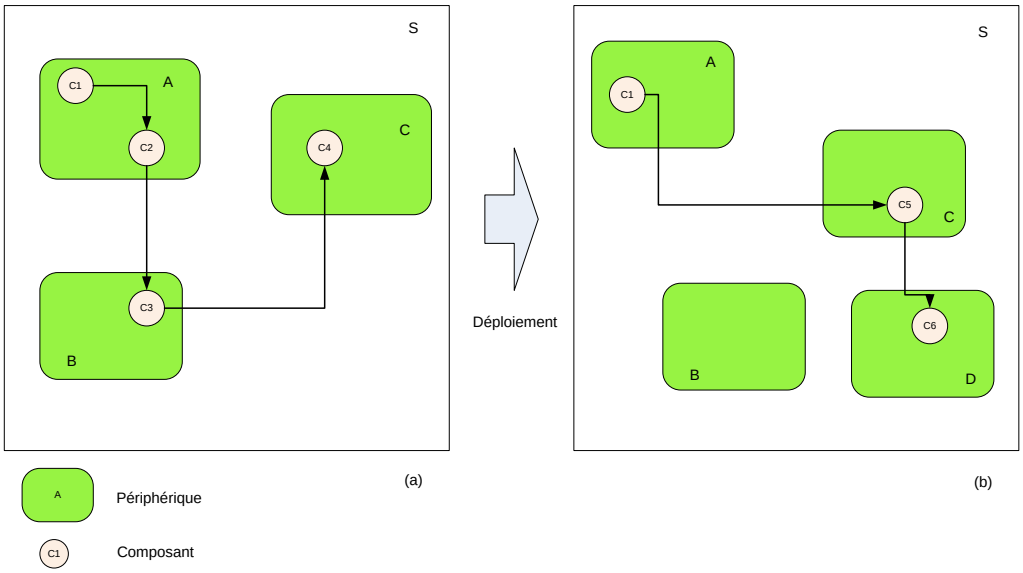

FIGURE 4.5 – Exemple d'un déploiement d'une nouvelle configuration d'un service S

L'ajustement de composant : consiste à ajuster la qualité de service d'un composant de l'assemblage. Dans le cas des composants paramétrables, la configuration du composant est modifiée. Sinon, l'ajustement consiste à remplacer un composant par un autre composant offrant la même fonctionnalité mais avec une qualité de service différente (supérieure, inférieure ou équivalente). Un composant peut proposer plusieurs qualités de service en proposant plusieurs fonctionnements.

4.3.1 Migration de service

La migration de service est une redistribution des composants réalisant le service sans modification de l'assemblage. Les composants sont les mêmes et l'architecture de l'application reste inchangée, simplement ils vont être déplacés vers d'autres périphériques.

Par exemple, comme l'illustre la figure 4.4(a), un service S est composé de quatre composants respectivement C1, C2, C3, et C4. Ces composants sont distribués et reliés selon cet ordre sur les périphériques A, B et C. Soudain B est en difficulté, son niveau d'énergie a atteint un seuil critique, le niveau de *QdS Pérennité* de l'application chute et la continuité du service est compromise. Le composant C3 doit alors être migré vers un autre périphérique. Si la *QdS Utilité* est toujours bonne, il faut alors tenter d'atteindre un meilleur niveau de *QdS Pérennité*.

Rappelons que pour les périphériques contraints, les transmissions réseau sont beaucoup plus coûteuses en énergie que les traitements. Dans un premier temps, la migration va être évaluée en utilisant les périphériques restants déjà supports du reste du service concerné c'est-à-dire dans l'exemple de la figure 4.6, les périphériques A et C.

Si la migration sur A et C ne permet pas d'atteindre un niveau satisfaisant de *QdS Pérennité*, il faut alors évaluer la configuration du service S en utilisant un ou plusieurs périphériques supplémentaires voisins de A et C selon l'heuristique d'évaluation de QdS de l'application (section 4.7.1.2). Par exemple dans la figure 4.7, D est situé à un saut de A et de C.

Enfin, si malgré l'introduction de périphériques voisins la migration n'est pas possible, il faut alors évaluer une autre solution comme choisir une nouvelle configuration pour le service S. C'est le déploiement de service.

4.3.2 Ajustement de composant

Un composant est défini par son type, sa fonctionnalité et sa qualité de service. Dans le cas des composants paramétrables, il est possible d'ajuster leur niveau de QdS. Il suffit que le composant fournisse une interface permettant d'accéder et de modifier ses propriétés. Par exemple, pour un composant de compression d'image, il peut proposer plusieurs formats de compression comme le *jpeg*, *gif* ou *png* correspondant chacun à un niveau de qualité de service par rapport au temps de compression et au rendu de l'image mais aussi par rapport à la consommation des ressources du périphérique. Une interface peut proposer une méthode permettant de choisir la qualité.

Lorsque la configuration de composant n'est pas possible, l'ajustement peut être effectué en remplaçant un ou plusieurs composants de l'assemblage par des composants offrant des fonctionnalités identiques mais proposant des QdS différentes.

4.3.3 Déploiement de service

Lorsque la migration du service n'est pas possible, il faut choisir un nouvel assemblage pour ce service afin de garantir l'objectif de gestion de qualité de service : la *continuité de service*. Ce choix d'un nouvel assemblage équivaut à déployer un nouveau service comme

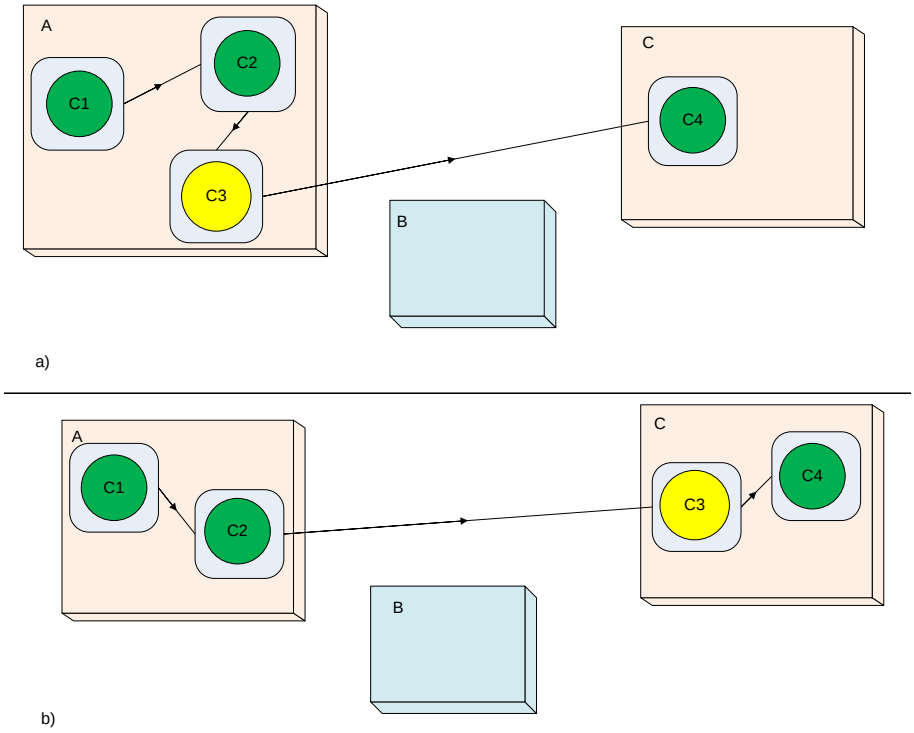

FIGURE 4.6 – Migration du service S sur les périphériques support de S

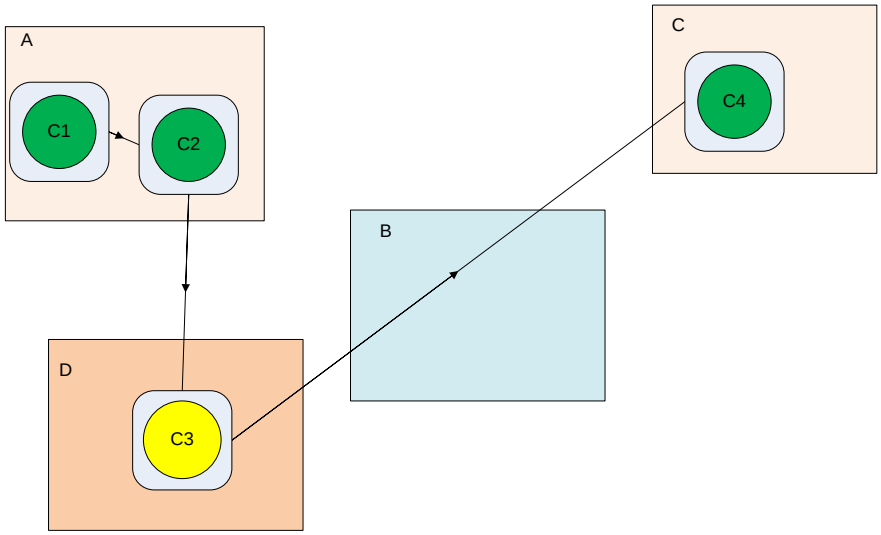

FIGURE 4.7 – Migration du service S en utilisant des périphériques voisins

par exemple lorsque les conditions d'utilisation le demandent ou lorsque l'utilisateur interagit avec le service.

En résumé, trois catégories d'évènements peuvent conduire à une reconfiguration :
- *Un évènement du contexte utilisateur* noté E_f pour évènement de fonctionnalité. L'utilisateur demande à utiliser une fonctionnalité non encore déployée.
- *Un évènement du contexte d'utilisation.* Des conditions d'utilisation particulières ont été détectées par la plate-forme qui impliquent des changements au niveau d'un ou de plusieurs services. Par exemple, un niveau sonore élevé indique qu'une conférence a démarré dans la salle où se trouve l'utilisateur. Afin qu'il n'occasionne aucune gène, le service *Guidage* et le service *Description* sont modifiés ou plutôt un nouvel assemblage est proposé pour éviter de diffuser du son à l'utilisateur. Nous notons ce type d'évènement E_c pour évènement de contexte. Parmi les situations décrites dans le contexte d'utilisation, il existe des situations d'urgence (section B) pour lesquelles une reconfiguration est impérative comme le cas d'un incendie. Nous notons de tels évènements E_u pour évènement d'urgence.
- *Un évènement du contexte d'exécution* entraine dans un premier temps une tentative de migration de service. Si celle-ci reste sans solution, le déploiement d'un nouvel assemblage est effectué. Un évènement de contexte d'utilisation peut être un évènement dû à une modification de ressources (CPU, mémoire, énergie), E_r, ou un évènement dû à la mobilité d'un périphérique, E_m.

A. **Déploiement suite à une demande d'une nouvelle fonctionnalité ou d'un nouveau service**.

Lorsqu'un utilisateur demande à pouvoir utiliser une fonctionnalité, la plate-forme doit la déployer. Par exemple, sur le service de *Description*, un utilisateur souhaite obtenir la fonctionnalité de traduction. La plate-forme doit alors déployer une configuration de ce service proposant la traduction. Dans un premier temps, la plate-forme évalue la configuration actuelle en y ajoutant le composant de traduction et les connexions associées. Tout comme pour le cas de la migration, l'évaluation est d'abord réalisée en utilisant les périphériques supportant déjà le service, puis en utilisant des périphériques supplémentaires voisins.

B. **Déploiement suite à un évènement du contexte d'utilisation**.

La plate-forme dispose de composants spécialement conçus afin de surveiller le contexte (section 4.5.1) et en particulier le contexte d'utilisation. Lorsqu'un évènement du contexte d'utilisation parvient à la plate-forme, dans un premier temps, elle parcourt les différentes règles d'utilisation permettant d'identifier une certaine condition d'utilisation (figure 4.8). Une condition d'utilisation est généralement liée à un rôle. A cette condition et à ce rôle est associée une fonction qui permet de dire si ce rôle doit être évité ou s'il est accepté dans le choix de la configuration du service concerné. Lorsque la condition est identifiée, le rôle en cause est également identifié et toutes les notes des configurations qui contiennent ce rôle sont modifiées par la fonction associée.

Certains évènements comme les évènements d'urgence, modifient les valeurs des seuils de niveau de qualité de service. Dans les situations d'urgence, la qualité d'utilisation est fortement privilégiée aux dépends de la consommation d'énergie des périphériques. L'utilisateur a besoin d'un service de haut niveau d'utilité plutôt que d'un service qui dure.

Une fois les notes de *QdS Utilité* et les seuils modifiés, le choix se déroule de la même manière que pour un déploiement de nouveau service.

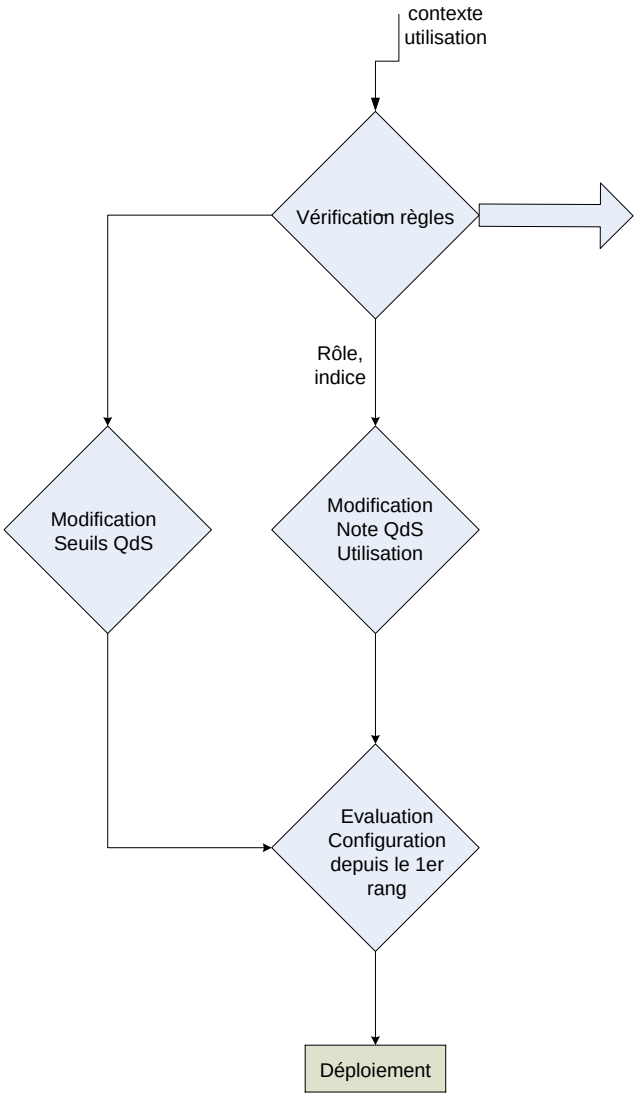

FIGURE 4.8 – Déploiement suite à la réception d'une information de contexte d'utilisation

C. **Déploiement suite à une migration sans solution**.

Lorsque que la migration d'une configuration ne trouve pas de solution, il faut alors choisir une nouvelle configuration du service concerné à déployer. Les configurations sont classées selon leur note de *QdS Utilité*. Puisque la configuration à migrer ne trouve pas de solution de migration, il faut descendre dans le classement des configurations et prendre la suivante, sous réserve que sa note de *QdS Utilité* soit toujours au dessus du seuil de tolérance.

4.3.4 Synthèse

Bien que l'utilisateur ne soit pas au centre de notre politique d'adaptation, nous prenons tout de même en compte l'influence de la reconfiguration sur le rendu final. Pour cela, nous nous basons sur la notion de fidélité. Le concept de fidélité a été introduit par Noble et al. [55] [56]dans Odyssey, une API pour l'adaptation centrée application. [55] définit la fidélité comme le degré de correspondance entre une copie d'une donnée présentée et la donnée de référence. Nous résumons cette définition de la fidélité comme reflétant l'incidence d'une adaptation par rapport à la configuration d'origine. Cette définition nous permet de définir un classement des moyens d'action en fonction de leur fidélité.

1. La migration est l'adaptation qui produit le moins d'impact sur le service rendu. Les composants sont inchangés, les fonctionnalités restent identiques. Nous l'appliquons en premier dans le cas des évènements qui ne demandent pas de modification de la structure *a priori* comme la mobilité d'un périphérique, E_m, ou la variation de ses ressources, E_r.

2. L'ajustement d'un service impacte le rendu final. La configuration est inchangée cependant les composants dans peuvent être ajustés ou remplacés dès lors qu'ils fournissent des fonctionnalités identiques. En pratique, ces ajustements correspondent dans notre méthode de conception (section 3.5) à des configurations indépendantes. Il est décrit autant de configurations qu'il y a d'ajustements. Finalement, l'ajustement d'un service est réalisé par la plate-forme par un déploiement de service.

3. Enfin, le déploiement est l'adaptation qui produit le plus d'impact sur le rendu final. La configuration est modifiée et par conséquent l'assemblage des composants aussi. Par exemple, pour le service de description, un nouveau déploiement peut proposer une description textuelle au lieu d'une description vidéo. Nous sélectionnons le déploiement en premier moyen de reconfiguration lorsque les évènements reçus impliquent des changements de structure comme la demande d'une nouvelle fonctionnalité, E_f ou l'apparition d'une situation de contexte, E_c/E_{cv} et E_u.

Le tableau 4 récapitule les évènements qui peuvent être reçus par la plate-forme et le moyen d'action associé.

Notation	Evènement	Moyen d'action
E_f	Evènement fonctionnalité	Déploiement nouveau service
E_c	Evènement de contexte	Envoi E_{cv}
E_{cv}	Evènement de contexte vérifié	MAJ note QdS Utilité et Déploiement nouveau service
E_u	Evènement d'urgence	MAJ seuil QdS et Déploiement nouveau service
E_r	Evènement de ressources	Migration
E_m	Evènement de mobilité	Migration

Table 4 – Tableau récapitulatif des évènements traités par la plate-forme

4.4 Interactions plate-forme / application

Kalimucho, telle que nous l'avons définie, est une plate-forme de supervision qui, par sa distribution sur tous les périphériques de l'application, offre une vision globale de cette dernière. A partir de cette vision, elle doit évaluer la qualité de service et restructurer l'application. Pour cela elle a besoin de suivre le fonctionnement des composants et la circulation des flux entre les composants. Afin de recueillir les informations sur ces deux entités, nous avons choisi d'utiliser des conteneurs qui, de plus, sont une solution à la gestion de l'hétérogénéité matérielle et logicielle des périphériques.

4.4.1 OSAGAIA : conteneur de composants métier

Le modèle OSAGAIA s'intéressait initialement aux applications multimédias réparties sur l'Internet et aux problèmes de synchronisation inter-flux. Dans ce modèle l'application est constituée de composants interconnectés par des flux d'information. Une plate-forme d'exécution supervise leur fonctionnement. Cette plate-forme reçoit de chaque constituant 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.

Bien que nous ne nous intéressions pas au problème de la synchronisation inter-flux, le modèle OSAGAIA fournit un cadre de travail intéressant pour le développement de composants supervisables. Il propose de séparer la logique métier représentée par un composant métier, du reste de l'application, gérée par un conteneur. Le **Composant métier** (CM) implémente un traitement particulier et un seul, appelé fonctionnalité. Il est guidé par les données : il ne peut s'exécuter que s'il existe des données dans l'un des ports d'entrée du conteneur qui l'encapsule. Le **conteneur** encapsule un et un seul composant métier (figure 4.9). Le conteneur implémente les propriétés non-fonctionnelles telles que la gestion du cycle de vie, la gestion de la qualité de service et la gestion des communications. Il possède trois unités : *l'Unité d'entrée (UE)*, *l'Unité de sortie (US)* et *l'Unité de contrôle (UC)*. L'*Unité d'Entrée* (UE) et l'*Unité de Sortie* (US) sont connectées respectivement aux ports d'entrée et aux ports de sortie. Elles permettent au CM de lire et écrire des données dans les ports d'entrée et de sortie. Lorsqu'un connecteur d'entrée écrit des données dans son port de sortie, il génère un événement qui prévient le conteneur que des données sont présentes dans un de ses ports d'entrée. Le composant métier peut alors lire les données via l'UE, effectuer son traitement et écrire le résultat dans l'US. L'écriture dans le port de sortie du conteneur génère un événement qui prévient le connecteur qu'il a des données à transporter. L'*Unité de contrôle* (UC) permet à la plate-forme de superviser le conteneur [10].

4.4.2 KORRONTEA : conteneur de connecteur

KORRONTEA fournit un conteneur pour les connecteurs [9]. La fonctionnalité principale d'un connecteur est de relier deux composants et de faire circuler l'information entre eux. Afin de pouvoir agir sur la circulation de l'information et influencer la QdS de l'application, [9] propose de considérer un connecteur comme un composant métier dont

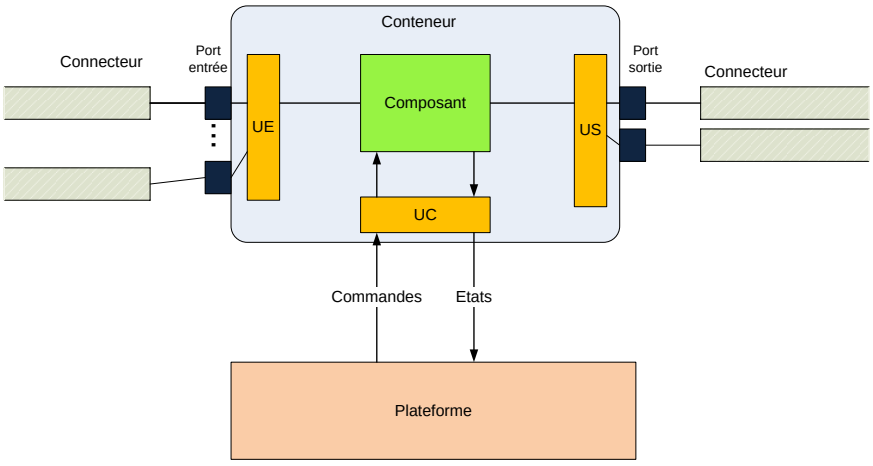

Figure 4.9 – Structure d'un conteneur de composant

le traitement implémente une politique de communication. De la même manière qu'OSA-GAIA, un connecteur est une entité de première classe. Il est encapsulé dans un conteneur doté d'une UE, d'une US et d'une UC, de façon identique au conteneur d'un composant métier (figure 4.9). Cependant, contrairement au conteneur de composant métier, le conteneur de connecteur ne peut connecter qu'une seule entrée et qu'une seule sortie. Enfin l'UC permet à la plate-forme de superviser le connecteur.

L'UE et l'US ont la particularité d'être reliées à des buffers permettant de temporiser l'arrivée des données et leur envoi (figure 4.10). L'utilisation de buffers permet d'éviter les pertes de données lors des reconfigurations. Les données sont stockées dans les buffers jusqu'à ce que la connexion soit rétablie. De plus, ils permettent d'identifier des anomalies comme la saturation ou la famine du buffer, qui peuvent engendrer des reconfigurations. La saturation d'un buffer de sortie peut indiquer que le composant suivant ne traite pas les données assez rapidement ou que la liaison réseau est défaillante. Une famine du buffer de sortie peut indiquer un dysfonctionnement du composant.

Osagaia et Korrontea sont des modèles de conteneurs fournissant des outils de supervision des composants et des connecteurs. De plus, ces deux modèles possèdent une similarité de structure qui permet à la plate-forme de considérer ces entités de façon identique et facilite les reconfigurations. nous détaillons cette plate-forme, Kalimucho, dans la section suivante.

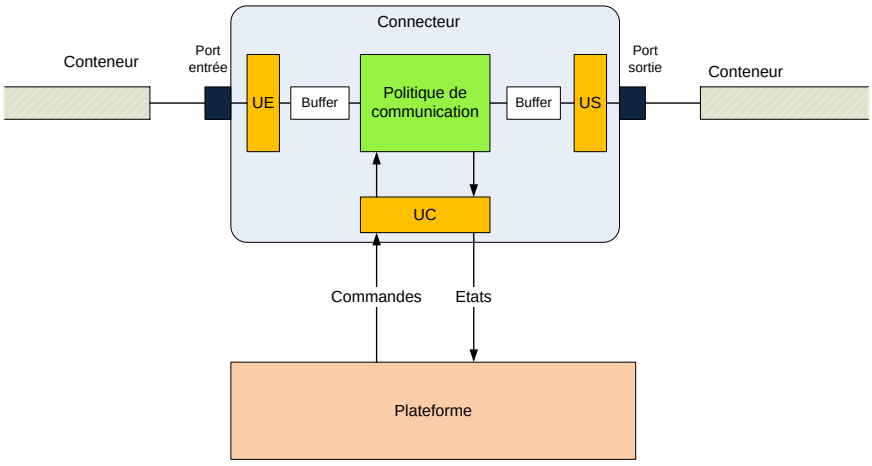

FIGURE 4.10 – Structure d'un conteneur de connecteur

4.5 Architecture de la plate-forme Kalimucho

Pour déployer les applications, la plate-forme a besoin de connaître l'état de l'application en cours d'exécution. Une application est composée d'un ou plusieurs services et chaque service est réalisé par un ou plusieurs assemblages de composants. Enfin les composants sont interconnectés par des connecteurs encapsulant les flux d'informations. L'état d'une application est un ensemble composé de l'état des composants, des périphériques, des connecteurs et de l'environnement. La plate-forme doit donc recueillir toutes ces données qui forment le contexte (figure 4.11) afin de les traiter et de mener l'action de reconfiguration qui convient. Pour chaque action de reconfiguration, la plate-forme évalue les configurations jusqu'à trouver une solution offrant une QdS suffisante. puis la déploie.

Dans cette description nous pouvons distinguer deux grandes fonctionnalités fournies par la plate-forme : la traitement des informations de contexte et l'évaluation des configurations. Ces deux fonctionnalités sont réalisées respectivement par les services *Superviseur* et *Générateur de Reconfiguration*. Ces deux services sont assistés par d'autres services leur permettant de disposer d'informations récentes sur les différentes configurations possibles et les composants disponibles tels que le *Registre de Configurations* et le *Registre de Composants*. Afin de pouvoir évaluer tous les déploiements possibles, la plate-forme a besoin de connaitre la topologie réseau de l'environnement. Ces informations seront recueillies et fournies par le service *Routage*. Enfin deux autres services permettent de mettre en oeuvre la solution générée par le service *Générateur de Reconfiguration*. Ce sont les services *Usine à Conteneur* et *Usine à Connecteur* qui permettent d'encapsuler les composants selon le modèle Osagaia et de créer des connecteurs selon le modèle Korrontea.

Les paragraphes qui suivent décrivent dans le détail le fonctionnement de chaque service et leurs interactions depuis la mesure du contexte jusqu'au déploiement d'une configuration.

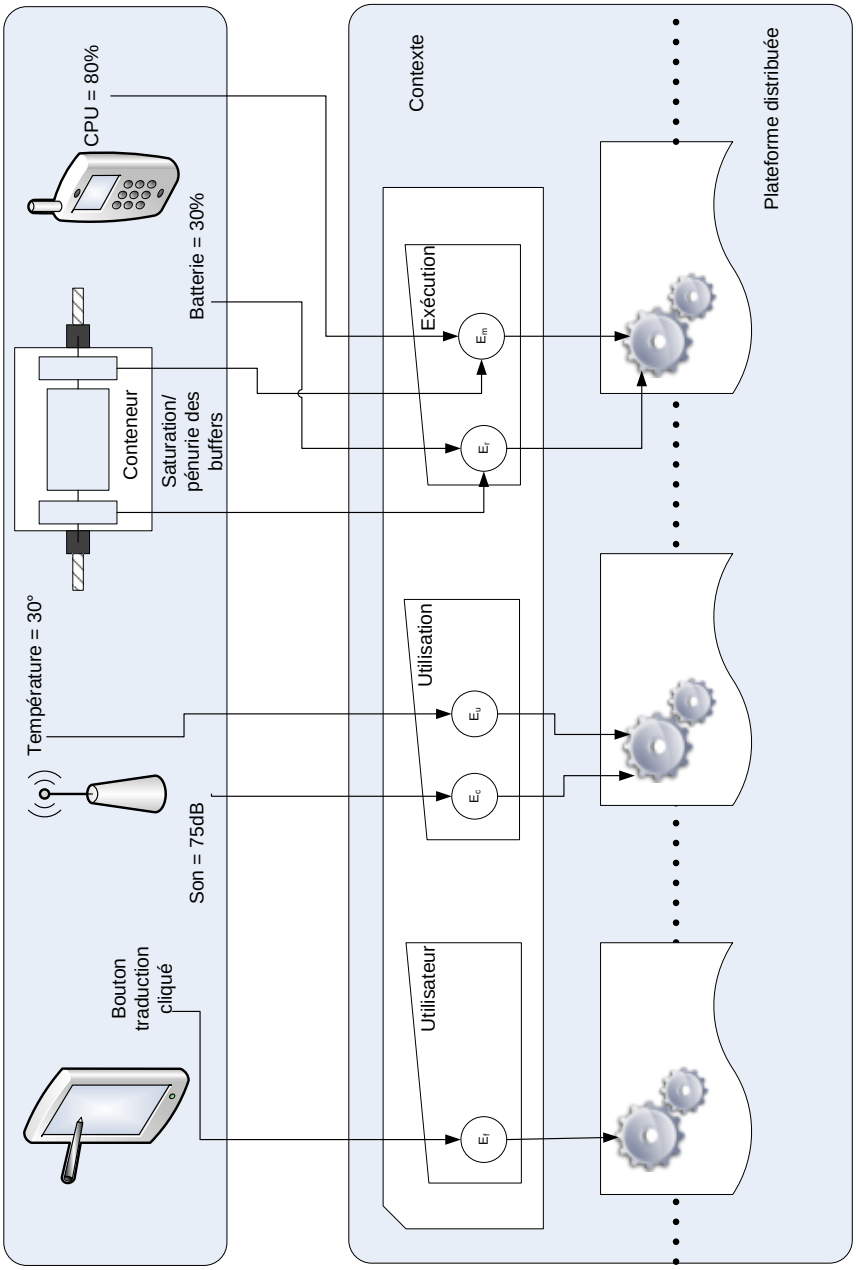

Figure 4.11 – schéma général de la plate-forme Kalimucho

4.5.1 Superviseur

Le service *Superviseur*, comme son nom l'indique, supervise le fonctionnement de l'application. Cette supervision se déroule en trois grandes étapes. Premièrement, le service

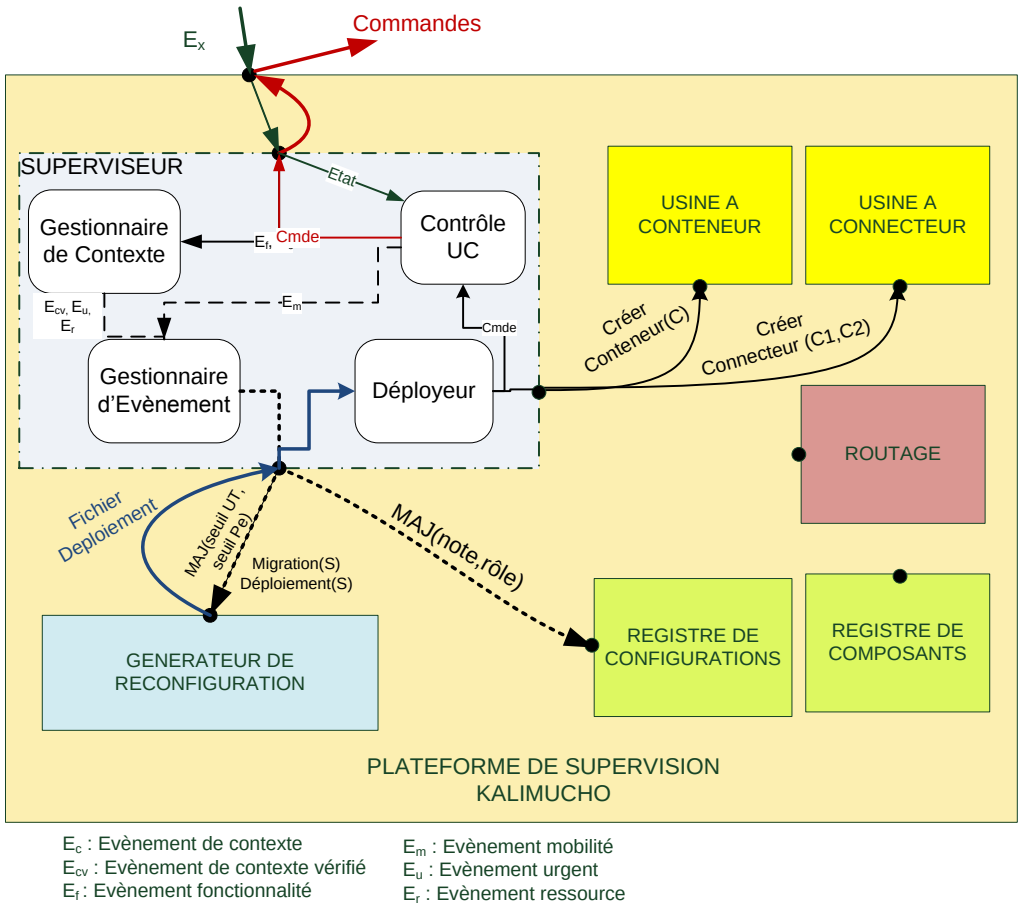

FIGURE 4.12 – Schéma général du service *Superviseur*

Superviseur reçoit des évènements en provenance des composants spécifiques de mesure du contexte et des conteneurs de composants métier et de connecteurs. Il peut également interroger les périphériques sur l'état de leurs ressources et sur la QdS offerte par les composants métiers. Deuxièmement il doit traiter ces mesures de contexte afin de fournir les informations nécessaires à l'évaluation. En effet, nous avons vu dans la section 4.3 qu'il existait plusieurs moyens d'action pour reconfigurer une application. Pour chaque moyen d'action, l'évaluation est différente et les informations doivent permettre d'identifier l'évaluation à mener. Troisièmement, lorsque l'évaluation est terminée et qu'elle a abouti à une configuration à déployer, le service *Superviseur* doit mettre en œuvre cette solution c'est-à-dire qu'il doit envoyer à l'application les commandes nécessaires à l'ajout, la suppression ou le déplacement de composants ou de connecteurs. Pour réaliser chacune de ces étapes, le service *Superviseur* est composé de quatre services, représentées dans la figure 4.12, respectivement le service *Gestionnaire de Contexte*, le service *Gestionnaire d'Evènement* et le service *Déployeur* et le service *Contrôle UC*.

4.5.1.1 *Gestionnaire de Contexte*

Le service *Superviseur* est le premier service qui interagit avec l'application notamment par l'intermédiaire du service *Gestionnaire de Contexte*. le service *Gestionnaire de Contexte* effectue plusieurs tâches convergeant toutes vers un seul but : recueillir et traiter les informations de contexte.

Tout d'abord le service *Gestionnaire de Contexte* permet de recueillir les trois types de contexte que nous avons définis à savoir le contexte d'utilisateur, le contexte d'utilisation et le contexte d'exécution. Concernant le contexte d'utilisateur, il s'agit ici de recueillir les souhaits de l'utilisateur par rapport à des fonctionnalités de l'application. Pour cela des composants spécifiques d'interaction homme-machine sont développés permettant d'associer chaque widget du service où l'utilisateur peut manifester un souhait à un certain type d'évènement. Les évènements de type contexte d'utilisateur sont notés E_f pour Évènement fonctionnalité (figure 4.13).

$$E_f(demande|arret, fonction) \tag{6}$$

L'utilisateur souhaite ou ne souhaite plus utiliser la fonctionnalité « fonction ».

Tout au long de l'exécution de l'application, le service *Gestionnaire de Contexte* est en attente d'un évènement E_f qu'elle transmettra au service *Gestionnaire d'Evènement* pour mener l'action qui convient.

Concernant le contexte d'utilisation, il s'agit de recueillir des informations sur l'environnement permettant de détecter des situations particulières nécessitant la reconfiguration de l'application.

Ces situations ont été préalablement décrites et détaillées lors de l'étape 2 (section 3.5.2) de la conception de l'application. Ce sont pour la plupart des informations abstraites telles que « une conférence a lieu »ou « il y a un incendie »ou encore « fermeture du musée ». Ces informations de contexte sont souvent le résultat de combinaisons de propriétés : ce sont les règles d'utilisation. Par exemple, l'information « une conférence a lieu »peut être le résultat de la règle suivante :

$$\text{niveau sonore } > 70\ dB \Rightarrow \text{début Conférence} \tag{7}$$

Ceci signifie qu'un niveau sonore supérieur ou égal à 70 décibels est synonyme qu'une conférence est en cours et donc déclenche l'évènement debutConference. Pour détecter la situation debutConference, la plate-forme a besoin de surveiller le niveau sonore de l'environnement.

D'une manière générale, afin de détecter ces situations de reconfiguration, la plate-forme doit surveiller toutes les propriétés entrant dans la composition de ces règles d'utilisation. Puisque seuls les périphériques dotés de capteurs peuvent mesurer les propriétés telles que le niveau sonore, la température, la présence de fumée, il ne parait pas judicieux de mettre en place une surveillance de ces propriétés directement depuis la plate-forme qui elle, est distribuée sur tous les périphériques. Pour mesurer ces propriétés nous mettons en place un type de composant spécial appelé *composant de contexte* sur les périphériques intéressés. Les composants de contexte sont chargés de mesurer les propriétés

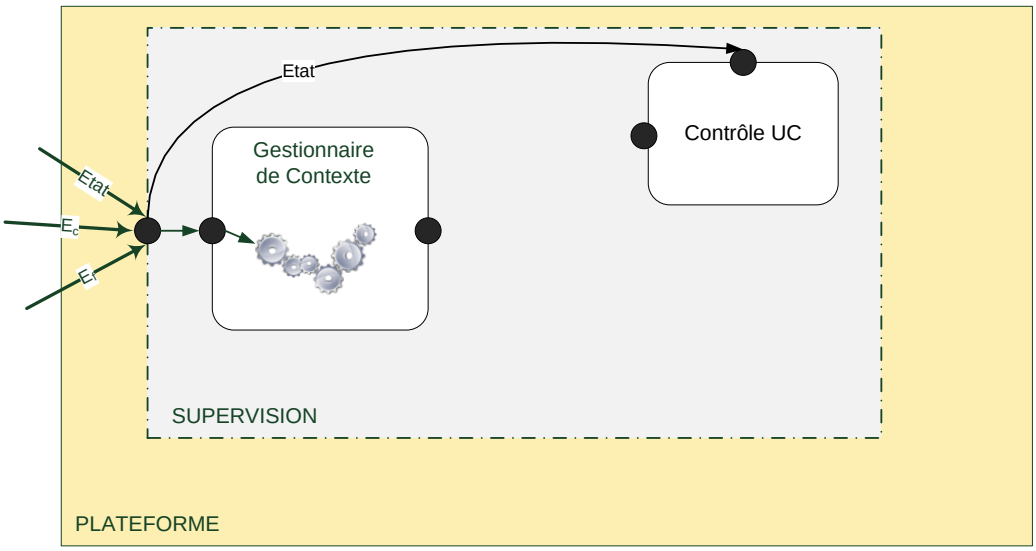

FIGURE 4.13 – Évènements de contexte capté par le service *Gestionnaire de Contexte*

précédemment citées et de comparer les mesures à un seuil fixé lors de la conception. Lorsqu'un seuil est dépassé, le *composant de contexte* envoie un évènement de contexte noté E_c au service *Gestionnaire de Contexte* (figure 4.13).

$$E_c(\text{niveau sonore} > 70 \text{ dB})$$

Le service *Gestionnaire de Contexte* renferme toutes les règles d'utilisation (formule 7) et attend un évènement de la part des composants de contexte. Lorsqu'elle reçoit un évènement E_c, elle vérifie les règles d'utilisation. Si une règle est vérifiée elle transmet un évènement de contexte vérifié noté E_{cv} au service *Gestionnaire d'Evènement*.

Parmi les règles d'utilisation, il y a des situations qui nécessitent des reconfigurations ne respectant pas idéalement le compromis entre une bonne *QdS Utilité* et une bonne *QdS Pérennité* c'est-à-dire une longue durée de vie. Par exemple, dans une situation d'incendie, il est impératif de fournir le service guidage indiquant la sortie de secours à l'utilisateur, sous sa meilleure forme quels que soient les niveaux des ressources du périphérique. On privilégie alors la *QdS Utilité* plutôt que la *QdS Pérennité*. Afin de guider le service *Gestionnaire d'Evènement* dans le choix de l'action à mener, les évènements déclenchés par les situations d'urgence sont notés E_u (figure 4.14).

La situation d'incendie peut être détectée comme suit :

$$\text{Si temperature } > 50^{\circ} \; ET \; \exists \text{fumée} \Rightarrow E_u(Incendie)$$

Concernant le contexte d'exécution, il s'agit de mesurer l'état des ressources des périphériques c'est-à-dire le niveau de charge CPU, le niveau de mémoire disponible et le niveau d'énergie disponible de chaque périphérique. Contrairement au contexte d'utilisation, le

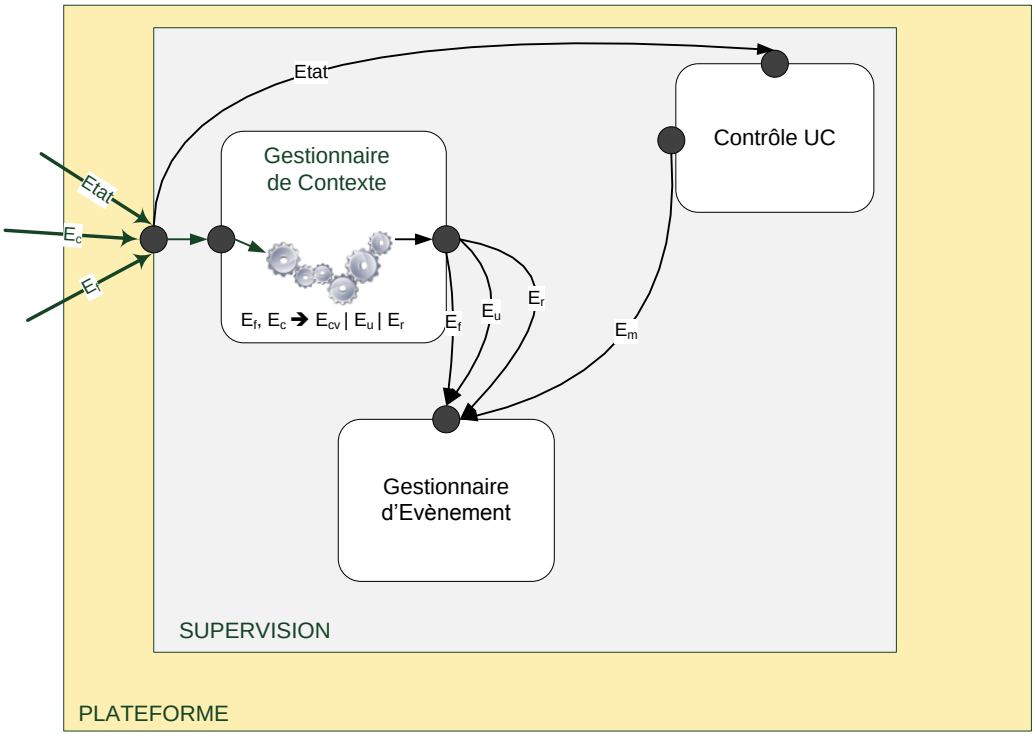

Figure 4.14 – Le service *Gestionnaire de Contexte* effectue un pré-traitement des évènements de contexte

contexte d'exécution est mesuré sur tous les périphériques quels qu'ils soient.

Puisque la plate-forme est présente sur tous les périphériques, les mesures de contexte d'exécution peuvent être faites directement depuis la plate-forme. Le service *Gestionnaire de Contexte* recueille les niveaux des trois ressources, CPU, mémoire et énergie. Une fois les informations recueillies, elles sont traitées afin de détecter des situations de reconfigurations. En effet, afin de garantir la *continuité de service*, des surveillances particulières peuvent être mises en place. Par exemple, lorsque le niveau d'énergie disponible d'un périphérique atteint le seuil critique de 10%, il est nécessaire de déplacer tous les composants qu'il supporte vers d'autres périphériques. Dans ce cas, le service *Gestionnaire de Contexte* renferme tous ces seuils et compare toutes les mesures aux seuils. Lorsqu'un seuil est franchi, elle envoie un évènement de ressource noté E_r (figure 4.14) au service *Gestionnaire d'Evènement* qui décidera de l'action à mener.

4.5.1.2 *Gestionnaire d'Evènement*

Une fois que le service *Gestionnaire de Contexte* a recueilli et pré-traité les informations, il faut décider de l'action de reconfiguration face à l'apparition de ces évènements. Cette tâche est réalisée par le service *Gestionnaire d'Evènement* représentée sur la figure 4.15.

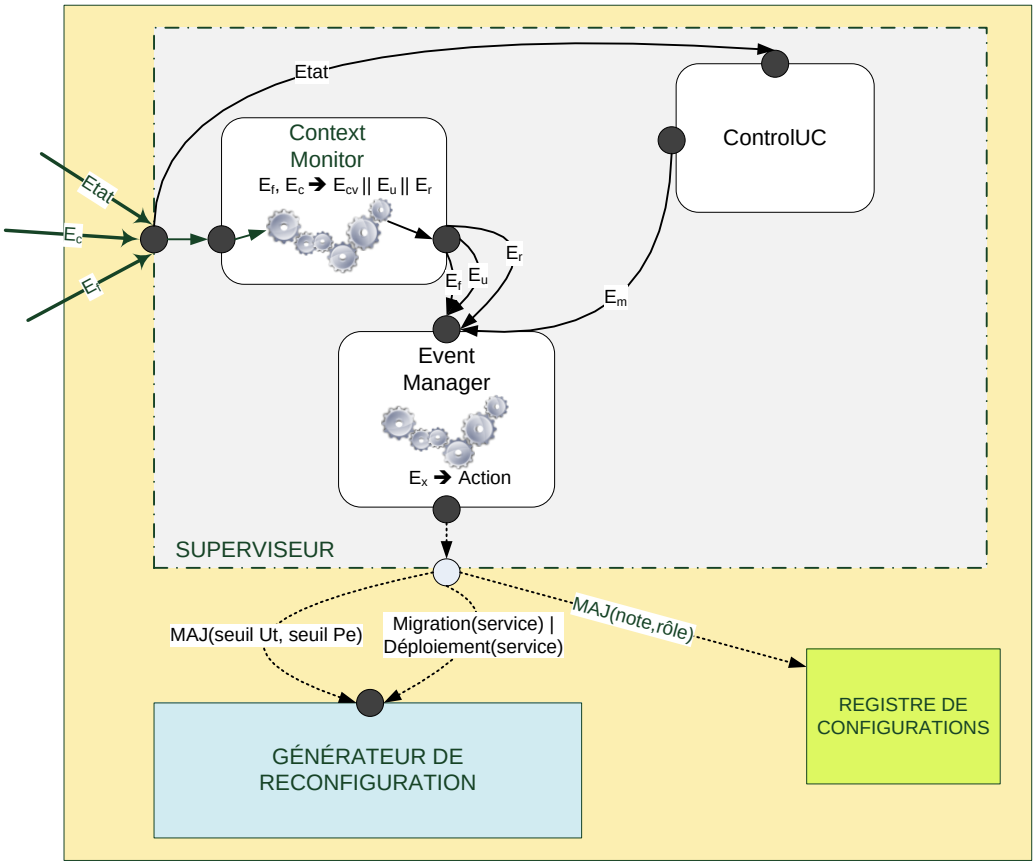

FIGURE 4.15 – Schéma des interactions du service *Gestionnaire d'Evènement*

Dans le paragraphe précédent, nous avons déterminé trois types d'évènements reçus par le service *Gestionnaire d'Evènement* : E_{cv}, E_u et E_r. Le but du service *Gestionnaire d'Evènement* est de mener l'action de reconfiguration adéquate selon un évènement : migration ou déploiement (section 4.3). Pour cela, elle possède une liste de règles d'association entre un évènement et une ou plusieurs actions.

Concernant les **évènements de contexte vérifié**, E_{cv}, ces évènements correspondent à des règles d'utilisation qui spécifient un fonctionnement particulier de l'application. Par exemple, lorsqu'une conférence a lieu, il est préconisé d'éviter de diffuser du son aux utilisateurs. Telle que décrite dans l'étape 5 de la conception (section 3.5.5, une règle définit que lorsque l'évènement « debutConference »est déclenché, il faut baisser les notes de *QdS Utilité* de toutes les configurations proposant le rôle « son »de 0,3. Il est également nécessaire de vérifier toutes les configurations déployées et de les remplacer si l'une d'entre elles fournit le rôle « son ». En résumé, lorsque le service *Gestionnaire d'Evènement* reçoit l'E_{cv} « debutConference », elle doit :

- modifier les notes de *QdS Utilité* des configurations intégrant le rôle « son »de $-0,3$
- déployer une nouvelle configuration des services visés

Les configurations étant stockées dans le Registre de configuration, c'est à lui que le service *Gestionnaire d'Evènement* doit envoyer la demande de modification. Quant au déploiement d'une nouvelle configuration, c'est au service *Générateur de Reconfiguration* de trouver la nouvelle configuration.

La formule 8 synthétise les actions à mener par le service *Gestionnaire d'Evènement* lorsqu'elle reçoit un évènement E_{cv} :

$$E_{cv} \Rightarrow MAJ(note, \text{rôle})\ ET\ \text{Déploiement}(service) \tag{8}$$

Les **évènements d'urgence** sont des cas particuliers d'évènements de contexte vérifié. Ce sont des évènements qui impliquent impérativement une reconfiguration et imposent parfois d'outrepasser les seuils de tolérance servant à garantir le meilleur compromis entre *QdS Utilité* et *QdS Pérennité*. Cependant ces seuils sont les bornes nécessaires au bon fonctionnement de l'heuristique d'évaluation des configurations. Il n'est par conséquent pas possible de passer outre ces seuils lors de l'évaluation. La seule solution est de modifier la valeur de ces seuils afin d'agrandir le champ de possibilités pour le choix d'une configuration. Ainsi chaque évènement d'urgence Eu est associé à une nouvelle valeur du seuil *QdS Utilité* et du seuil *QdS Pérennité* comme dans la formule 9. De plus un évènement E_u peut être associé au déploiement d'un service particulier comme dans l'exemple de l'incendie qui impose le déploiement du service guidage.

Lorsque le service *Gestionnaire d'Evènement* reçoit un évènement E_u, elle procède alors à deux actions (formule 9) :

- mettre à jour les valeurs des seuils de QdS
- déployer un nouveau service

$$E_u \Rightarrow MAJ(\text{seuil Ut, seuil Pe})\ ET\ \text{Deploiement(service)} \tag{9}$$

Les **évènements de ressources**, E_r, impliquent, dans la majorité des cas, de déplacer tout ou une partie des composants supportés par le périphérique en difficulté vers d'autres périphériques. Cependant, au cours de l'exécution de l'application, il peut arriver que la majorité des périphériques aient atteint un niveau d'énergie disponible tel que le seuil de *QdS Pérennité* fixé jusqu'alors ne permet plus de trouver une configuration « déployable ». Pour pouvoir continuer de trouver des configurations que la plate-forme peut déployer, il faut modifier les seuils de QdS. Comme dans le cas des Eu, le service *Gestionnaire d'Evènement* peut envoyer une demande de mise à jour des seuils de QdS au service *EvaluerReconfiguration* du service *Générateur de Reconfiguration*.

En résumé, lorsque le service *Gestionnaire d'Evènement* reçoit un évènement E_r, elle peut effectuer deux opérations (formule 10) :

- demander une mise à jour les valeurs des seuils de QdS
- demander une migration du service intéressé

$$E_r \Rightarrow MAJ(\text{seuil Ut, seuil Pe})\ ET\ Migration(service) \tag{10}$$

Enfin, le service *Gestionnaire d'Evènement* peut recevoir un quatrième type d'évènement : un évènement lié à la mobilité d'un périphérique ou au fonctionnement d'un composant,

E_m. En effet nous développons nos applications selon le modèle de composant Osagaia et le modèle de flux Korrontea. Ces deux modèles proposent une architecture similaire pour le développement de composants et de connecteurs. Ils sont toux deux encapsulés dans un conteneur constitué de trois unités : l'UE, l'US et l'UC. L'UC est une unité d'échange servant d'intermédiaire entre le composant ou le connecteur et la plate-forme. Elle est le seul point de communication avec la plate-forme.

L'UC surveille le fonctionnement du composant ou du connecteur via ses UE et US. Lorsqu'elle détecte une saturation ou une pénurie de données dans les zones tampons de l'UE ou l'US, c'est que l'une des trois situations suivantes est apparue :

- Le composant précédent produit trop de données ou ne produit pas assez de données
- Le composant suivant ne consomme pas assez de données ou en consomme trop
- Le périphérique supportant le composant précédent ou le composant suivant s'est déplacé

Dans les deux premiers cas, la configuration ne parait pas adaptée, les composants ne fonctionnent pas en harmonie et la qualité du service n'est pas la meilleure, il faut déployer une nouvelle configuration. Dans le dernier cas, il faut essayer de trouver un nouveau support pour le ou les composants qui ont bougé. Il faut donc migrer la configuration.

Notons E_m les évènements envoyés par les UC via le service *Contrôle UC*. Lorsque le service *Gestionnaire d'Evènement* reçoit un évènement E_m, elle procède aux opérations suivantes :

- Vérifier la topologie de l'application
- Si aucun périphérique n'a bougé, alors elle demande un déploiement au service *EvaluerReconfiguration*
- Si un périphérique a bougé, alors elle demande une migration du service intéressé au service *EvaluerReconfiguration*

Une fois que les informations de contexte ont été traitées et que toutes les demandes de reconfiguration ont été transmises au service Générateur de Reconfiguration, il ne reste plus qu'à déployer la configuration générée par ce dernier. Cette dernière tâche est effectuée par le service *Déployeur*.

4.5.1.3 *Déployeur* et *Contrôle UC*

La dernière étape du service *Superviseur* est de déployer la solution proposée par le service *Générateur de Reconfiguration*. Ce dernier envoie à la plate-forme des périphériques concernés toutes les opérations à effectuer pour mettre en œuvre la configuration proposée. le service *Déployeur* envoie les commandes aux services concernés leurs commandes. Pour chaque composant à déployer, le service *Déployeur* envoie une commande de création de conteneur au service *Usine à Conteneur*. De la même façon, pour chaque connexion entre deux composants, une commande de création de connecteur est envoyé au service *Usine à Connecteur*. Enfin pour pouvoir supprimer, déplacer ou lancer de nouveaux composants ou de nouveaux conteneurs, la plate-forme a besoin d'envoyer des commandes directement aux composants ou connecteurs.

le service *Contrôle UC* est le seul contact avec les composants et les connecteurs de l'application. Les conteneurs et connecteurs sont dotés de trois unités, UE, US et UC. L'UC

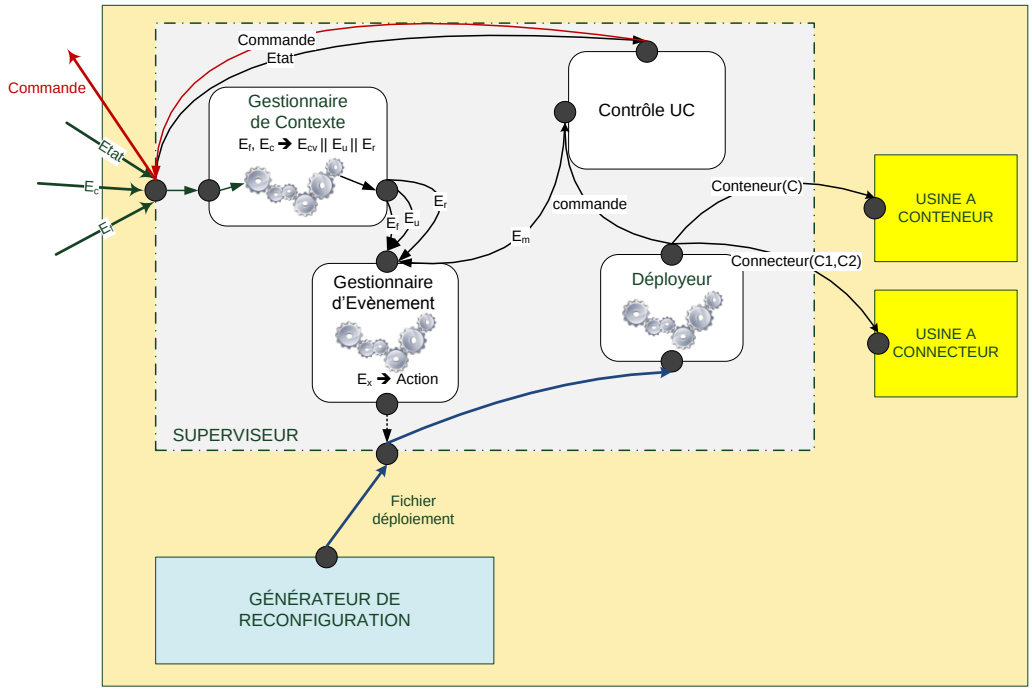

Figure 4.16 – Schéma des interactions des services *Déployeur* et *Contrôle UC*

peut contrôler le bon déroulement de lecture et d'écriture du composant via l'UE et l'US. Elle peut détecter les situations de saturation ou inversement des situations de pénurie dans les buffers de lecture et d'écriture. Ce type de situation signifie que le fonctionnement du composant précédent ou du composant suivant a changé ou que ces composants ont bougé. La QdS du service intéressé change et il est nécessaire de reconfigurer. Lorsque le service *Contrôle UC* détecte une de ces situations, elle envoie un évènement de mobilité noté E_m (figure 4.16) au service *Gestionnaire d'Evènement* qui se chargera de vérifier la topologie de l'application et décidera d'une action de reconfiguration.

le service *Contrôle UC* administre également les composants et les connecteurs de l'application. Ce service permet de contrôler le cycle de vie d'un composant ou d'un connecteur par l'intermédiaire de l'*unité de contrôle* (UC) du conteneur qui l'encapsule. Ainsi lors des déploiements, le service *Contrôle UC* peut envoyer des commandes au conteneur du composant pour ajuster sa QdS lorsqu'il le permet.

Une instance du service *Superviseur* est embarquée sur tous les périphériques. Lorsque le *Gestionnaire de Contexte* reçoit une information de contexte, il vérifie l'ensemble des règles d'utilisation. Si une règle est vérifiée, il envoie un évènement au *Gestionnaire d'Evènement*. Le *Gestionnaire d'Evènement* associe l'évènement de contexte à des informations de mesure de la QdS et transmet le tout au service *Générateur de Reconfiguration*. Lorsque que le service *Déployeur* reçoit une carte de déploiement de la part du service *Générateur de Reconfiguration*, c'est à dire la liste des composants, leur ordonnancement et leur distribution, alors il procède au déploiement. Pour cela il envoie les commandes

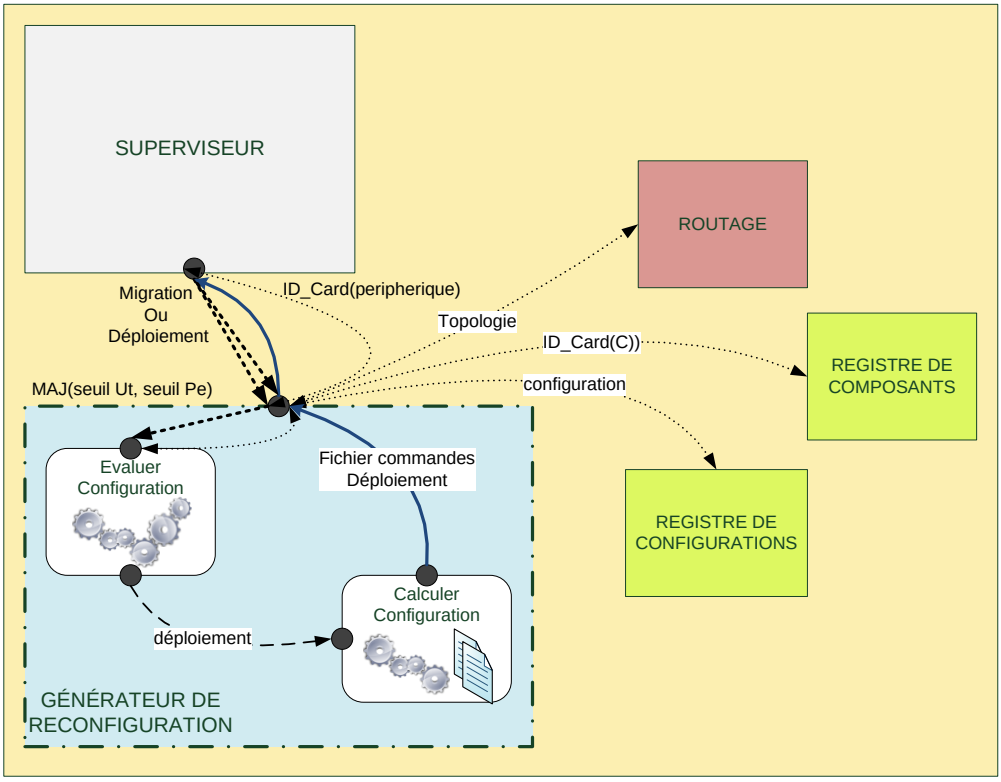

FIGURE 4.17 – Schéma des interactions du service *Générateur de Reconfiguration*

nécessaires au service *Usine à Conteneur* et au service *Usine à Connecteur* ainsi qu'au service *Contrôle UC* qui se chargera d'envoyer des commandes aux composants et aux connecteurs de l'application dans le cas où il faut supprimer ou arrêter des composants et des connecteurs.

4.5.2 Générateur de reconfiguration

Le service *Générateur de Reconfiguration* a pour objectif d'évaluer les déploiements des configurations d'un service donné et de générer le déploiement qui offre la meilleure qualité de service. Ces deux fonctionnalités sont réalisées respectivement par le service *EvaluerConfiguration* et le service *CalculerConfiguration*.

Le service *Générateur de Reconfiguration* reçoit toutes les demandes de reconfiguration de la part du service *Superviseur*. Plus précisemment c'est le service *EvaluerConfiguration* qui réceptionne ces demandes (figure 4.17).

4.5.2.1 *EvaluerConfiguration*

Le service *EvaluerConfiguration* est le service qui met en œuvre l'heuristique d'évaluation des configurations d'un service, détaillée dans la section 4.7, en vue d'une migration ou

d'un déploiement. Selon le moyen d'action demandé, le processus se déroule différemment.

Afin de restreindre le nombre de possibilités lors de l'évaluation et dans le but d'obtenir un résultat le plus rapidement possible, nous avons choisi d'utiliser une heuristique dont les bornes sont fixées par les seuils de QdS [42]. L'évaluation se déroule telle qu'elle a été décrite dans le modèle de QdS (section 3.3). Lors d'une évaluation, l'heuristique n'évalue pas toutes les configurations. Elle n'évalue que les configurations dont la note de *QdS Utilité* se situe entre la note maximale, 1, et le seuil de *QdS Utilité* (figure 3.5). Un seuil de *QdS Pérennité* est également fixé afin de garantir la meilleure durée de vie à l'application. Ainsi pour chaque configuration située dans l'espace défini pour la *QdS Utilité*, les déploiements dont la note de *QdS Pérennité* ne se trouve pas dans la surface formée par les bornes maximales et les seuils, seront écartés.

Ces seuils, initialement fixés à la conception, peuvent évoluer au cours du temps, suite à des changements de contexte comme des niveaux d'énergie trop bas ou des évènements d'urgence. Lorsque des demandes de reconfiguration arrivent, le service *EvaluerConfiguration* agit dans l'ordre suivant :

- mise à jour des seuils de QdS
- évaluation d'une configuration pour une migration ou
- évaluation des configurations pour un déploiement d'un nouveau service.

Evaluation d'une migration La migration d'un service consiste à redéployer d'une façon différente la même configuration. Pour le service *EvaluerConfiguration* il s'agit d'évaluer les différents déploiements possibles pour la configuration actuelle du service. Il faut donc dans un premier temps récupérer les informations sur la configuration actuellement déployée. A chaque déploiement, le service *Déployeur* du service *Superviseur* sauvegarde le dernier déploiement de chaque service en cours d'exécution. Puis, pour chaque composant participant à la configuration, le service *EvaluerConfiguration* a besoin des informations de QdS, c'est-à-dire les informations liées à la consommation des ressources, permettant de calculer sa note de *QdS Pérennité*. Pour cela elle s'adresse au Registre des Composants qui détient toutes les cartes d'identité des composants. Ensuite, elle récupère la carte topologique du réseau formé par les périphériques supportant l'application. Cette topologie réseau fournie par le service *Routage* permet de connaitre la situation réseau des périphériques supportant actuellement la configuration ainsi que celle des périphériques voisins susceptibles d'accueillir la configuration. Enfin, une fois que le service *EvaluerConfiguration* connait l'ensemble des périphériques disponibles, il ne lui reste plus qu'à demander au service *Gestionnaire de Contexte* l'état des ressources des périphériques afin de lancer l'évaluation.

A la fin de l'évaluation, si un déploiement a été trouvé, il est envoyé au service *CalculerConfiguration*. Sinon, une autre évaluation est lancée, mais cette fois-ci, non plus pour une migration, mais pour le déploiement d'une nouvelle configuration.

L'évaluation du déploiement d'une nouvelle cofiguration est présentée dans le paragraphe suivant.

Evaluation d'un déploiement Lorsqu'il est question de déployer une nouvelle configuration d'un service, bien entendu le résultat doit fournir le déploiement offrant une qualité

de service suffisante. Étant donné que la QdS peut autant baisser qu'augmenter (cas du début et fin d'une conférence par exemple), nous ne pouvons pas nous contenter d'évaluer les configurations en remontant ou en baissant dans le classement, depuis la position de la configuration actuelle. Le classement étant mis à jour au fur est à mesure des changements de contexte, la configuration actuelle n'a peut-être plus la même position qu'au moment de son déploiement. Afin de trouver la meilleure configuration, il est naturel de démarrer l'évaluation depuis le premier rang du classement.

Pour calculer la note d'un déploiement nous avons besoin de plusieurs informations. Premièrement, les taux de consommation des ressources des différents composants participant à la configuration et leur propriété d'affectation (cas des composants liés à un matériel). Pour cela, le service *EvaluerConfiguration* envoie une demande de carte d'identité des composants au Registre des Composants. Deuxièmement, la topologie réseau. Elle est fournie par le service *Routage*. Troisièmement, le service *EvaluerConfiguration* récupère via le service *Gestionnaire de Contexte*, les cartes d'identité des périphériques afin de connaitre le niveau de leurs ressources et leurs propriétés matérielles.

L'évaluation se déroule ensuite selon la figure B. Afin de pouvoir fournir une QdS suffisante, l'heuristique d'évaluation (section 4.7.1.2) classe les périphériques selon leur position dans la topologie du réseau, leur type, leur niveau d'énergie, leur charge CPU et leur niveau de mémoire, le premier étant le périphérique ayant les ressources les plus disponibles et le dernier, les ressources les moins disponibles.

Lorsqu'un déploiement a été trouvé, il est transmis au service *CalculerConfiguration* afin d'établir les commandes à effectuer par chaque plate-forme des périphériques concernés. *CalculerConfiguration* est détaillée dans la section suivante.

4.5.2.2 *CalculerConfiguration*

Lorsque le service *CalculerConfiguration* reçoit un déploiement de le part du service *EvaluerConfiguration*, il doit générer toutes les commandes nécessaires à la mise en place du déploiement. En se rapportant au déploiement actuel, le service repère les composants qui ne bougent pas de place. Puis pour chaque nouveau composant ou chaque déplacement, elle génère une commande de création d'un conteneur. De la même façon pour les connexions entre composants, elle génère les commandes de création de connecteur. A la sortie, le service *CalculerConfiguration* indique au service *Déployeur*, la liste des composants à ajouter, à supprimer, ou à déplacer. Cette liste fournit également l'ordonnancement des composants et les périphériques sur lesquels ils doivent être déployés.

Du fait de sa complexité et du nombre élevé de calculs et de communications demandés par le service *Générateur de Reconfiguration*, ce dernier causerait un épuisement énergétique prématuré s'il était déployé sur un périphérique contraint. Il n'est alors déployé que sur des périphériques non contraints en ressources, les périphériques fixes. Cependant cette distribution peut engendrer des conflits entre les différents services Générateur de reconfiguration. Nous devons alors nous assurer que pour un service donné, un seul service *Générateur de Reconfiguration* peut proposer des adaptations. Pour cela, nous proposons d'utiliser un mécanisme de désignation du service *Générateur de Reconfiguration* en charge du service lors du déploiement d'un service. Lors de la désignation du Générateur

de reconfiguration, les autres plate-formes participant au service sont notifiées par le service référent, leur indiquant que tous les évènements de reconfiguration doivent lui être transmis. Lorsqu'une reconfiguration doit être déployée, le Générateur de reconfiguration référent envoie aux autres plate-formes la liste des commandes ainsi que la liste des composants et des connecteurs qu'elles doivent déployer localement.

Pour pouvoir déployer la configuration choisie, les composants et les connecteurs doivent être encapsulés dans des conteneurs. Cette encapsulation est réalisé par les services *Usine à Conteneur* et *Usine à Connecteur*, détaillés dans les sections suivantes.

4.5.3 *Usine à Conteneur*

Le service *Usine à Conteneur* est un service permettant de construire des conteneurs conformes au modèle Osagaia. La particularité de ce service est qu'il crée un conteneur adapté au composant métier qu'il doit encapsuler ainsi qu'au périphérique le supportant (contraint ou non). Le service *Usine à Conteneur* reçoit des commandes de la part du service *Déployeur* du service *Superviseur* afin de créer des conteneurs. pour cela le service *Déployeur* fournit la carte d'identité du composant à encapsuler et le périphérique sur lequel il doit être déployé. Les composants sont disponibles dans un entrepôt distribué sur le réseau. Dans le cas des périphériques contraints, comme le défini le standard CLDC, il n'est pas possible de charger dynamiquement des classes. Ces périphériques embarqueront alors leur entrepôt dans lequel seront disposés les composants qu'ils seront susceptible de supporter. Le service *Usine à Conteneur* doit donc se connecter à l'entrepôt pour charger le CM demandé par le service *Superviseur*. Ensuite il adapte le conteneur et le déploie.

4.5.4 *Usine à Connecteur*

Le service *Usine à Connecteur* est un service permettant de construire des connecteurs conformes au modèle Korrontea. Un Connecteur encapsule un composant de communication implantant une politique de communication. L'*Usine à Connecteur* reçoit des commandes de la part du service *Déployeur* du service *Superviseur*. Cette dernière lui transmet les noms des composants à relier ainsi que leur localisation. Si le connecteur est distribué, chaque service *Usine à Connecteur* construit une extrémité du connecteur et la déploie.

4.5.5 Routage

Le service *Routage* est un service permettant de construire la topologie de l'application à la demande du service *Générateur de Reconfiguration*. L'application est supportée par des périphériques hétérogènes qui utilisent des réseaux différents pour communiquer : Wifi, Bluetooth, ZigBee, etc. Un périphérique qui utilise le réseau Wifi ne peut percevoir autour de lui que les autres périphériques qui utilisent également le réseau Wifi. Il en est de même pour tous les autres réseaux. Afin d'obtenir la topologie de l'application, le service *Routage* doit alors dresser une table de routage pour chaque réseau. Pour cela, il cherche toutes les routes possibles entre deux périphériques données, à la demande du service *Générateur*

de Reconfiguration. l'ensemble de ces routes permettent ensuite de former un domaine d'évaluation [64] dans lequel seront évaluées les reconfigurations.

4.5.6 Registre de Composants et Registre de Configurations

Les registres de Composants et de Configuration sont deux entités permettant d'entreposer la liste des composants et des configurations disponibles . Il est utilisé par la plate-forme pour effectuer des restructurations. Le registre de composants contient toutes les cartes d'identité des composants disponibles par l'application. Il tient essentiellement un rôle d'information pour le service *Générateur de Reconfiguration*. Le registre des configurations quant à lui contient le classement des configurations de chaque service selon leur note de *QdS Utilité*. Ce registre est mis à jour chaque fois que le service *Gestionnaire d'Evènement* du service *Superviseur* identifie une situation de contexte associée à une modification de note. Il est utilisé par le service *Générateur de Reconfiguration* chaque fois qu'il effectue l'évaluation d'une configuration.

4.6 Principe de déploiement de Kalimucho

Puisque nous utilisons des périphériques contraints, nous devons adapter le déploiement de la plate-forme aux capacités des périphériques. Nous distinguons deux types de déploiement : la plate-forme classique (Périphérique fixe) et la plate-forme contrainte (Périphérique CLDC) illustrés dans la figure 4.18. La plate-forme classique est composée de tous les services. C'est le type de plate-forme qui est déployé sur des périphériques sans limitation de ressources tels les ordinateurs de bureau. La plate-forme contrainte ne dispose pas de la totalité des services notamment, elle ne dispose pas du service *Générateur de Reconfiguration*. Le service *Générateur de Reconfiguration* demande un grand nombre de calculs et de communications pour obtenir les informations nécessaires aux choix de reconfiguration. Le service *Superviseur* est également allégé afin de ne pas surcharger les périphériques. Il ne dispose pas du service *Gestionnaire d'Evènement* qui renferme toutes les règles d'adaptation.

Lorsque l'information de contexte est reçue par un service *Superviseur* d'une plate-forme contrainte, il doit tout d'abord relayer l'information jusqu'au service *Superviseur* d'une plate-forme voisine la plus proche qui a accès à tous les autres services. Les informations sont relayées au service *Gestionnaire d'Evènement* de la plate-forme classique. Ce dernier associe l'évènement de contexte à des informations de mesure de la QdS et transmet le tout au service *Générateur de Reconfiguration* référent. Le service *Générateur de Reconfiguration* référent envoie les commandes de déploiement au service *Générateur de Reconfiguration* de la plate-forme classique, qui les relaie à son tour au service *Déployeur* de la plate-forme contrainte. le service *Déployeur* peut alors procéder au déploiement. Pour cela il envoie les commandes nécessaires au service *Usine à Conteneur* et au service *Usine à Connecteur* qui se chargeront de créer et supprimer des composants et des connecteurs.

Le fait que la plate-forme soit totalement répartie permet d'effectuer toutes les opérations depuis n'importe quel périphérique puisque l'ensemble des plate-formes communiquent entre elles. Ainsi il est possible de déléguer des opérations sans aucun problème.

Le paragraphe suivant détaille la mise en œuvre des reconfigurations en fonction des évènements reçus par Kalimucho.

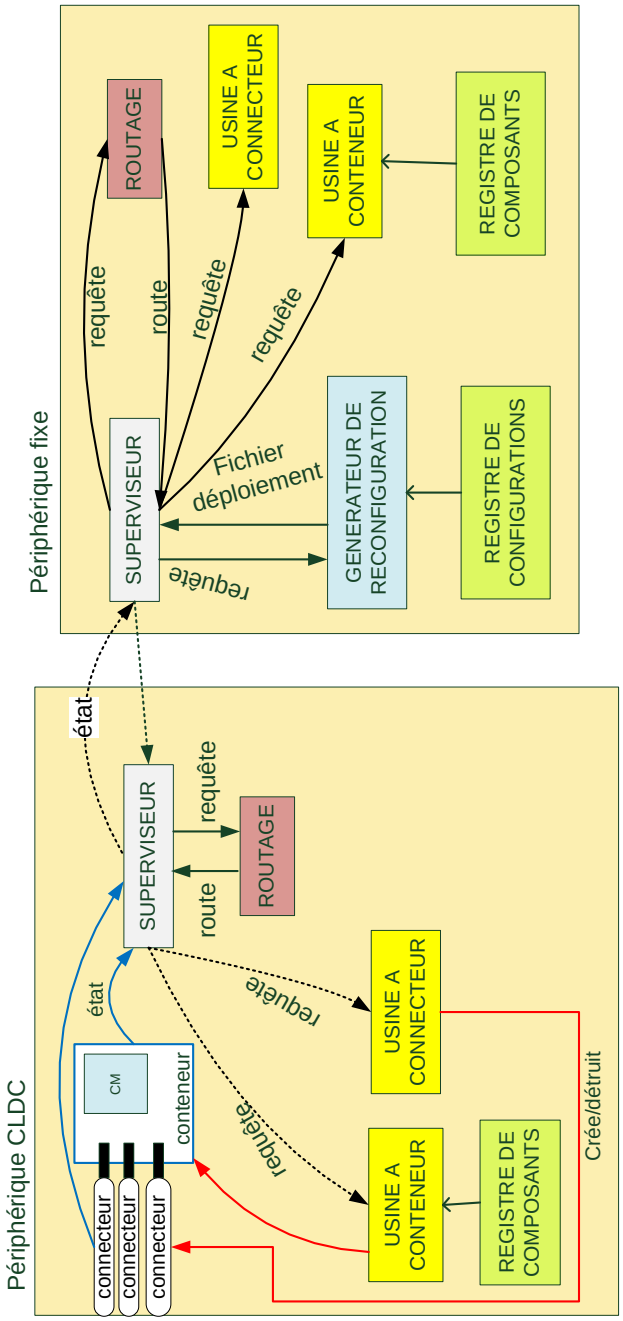

FIGURE 4.18 – schéma de déploiement de la plate-forme Kalimucho

4.7 Modèle d'exécution de Kalimucho

Il s'agit ici de définir un assemblage optimum de composants réalisant un service. Or ce problème est connu comme étant NP-complet dans le cas général. Nous devons donc définir des règles permettant de réduire le nombre d'assemblages étudié.

Un service est réalisé par un assemblage de composants. Afin de faciliter le choix d'un déploiement, il est nécessaire de prendre en compte les propriétés de l'assemblage.

- La disponibilité des composants. Les configurations indiquent les fonctionnalités qui doivent être fournies pour réaliser le service. Pour évaluer une configuration, nous devons préalablement nous assurer que l'ensemble des composants disponibles possèdent toutes les fonctionnalités requises.
- Les composants non déplaçables. Parmi les composants d'un assemblage existent des composants dont le fonctionnement est dirigé par la présence d'un matériel ou est lié à un utilisateur. C'est le cas pour un composant d'acquisition sonore qui a besoin d'un micro pour fonctionner ou pour un composant de lecture vidéo qui nécessite un écran qui doit être sur le périphérique de l'utilisateur qui demande cette fonctionnalité. Le déploiement de l'assemblage doit alors respecter la contrainte de ces composants non déplaçables.
- Les composants communiquent et s'échangent des données. Un composant produit, l'autre consomme, etc. Il existe des contraintes de précédences que le déploiement doit respecter sans quoi l'assemblage ne peut pas fonctionner. Par exemple, un composant de compression impose qu'un composant de décompression le suive et pas l'inverse.

Étant donné que nos applications évoluent dans un environnement mouvant dû au contexte et à la mobilité des périphériques, nous ne pouvons pas choisir la meilleure configuration par une approche itérative telle que [42]. L'approche proposée par [42] est une heuristique qui, à chaque itération, recherche une configuration meilleure que la configuration courante et l'implante. Une telle approche est réaliste dans un environnement sans contrainte énergétique. Elle avance à tâtons jusqu'à se stabiliser. Dans notre cas, nous ne pouvons pas agir de la sorte et risquer d'épuiser les ressources des périphériques à chaque itération. Nous devons trouver la meilleure configuration le plus rapidement possible. Comme [77], nous évaluons les configurations depuis la meilleure selon nos critères de *QdS Utilité* jusqu'à trouver la première qui respecte les critères de *QdS Pérennité* et nous la déployons. Si aucune configuration n'est trouvée, nous élargissons alors peu à peu l'ensemble des configurations possibles en modifiant tout d'abord les critères de *QdS Utilité* puis en modifiant les critères de *QdS Pérennité*. Si après le déploiement, la solution implantée n'est pas la meilleure, la plate-forme le détecte et recommence le processus de recherche d'une configuration.

4.7.1 Heuristique de choix d'une configuration a deployer

Il s'agit ici de trouver une configuration qui satisfasse les contraintes du contexte et permette de fournir une application obéissant aux critères de qualité de service. La recherche de la meilleure configuration, c'est-à-dire de l'assemblage optimum de composants réalisant un service, est un problème connu comme étant NP-complet dans le cas général,

tant sur le choix des composants [83][40] que sur le choix de l'ordonnancement de ces composants [29][62][25][79]. Nous devons alors définir des règles permettant d'obtenir rapidement une solution réalisable mais pas nécessairement optimale. C'est la définition de l'heuristique.

Dans ce paragraphe nous proposons une heuristique de choix de la configuration à déployer permettant de guider l'ordre d'évaluation des déploiements possibles de façon à pouvoir garder la première solution viable trouvée. Pour cela nous proposons un ensemble de règles basé sur les définitions des deux types de qualité de service, *QdS Utilité* et *QdS Pérennité*. Ainsi, cette heuristique se décompose en deux parties : (A) la première permet de sélectionner une configuration. Cette sélection correspond à la mesure de la *QdS Utilité*. (B) La seconde évalue quant à elle la *QdS Pérennité* de la configuration sélectionnée.

A) *Sélection d'une configuration* La recherche d'une configuration se base sur le classement des configurations réalisé lors de l'étape 4 de la méthodologie (section 3.5.4). La première configuration sélectionnée est la configuration qui possède la meilleure note de *QdS Utilité* par rapport à la situation de contexte courante, puis cette configuration est évaluée. Si le résultat de l'évaluation conclut que le déploiement est faisable, alors la configuration est déployée. Dans le cas contraire, la configuration suivante dans le classement est évaluée, jusqu'à ce que l'évaluation obtienne un résultat positif ou jusqu'à ce que la totalité des configurations soient évaluées.

B) *Evaluation d'un déploiement* L'évaluation commence par le placement des composants liées aux périphériques de par la configuration choisie (exemple : un composant de restitution vidéo sur le périphérique de l'utilisateur). Après quoi, il faut trouver un périphérique d'accueil à chacun des composants non placés. Dès qu'un périphérique possible pour un composant est proposé, une étude de faisabilité est menée sur ce périphérique en termes de consommation de ressources physiques, CPU, mémoire et énergie. Lorsqu'un périphérique satisfaisant est sélectionné, on considère ce composant comme étant lié à ce périphérique et on peut procéder récursivement pour les composants suivants. L'heuristique consiste dans l'ordre de choix des composants à sélectionner et à placer. L'idée est de minimiser le nombre de périphériques utilisés ainsi que le nombre de liaisons réseau parcourues (pénalisantes dans le cas des périphériques contraints). C'est pourquoi le composant à placer sera choisi parmi ceux en lien direct avec l'un de ceux déjà placés. De la même façon, l'heuristique consiste dans le choix de l'ordre des périphériques proposés pour un composant. C'est pourquoi le composant sélectionné sera en priorité placé sur un l'un des périphériques accueillant un composant avec lequel il est en liaison directe (Remarque : le coût d'une liaison entre deux composants sur un même périphérique est négligeable). Les périphériques envisagés ensuite seront pris en s'éloignant de 1 puis 2, etc...hops. Comme tant que tous les composants de la configuration ne sont pas placés, il n'est pas possible d'évaluer la charge du réseau, cette évaluation n'est faite que lors du placement du dernier composant de la configuration. Si cette évaluation montre qu'au moins l'un des liens ne peut être assuré, le périphérique suivant de la liste est proposé pour le dernier composant placé. Si aucun autre périphérique n'est acceptable, on remonte dans la récursivité au composant précédent (backtracking), etc. Ainsi, l'heuristique teste toutes les solutions de placement possibles des composants jusqu'à trouver la première solution qui respecte la *QdS Pérennité* tant au niveau de la consommation des ressources physiques

que de la consommation du réseau. Si l'heuristique échoue, alors il n'existe pas de solution de déploiement pour cette configuration. L'heuristique reprend la sélection (A), descend dans le classement et réitère l'évaluation jusqu'à trouver une configuration qui offre une solution viable à déployer.

Le seul échec total de l'heuristique est lorsque celle-ci ne peut pas proposer de solution, c'est-à-dire que le classement des configurations a été parcouru dans sa totalité sans pouvoir parvenir à un déploiement viable. Dans ce cas, l'application ne peut pas être adaptée à la situation de contexte.

La sélection d'une configuration repose donc sur un classement des configurations qui correspond à la préoccupation de respect des contraintes d'utilisation (*QdS Utilité*) puis de la meilleure *QdS Pérennité*.

Ainsi l'heuristique cherche à satisfaire dans l'ordre :

A. Le respect de la *QdS Utilité*
   a. Respect des contraintes d'utilisation en accord avec la mise à jour des notes de *QdS Utilité* des configurations relative au contexte actuel (section 3.5.4).
   b. Sélection d'un ensemble de configurations possibles respectant le seuil de *QdS Utilité*.
B. Respect de la *QdS Pérennité*
   a. Respect de la *QdS Pérennité* au niveau de la consommation des ressources physiques.
   b. Respect de la *QdS Pérennité* au niveau de la consommation du réseau.

4.7.1.1 Sélection d'une configuration

L'heuristique de choix de la configuration à déployer doit répondre à la problématique de QdS que nous avons définie : Utilité et Pérennité. La *QdS Utilité* regroupe le respect des contraintes d'utilisation et le respect de la demande de l'utilisateur. Ces contraintes font partie du cahier des charges de l'application. Nous estimons que le fonctionnement de l'application ne peut pas aller à l'encontre de ce qui a été spécifié dans le cahier des charges. C'est pourquoi le processus de sélection d'une configuration que nous proposons teste tout d'abord le respect des contraintes d'utilisation et de l'utilisateur. Le respect de ces contraintes se traduit par la mise à jour des notes de *QdS Utilité* des configurations que nous avons définies dans l'étape 3 de la méthodologie. Les configurations sont ensuite classées par ordre croissant de note de QdS. Prenons l'exemple de l'application de visite d'un musée. Cette application propose trois services : *Description*, *Guidage* et *Statistiques*. Pour chaque service nous décrivons les différents assemblages qui sont capables de les réaliser.

Par exemple, dans la figure 4.19, le service Description est décrit selon 7 assemblages, appelés CF1 à CF7, et classés en fonction de leur *QdS Utilité*, . Ces configurations sont ensuite classées en fonction du contexte (section 3.5.4). Ainsi par exemple, dans le cas où le contexte autorise l'utilisation du son (pas de conférence en cours) le classement pourrait être : CF1-CF2-CF3-CF4-CF5-CF6-CF7 tandis que lorsqu'il est préférable de ne pas utiliser de son (conférence en cours) le classement deviendrait : CF4-CF5-CF1-CF2-CF3-CF6-CF7. L'heuristique teste alors les configurations en descendant dans ce

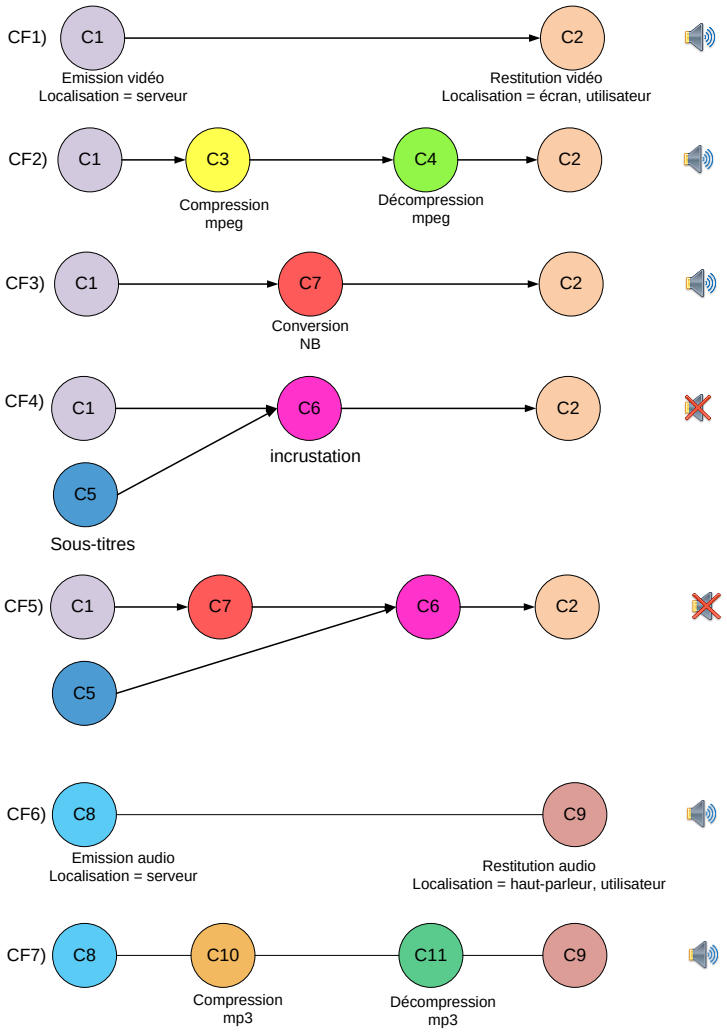

FIGURE 4.19 – Configurations du service Description

classement depuis la meilleure configuration jusqu'à trouver la première qui réponde aux critères de QdS. Ce test est effectué par l'algorithme d'évaluation d'une configuration décrit dans le paragraphe suivant.

4.7.1.2 Evaluation d'un deploiement

A présent que nous avons sélectionné une configuration, nous devons l'évaluer afin de trouver un déploiement correspondant au critère de *qualité de service Pérennité* que nous avons défini dans le chapitre 2. Dans la troisième étape de la méthode de conception (section 3.5.3), chaque configuration a été décrite comme un assemblage C ordonné de composants : $C = \{C_1, \ldots, C_n\}$.

Prenons l'exemple de la configuration CF5 du service Description (figure 4.19), nous

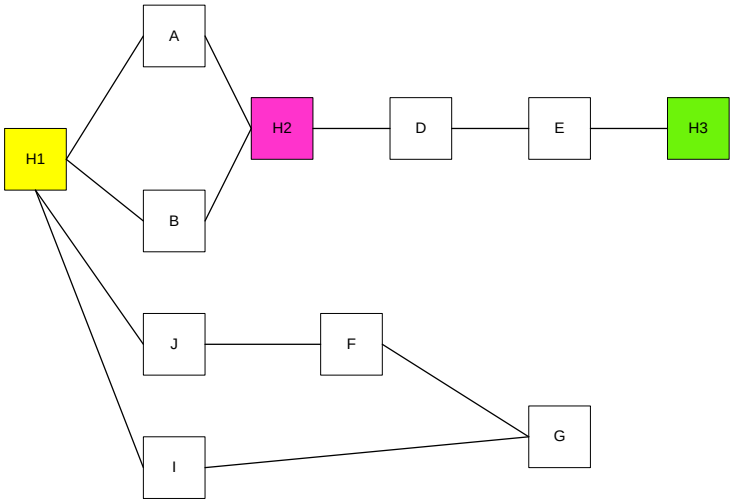

Figure 4.20 – Exemple du réseau du musée à un instant donné

obtenons l'assemblage C suivant : $C = \{C_1,\ C_3,\ C_5,\ C_6,\ C_7\}$. Certains de ces composants sont des composants non-déplaçables. En effet, il existe des composants dont le fonctionnement est lié à un matériel ou à un utilisateur ce qui signifie qu'il est lié au périphérique de cet utilisateur. Par exemple : dans le cas d'un utilisateur qui demande le service Description, une configuration possible comporte un composant de restitution vidéo (C_2). Ce composant ne peut être utilisé qu'avec la présence d'un écran. Cependant, il dépend également de la localisation. Il ne peut pas être déployé sur n'importe quel dispositif, il doit être déployé sur le dispositif de l'utilisateur qui a demandé le service. Par conséquent, parmi les périphériques disponibles, certains sont imposés par les composants de la configuration sélectionnée. Nous notons H, l'ensemble des périphériques imposés : Ainsi, parmi les composants de CF5, C_1 et C_2 sont des composants non-déplaçables. Ils sont respectivement associés au serveur de description H_1 et au périphérique de l'utilisateur H_3 (figure 4.20). L'ensemble H est alors composé de la façon suivante : $H = \{H_1, H_3\}$.

Remarque. L'exemple du musée étant volontairement simple pour illustrer les concepts de base, il ne se prête pas à une démonstration complète du fonctionnement de l'heuristique. C'est pourquoi, pour illustrer la suite du déroulement de cette étape, nous considérerons le graphe de configuration de la figure 4.21 , le graphe du réseau étant celui présenté à la figure 4.20.

Composants non déplaçables de par la configuration étudiée :
- Le composant C_1 est fixé sur le périphérique H_1
- Les composants C_3 et C_4 sont fixés sur le périphérique H_2
- Les composants C_7 et C_8 sont fixés sur le périphérique H_3

L'heuristique se décompose en 5 étapes décrites ci-dessous.

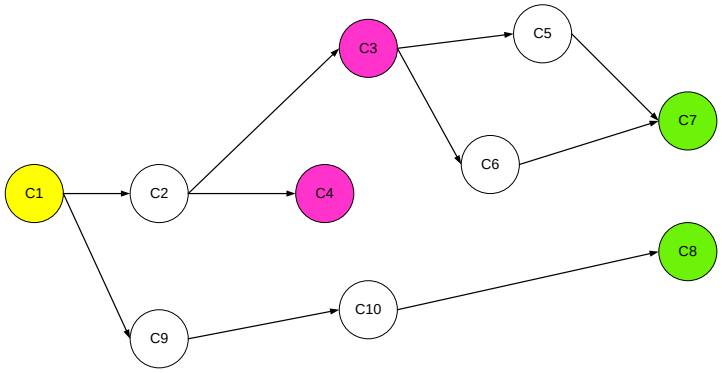

FIGURE 4.21 – Exemple d'une configuration

Etape 1 : Evaluation du déploiement des composants non-déplaçables Nous commençons tout d'abord par évaluer le déploiement des composants C_i placés sur des périphériques définis h_j. Pour cela nous procédons en deux sous-étapes :

1. Disponibilité du matériel. Cette étape consiste à vérifier que les périphériques disposent du matériel imposé par les composants non-déplaçables.
2. Evaluation des ressources. Dans la suite de ce paragraphe nous appelons ressources l'ensemble des trois ressources physiques communes à tous les périphériques : charge CPU, mémoire et énergie. Cette étape consiste à calculer le rapport de consommations des ressources du périphérique par le composant. Puis nous vérifions que ce rapport est bien supérieur au seuil de QdS défini.

Disponibilité du matériel Tout d'abord nous devons vérifier que le périphérique dispose du matériel imposé par le composant. Par exemple pour un composant de restitution vidéo, le périphérique doit disposer d'un écran. Nous comparons les cartes d'identité du composant (balise `<dependance>`) et du périphérique (balise `<propriete>`) qui ont été établies préalablement lors de l'étape 6 de la méthode de conception (section 3.5.6). Si elles correspondent, nous pouvons passer à la sous-étape suivante. Sinon, l'évaluation se termine pour cette configuration et nous devons recommencer le processus avec la configuration suivante du classement.

```
1  <Composant nom= C_1, fonction=EmissionVideo>
2  <role> video </role>
3  <type> Fixe </type>
4  <dependance>ecran/presence=oui</dependance>
5  <Statique>
6      <SE> linux </SE>
7      <CPU> 0.2 </CPU>
8      <Mem> 0.2 </Mem>
9      <Energie> 0.3 </Energie>
10     <DebitS> 24 </DebitS>
11 </Statique>
12 <Dynamique>
```

```
13      <etat> stop </etat>
14      <periph> H_1 </periph>
15      <prec> </prec>
16      <suiv> $C_5$ </suiv>
17  </Dynamique>
18  <Propriete>
19  </Propriete>
20  </Composant>
```

Listing 4.1 – Exemple de carte d'identité : le composant C_1

```
1
2  <Peripherique nom=H_3>
3      <type>CDC</type>
4  <Statique>
5      <SE>Windows CE</SE>
6  </Statique>
7  <Dynamique>
8      <CPU>0.7</CPU>
9      <Mem>0.7</Mem>
10     <Energie>1</Energie>
11 </Dynamique>
12 <propriete>
13     <ecran>
14     <presence>oui</presence>
15     <resolution>320x200</resolution>
16     <couleur>65535</couleur>
17     </ecran>
18 </propriete>
19 </Peripherique>
```

Listing 4.2 – Exemple d'une carte d'identité : le périphérique H_1

Evaluation des ressources Après avoir vérifié la disponibilité du matériel, nous devons nous assurer que le périphérique dispose des ressources nécessaires pour accueillir le composant. Pour cela, la carte d'identité des composants recense la consommation moyenne en charge CPU, en mémoire et en énergie. Nous évaluons la capacité d'un périphérique à accueillir un composant par la formule de *QdS Pérennité* de consommation des ressources :

$$\text{Note QdS } \mathrm{Pe_R(C_i)} = \max(0, \min(C_{H_j} - C_{C_i}, M_{H_j} - M_{C_i}, E_{H_j} - E_{C_i}))$$

C_{H_j} : % de charge CPU disponible sur le périphérique H_j.

C_{C_i} : % de charge CPU consommée par le composant C_i.

M_{H_j} : % de mémoire disponible sur le périphérique H_j.

M_{C_i} : % de mémoire consommée par le composant C_i.

E_{H_j} : % d'énergie disponible sur le périphérique H_j.

E_{C_i} : % d'énergie consommée par le composant C_i.

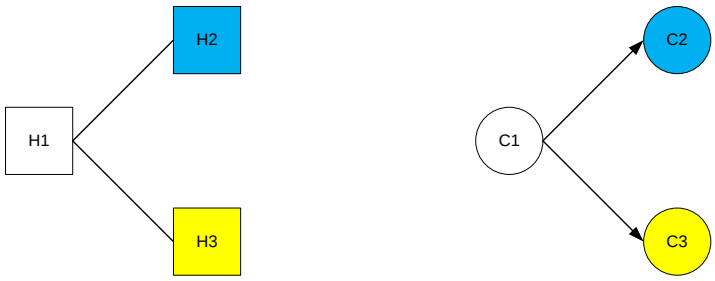

FIGURE 4.22 – Exemple d'un placement sans solution si le graphe de réseau est considéré orienté

Dans notre exemple :

$$\begin{aligned} \text{Note QdS } \mathrm{Pe_R(C_1)} \textit{ sur } H_3 &= \max(0, \min(0.7-0.2, 0.7-0.2, 1-0.3))) \\ &= \max(0, \min(0.7-0.2, 0.7-0.2, 1-0.3)) \\ &= \max(0, \min(0.5, 0.5, 0.7)) \end{aligned} \quad (11)$$

Si la note de qualité de service est supérieure ou égale au seuil de QdS, alors le placement pour ce composant est retenu et nous réitérons ces deux sous-étapes avec le composant non-déplaçable suivant. Sinon le processus d'évaluation prend fin pour cette configuration et recommence avec la configuration suivante dans le classement. Par exemple, le seuil fixé est de 0.2. La note de QdS de C_1 sur H_1 est supérieure au seuil, la QdS est garantie pour ce placement, nous pouvons passer au composant non-déplaçable suivant, à savoir C_2. Si le déploiement de tous les composants non-déplaçables respectent le seuil de QdS alors nous pouvons passer à l'étape suivante (étape 2) afin de placer les autres composants.

Etape 2 : Recherche dans la configuration des chemins entre les composants placés pris deux à deux. Cette étape consiste à trouver tous chemins possibles entre les composants non déplaçables afin de pouvoir ensuite les faire correspondre avec des chemins du graphe du réseau. Pour cela nous devons considérer la configuration comme un graphe non-orienté. En effet, les liaisons réseau étant bidirectionnelles, le réseau est représenté par un graphe non-orienté. Par conséquent, si nous considérions la configuration comme un graphe orienté, nous ne trouverions pas toujours de solution. L'exemple de la figure 4.22 en montre la raison.

Dans cet exemple nous définissons C_2 et C_3 comme des composants non-déplaçables que nous plaçons respectivement sur les périphériques H_2 et H_3. La recherche de chemin entre C_2 et C_3 dans le graphe orienté de configuration n'aboutit pas puisque C_2 et C_3 n'ont pas de suivant. Par conséquent, l'heuristique ne peut pas proposer de placement pour C_1 alors qu'il existe des solutions sur le graphe du réseau : C_1 peut être placé sur H_1 ou sur H_2 ou sur H_3.

En ne tenant pas compte du sens des liaisons dans le graphe de la configuration, nous trouvons les chemins : $C_2 - C_1 - C_3$ et $C_3 - C_1 - C_2$. En termes de routes réseau,

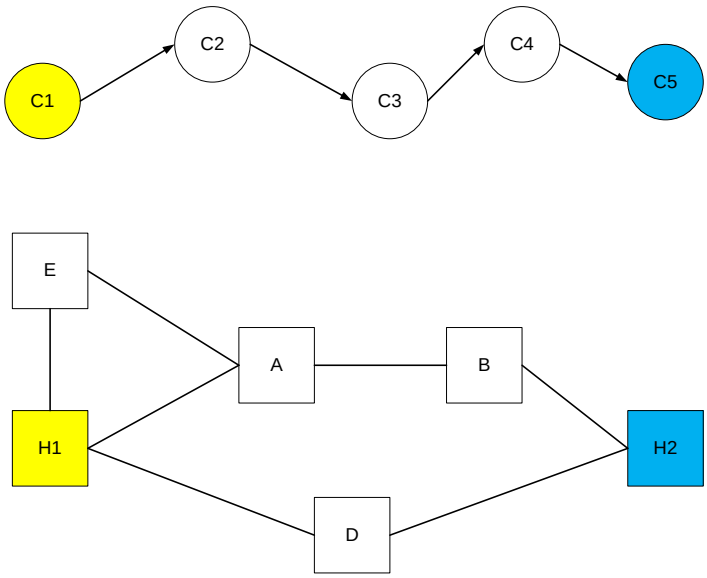

Figure 4.23 – Exemple d'une topologie comportant plusieurs choix de placements des composants

nous cherchons les routes entre H_2 et H_3 et entre H3 et H2 ce qui nous donne les routes suivantes : $H_2 - H_1 - H_3$ et $H_3 - H_1 - H_2$. L'heuristique pourra alors tenter de placer le composant C_1 sur l'un des périphériques de ces routes, c'est-à-dire sur H_2 ou sur H_1 ou sur H_3.

Nous recherchons donc les chemins entre les composants liés à des périphériques dans la configuration sans tenir compte de l'orientation des liens entre composants. Par exemple, lors de la première exécution de l'heuristique, les composants placés sont C_1, C_3, C_4, C_7 et C_8 et les périphériques utilisés sont H_1, H_2 et H_3. Nous calculerons donc les chemins entre les composants sur H_1 et ceux sur H_2, entre les composants sur H_1 et ceux sur H_3, et entre les composants sur H_2 et ceux sur H_3.

Nous obtenons la liste de chemins suivants :
Chemins $H_1 - H_2 : CH_{H_1-H_2} = \{C_1 - C_2 - C_3,\ C_1 - C_2 - C_4\}$
Chemins $H_1 - H_3 : CH_{H_1-H_3} = \{C_1 - C_2 - C_3 - C_5 - C_7,\ C_1 - C_2 - C_3 - C_6 - C_7,\ C_1 - C_9 - C_10 - C_8\}$
Chemins $H_2 - H_3 : CH_{H_2-H_3} = \{C_3 - C_5 - C_7,\ C_3 - C_6 - C_7,\ C_4 - C_2 - C_3 - C_5 - C_7,\ C_4 - C_2 - C_3 - C_6 - C_7,\ C_3 - C_2 - C_1 - C_9 - C_10 - C_8,\ C_4 - C_2 - C_1 - C_9 - C_10 - C_8\}$
A laquelle, viennent s'ajouter les chemins miroirs ($CH_{H_2-H_1}$,$CH_{H_3-H_1}$,$CH_{H_3-H_2}$).

Lors de l'étape 4 (section 4.7.1.2) consistant en l'évaluation d'un déploiement, ces chemins seront utilisés pour effectuer un classement permettant de déterminer l'ordre dans lequel seront pris les composants pour une tentative de placement. L'exemple de la **figure 4.23**. permet d'expliquer les raisons du choix du mode de classement retenu. On suppose dans cet exemple que C_1 est déjà placé sur H_1 et C_5 sur H_2 et que nous devons placer l'un des composants restant (C_2, C_3 ou C_4).

Le placement d'un composant comme C_3 dépend du placement de C_2 et de C_4. Placer C_3 en premier pourrait induire des placements pour C_2 et C_4 qui ne feraient pas partie des solutions les plus simples. Par exemple, si l'on place C_3 sur le périphérique E proche de H_1 mais pas de H_2, l'heuristique est ensuite obligée de « faire un détour »par E pour chercher des placements pour C_2 et C_4 alors qu'il existe des solutions plus directes, donc moins gourmandes en énergie. Des solutions plus raisonnables consistent à placer C_2 près de H_1 et C_4 près de H_2 et à placer ensuite C_3 au mieux afin de réduire les liaisons réseaux.

Afin de mettre en place un tel ordre d'évaluation, nous définissons un poids pour chaque composant.

Définition 4.1 (Définition du poids d'un composant) *Nous définissons le poids d'un composant comme étant son rang minimal dans les chemins précédemment définis auxquels il appartient (incluant les chemins miroirs). Le poids représente la distance minimale qu'a ce composant avec un composant fixé sur un périphérique. Comme les composants fixés sur des périphériques sont toujours au début d'un chemin (par construction) ils ont toujours un poids de* 0*.*

Théorème 4.1 *Si tous les composants ne sont pas de poids* 0 *alors il en existe au moins un qui est de poids* 1*.*

Démonstration. Nous définissons l'ensemble E comme celui de tous les composants dont le poids est différent de 0. Puisque tous les composants ne sont pas de poids 0 (hypothèse du théorème), E est non vide. Considérons C le composant ayant le poids minimal dans cet ensemble E. Nous avons alors deux hypothèses :

- - Soit C a un poids égal à 1 et il existe bien un composant de poids 1 (CQFD),
- - Soit C a un poids égal à N où $N \geqslant 2$.

D'après la définition du poids d'un composant, si C a un poids égal à N, alors il existe au moins un chemin dans la liste déterminée précédemment, dans lequel C est en position N. Ce chemin est de la forme : $K - \ldots X - C - \ldots$ donc il existe un composant X dont le poids est au plus égal à $N - 1$ puisque X précède C dans au moins un chemin et que le poids de C est égal à N.

Le poids de X ne peut pas être égal à 0 sinon celui de C serait égal à 1 puisqu'il est le suivant de X dans ce chemin. Par conséquent, le poids de X est supérieur à 0 et, selon la définition de l'ensemble E, X appartient à E. Ceci qui implique que C n'est pas le composant de rang minimal de E puisque X est dans E et a un rang inférieur à celui de C. Nous tombons sur une contradiction. Cette démonstration par l'absurde montre qu'il est impossible que le poids de C ne soit pas égal à 1.

Ainsi, selon la définition du poids d'un composant, dans l'exemple de la figure 4.23, les composants ont les poids suivants :

poids 0 : $C_1,\ C_5$.

C_1 et C_5 sont des composants non-déplaçables. Ils sont fixés sur des périphériques et par construction des chemins, ils apparaissent en début et fin de chemin.

poids 1 : $C_2,\ C_4$

C_2 et C_4 sont des suivants et précédents directs de C_1 et C_5 sur au moins un des chemins. Ils sont bien à une distance de 1 de composants de poids 0.

poids 2 : C_3

C_3 est à une distance minimale de 2 de C_1 et de C_5 dans tous les chemins.

Nous obtenons la liste classée de composants LC : $LC = \{C_1,\ C_5,\ C_2,\ C_4,\ C_3\}$

Après avoir déterminé tous les chemins possibles dans la configuration et les poids des composants, nous allons déterminer l'ensemble des routes possibles dans le réseau permettant de relier deux à deux les périphériques déterminés par les composants déjà placés.

Etape 3 : Recherche de toutes les routes possibles entre les périphériques imposés pris deux à deux Comme les transmissions réseau sont des activités coûteuses en énergie, nous déterminons la liste des routes possibles (également appelé domaine [MUSIC09]) permettant d'atteindre deux à deux l'ensemble des périphériques auxquels des composants ont été affectés dans l'étape 1 (périphériques imposés). Cette liste est ensuite soumise à une série de classements permettant de définir l'ordre d'évaluation des périphériques : le périphérique situé en haut du classement est celui qui a le moins de contraintes en termes de ressources.

Tout d'abord, nous déterminons les routes possibles entre les périphériques imposés pris deux à deux. Pour cela nous définissons une route comme un ensemble ordonné de périphériques dans lequel chacun ne peut apparaître qu'une seule fois.

Si nous prenons l'exemple de la **figure 4.20**, nous trouvons les routes suivantes :
$\text{Routes}\mathrm{H_1 - H_2} : \mathrm{R_{H_1-H_2}} = \{\mathrm{H_1 - A - H_2},\ \mathrm{H_1 - B - H_2}\}$
$\text{Routes}\mathrm{H_1 - H_3} : \mathrm{R_{H_1-H_3}} = \{\mathrm{H_1 - A - H_2 - D - E - H_3},\ \mathrm{H_1 - B - H_2 - D - E - H_3}\}$
$\text{Routes}\mathrm{H_2 - H_3} : \mathrm{RH_2 - H_3} = \{\mathrm{H_2 - D - E - H_3}\}$
Auxquelles s'ajoutent les routes miroir.

De la même façon que nous l'avons fait pour les composants ces routes seront utilisées lors de l'étape 4 pour classer les périphériques candidats à l'accueil des composants. Nous définissons un poids pour les périphériques. Tout comme pour les composants, ce poids permettra d'établir une liste ordonnée de périphériques à tester pour le placement d'un composant.

Définition 4.2 (Définition du poids d'un périphérique) *Nous définissons le poids d'un périphérique dans une liste de routes comme étant son rang minimal dans les routes auxquelles il appartient. Le poids représente la distance minimale (en nombre de hops) qu'a ce périphérique avec un périphérique accueillant un composant déjà affecté. Comme les périphériques imposés sont toujours au début d'une route (par construction) ils ont toujours un poids de 0.*

Sur l'exemple, les routes trouvées font apparaître les périphériques H_1, A, H_2, B, D, E, F et H_3.

Lorsqu'il existe des périphériques de poids identique, nous décidons de les classer selon leur type. Ce deuxième classement permet de considérer tout d'abord les périphériques possédant le moins de contraintes et donc permettant d'assurer un fonctionnement le

Périphérique	H_1	H_2	H_3	A	B	C	D	F	E
Poids	0	0	0	1	1	1	1	1	2

TABLE 5 – Liste classée par poids des périphériques candidats à la tentative de placement de C_2

plus long possible. Un tel classement permet de proposer un déploiement qui sollicite au maximum les périphériques les moins contraints.

Nous distinguons trois types de périphériques :
- **Fixe** : ce sont les périphériques dont les contraintes de ressources sont négligeables face aux deux autres types. Typiquement nous classons parmi les périphériques fixes les ordinateurs de bureau et les ordinateurs portables.
- **CDC** : ce sont les périphériques dont les contraintes de ressources correspondent aux contraintes prises en compte dans la spécification du standard CDC de J2ME. Nous classons parmi ces périphériques les PDA et les téléphones portables dont les caractéristiques répondent au standard, comme les téléphones de dernière génération (IPhone, Nexus One, etc.).
- **CLDC** : ce sont les périphériques dont les contraintes de ressources correspondant aux contraintes prises en compte dans la spécification du standard CLDC de J2ME. Nous classons parmi ces périphériques les téléphones portables « simples »et les capteurs.

Prenons l'exemple des périphériques de la table 5 :

Périphérique	H_1	H_2	H_3	A	B	C	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	CDC	Fixe	CLDC	CLDC	CLDC

TABLE 6 – Types des périphériques de la table 5

Après le classement selon le type nous obtenons la table 7 :

Périphérique	H_1	H_2	H_3	A	C	B	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	Fixe	CDC	CLDC	CLDC	CLDC

TABLE 7 – Classement des périphériques voisins selon leur type

En gris nous obtenons les périphériques classés, les autres périphériques nécessitent un autre classement puisqu'ils possèdent encore des caractéristiques identiques : même poids et même type.

Si après ce deuxième classement il existe toujours des périphériques ex aequo, c'est-à-dire qui ont le même poids et le même type, toujours pour des raisons de préservation de l'énergie, nous effectuons un troisième classement en fonction du taux d'énergie disponible sur les périphériques. L'énergie disponible est un critère important puisqu'il concerne directement la durée de vie du périphérique et donc de l'application. Ses variations risquent

de provoquer des reconfigurations plus fréquemment que les variations d'utilisation du CPU ou de la mémoire.

Dans la table 8, nous renseignons les valeurs d'énergie disponible pour chaque périphérique que nous devons encore classer.

Remarque. Concernant les périphériques de type Fixe, il n'y a pas de valeur d'énergie disponible. En effet, l'énergie dont il dispose ne varie pas puisqu'il est branché à une source d'énergie en continu.

Périphérique	H_1	H_2	H_3	A	C	B	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	Fixe	CDC	CLDC	CLDC	CLDC
Energie disponible		1	0.8	-	-		0.2	0.3	

Table 8 – Exemple de valeur d'énergie disponible pour les périphériques *ex aequo*

Après l'application de la règle de classement sur l'énergie disponible, nous obtenons le tableau 9. Les périphériques en gris sont les périphériques désormais classés et en blanc, ce sont les périphériques qui possèdent encore des caractéristiques identiques.

Périphérique	H1	H2	H3	A	C	B	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	Fixe	CDC	CLDC	CLDC	CLDC
Energie disponible		1	0.8	-	-		0.2	0.3	

Table 9 – Classement selon l'énergie disponible

Pour départager les périphériques encore identiques après ces trois classements, nous effectuons un quatrième classement en fonction du taux de CPU encore disponible sur les périphériques. Le taux d'utilisation du CPU est un critère moins contraignant que l'énergie cependant, un taux élevé peut ralentir le fonctionnement des composants supportés par le périphérique.

Dans la table 10, nous renseignons les valeurs d'utilisation du CPU disponible pour chaque périphérique que nous devons encore classer.

Périphérique	H1	H2	H3	A	C	B	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	Fixe	CDC	CLDC	CLDC	CLDC
Energie disponible		1	0.8	-	-		0.2	0.3	
CPU disponible				0.9	0.75				

Table 10 – Exemple de valeur de CPU disponible pour les périphériques *ex aequo*

Après l'application de la règle de classement sur le taux d'utilisation de CPU disponible, nous obtenons le tableau 11. Désormais, tous les périphériques sont en gris, ils sont tous classés.

Périphérique	H1	H2	H3	A	C	B	D	F	E
Poids	0	0	0	1	1	1	1	1	2
Type	Fixe	CDC	CDC	Fixe	Fixe	CDC	CLDC	CLDC	CLDC
Energie disponible		1	0.8	-	-		0.2	0.3	
CPU disponible				0.9	0.75				

TABLE 11 – Classement selon le CPU disponible

Si toutefois, il restait des périphériques dont les caractéristiques sont toujours identiques, nous effectuons un ultime classement en fonction de la mémoire disponible bien que celle-ci n'ait pas d'influence sur la pérennité à ce stade du fonctionnement de l'application. En effet, la mémoire disponible sur les périphériques n'intervient que lorsqu'on souhaite ajouter d'autres composants sur un périphérique.

Enfin, si malgré cette série de cinq classements, il reste toujours des périphériques ex aequo, ils seront classés au hasard.

Cette liste détermine l'ordre dans lequel sont évalués les périphériques d'accueil des composants restants. En effet, afin d'éviter un trop grand nombre de liaisons réseau, nous tentons tout d'abord de placer les composants sur les périphériques imposés. Puis nous essayons de les placer sur les périphériques les plus accessibles (en nombre de *hops*) et les moins contraints (*ressources*).

L'objectif d'une heuristique étant de trouver une solution rapide à un problème NP-difficile, nous proposons de limiter le nombre de solutions à explorer dans un premier temps. Si aucune solution n'est obtenue, ce nombre sera augmenté. Dans ce but nous définissons deux opérateurs sur les routes : l'opérateur d'intersection et l'opérateur d'union.

Définition 4.3 (Opérateur d'intersection) *Nous définissons un opérateur d'intersection de routes comme étant l'ensemble des périphériques apparaissant dans toutes les routes R_i :*

$$\bigcap_{i=0}^{n} R_i = \{\textit{périphériques apparaissant dans tous les Ri}\}$$

un tel ensemble peut etre vide

Sur l'exemple :
$R_{H_1-H_2} \bigcap R_{H_1-H_3} = \{H_1, A, H_2, B\}$
$R_{H_1-H_2} \bigcap R_{H_2-H_3} = \{H_2\}$
$R_{H_1-H_3} \bigcap R_{H_2-H_3} = \{H_2, D, E, H_3\}$
$R_{H_1-H_2} \bigcap R_{H_1-H_3} \bigcap R_{H_2-H_3} = \{H_2\}$

Définition 4.4 (Opérateur d'union) *Nous définissons un opérateur d'union de routes comme étant l'ensemble des périphériques apparaissant dans au moins l'une des routes*

R_i :

$$\bigcup_{i=0}^{n} R_i = \{\textit{périphériques apparaissant dans au moins l'une des Ri}\}$$

un tel ensemble ne peut pas etre vide.

Sur l'exemple :
$R_{H_1-H_2} \bigcup R_{H_1-H_3} = \{H_1, A, H_2, B, D, E, H_3, C, F\}$
$R_{H_1-H_2} \bigcup R_{H_2-H_3} = \{H_1, A, H_2, B, D, E, H_3\}$
$R_{H_1-H_3} \bigcup R_{H_2-H_3} = \{H_1, A, H_2, B, D, E, H_3, C, F\}$
$R_{H_1-H_2} \bigcup R_{H_1-H_3} \bigcup R_{H_2-H_3} = \{H_1, A, H_2, B, D, E, H_3, C, F\}$

Etape 4 : Evaluation d'un déploiement Pour évaluer la pérennité d'un déploiement nous procédons de la façon suivante :

1. Nous reprenons la liste LC des composants et les classons par leur poids comme expliqué lors de l'étape 2 (section 4.7.1.2).
2. Nous sélectionnons le premier composant de poids 1 de cette liste.
   a. S'il n'en existe aucun, cela signifie que tous les composants ont un poids égal à 0 (section 4.1) et qu'ils sont tous placés. L'évaluation se termine avec succès, il ne reste plus qu'à déployer la configuration.
3. Nous reprenons la liste LP des périphériques et la classons par leur poids et ressources comme expliqué à l'étape 3 (section 4.7.1.2).
4. Nous évaluons l'incidence du placement du composant choisi sur le premier périphérique de LP (calcul identique à celui de l'étape 1 (section 4.7.1.2)).
   a. Si la note de QdS Pérennité de consommation des ressources est supérieure ou égale au seuil, alors le placement de ce composant sur ce périphérique est retenu. Si le placement retenu concerne le dernier composant de la liste LC, alors nous pouvons évaluer l'incidence du déploiement trouvé sur le réseau. Si la configuration peut être déployée sur le réseau au regard des ressources disponibles, le placement de ce composant sur ce périphérique est retenu. On considère alors ce composant comme placé, et on appelle récursivement l'heuristique à l'étape 1.
   b. Sinon nous revenons en (4) avec le périphérique suivant de LP s'il en reste, sinon l'appel récursif en cours se termine par un échec qui provoquera une nouvelle tentative pour le composant précédemment placé.

A présent nous allons illustrer le déroulement de cette étape (étape 4) à l'aide de l'exemple de la **figure 4.21**.

Tout d'abord, l'heuristique calcule le poids des composants (cf. étape 2). Si tous les composants sont de poids 0, cela signifie qu'ils tous sont placés. L'heuristique se termine alors avec succès. Le déploiement trouvé est ensuite transmis au service CalculerConfiguration pour comparer la configuration en cours d'exécution et la nouvelle configuration que l'heuristique vient de trouver et déterminer les commandes à envoyer aux plates-formes des périphériques concernés.

Sinon, l'heuristique sélectionne le premier composant de poids 1 dans la liste et tente de le placer. Sur notre exemple, tous les composants non encore placés sont de poids 1. Au hasard, nous sélectionnons le composant C_2.

Afin de trouver le meilleur placement possible pour C_2, nous sélectionnons les périphériques candidats. Pour cela nous recherchons tous les chemins dans lesquels apparaît C_2 :

$$H_{C_2} = \{CH_i / C_2 \in CH_i\}$$

Sur l'exemple $H_{C_2} = \{CH_{H_1-H_2},\ CH_{H_1-H_3},\ CH_{H_2-H_3}\}$ car C_2 appartient à $CH_{H_1-H_2}$ et $CH_{H_1-H_3}$ et $CH_{H_2-H_3}$.

C_2 peut alors être placé sur n'importe quel périphérique appartenant à l'union des routes, $\bigcup_{i=0}^{n} R_i$, $\{H_1 - H_2\}$, $\{H_1 - H_3\}$ et $\{H_2 - H_3\}$.

Cependant selon la définition de l'intersection (définition 4.3), il parait plus intéressant de placer C_2 parmi les périphériques appartenant à l'intersection des routes $\bigcap_{i=0}^{n} R_i$. En effet, quoi qu'il arrive, toutes les routes passant par C_2 devront obligatoirement passer par l'un des périphériques appartenant à $\bigcap_{i=0}^{n} R_i$. Selon un point de vue réseau, il faudra impérativement que ces périphériques soient utilisés, au moins en tant que relais d'informations, pour interconnecter les composants. En plaçant C_2 sur l'un d'eux, nous profitons du fait qu'il est obligatoire de passer par ce périphérique pour effectuer le traitement fourni par C_2. En revanche, il est toujours possible de placer C_2 sur l'un des périphériques de l'union $\bigcup_{i=0}^{n} R_i$, quitte à « faire un détour »sur le réseau pour passer par C_2. A partir de ce constat, il existe au moins deux façons de procéder :

A. Nous sélectionnons l'union afin d'avoir le plus grand choix possibles pour une tentative de placement. Dans cette union, nous interclassons les périphériques et nous mettons en début de liste ceux qui appartiennent aussi à l'intersection.

B. Nous évaluons la tentative de placement en sélectionnant dans un premier temps uniquement les périphériques de l'intersection. Si la tentative échoue, nous essayons de nouveau en sélectionnant cette fois-ci les périphériques de l'union.

Raisonnons à partir de la solution A. L'heuristique tente tout d'abord de placer les composants parmi les meilleures solutions, c'est-à-dire les périphériques de l'intersection. Puis en cas d'échec, elle va descendre dans la liste des périphériques possibles et va tenter un placement parmi les solutions moins bonnes, c'est-à-dire les périphériques de l'union. La conséquence d'un tel fonctionnement est que l'heuristique peut trouver une solution dans laquelle certains composants ont été placés sur des solutions moins bonnes alors que si l'heuristique était revenue en arrière, elle aurait pu trouver un bon placement (dans l'intersection) d'un composant précédent qui aboutissait à un bon placement de tous les autres. Donc le choix qui a été fait sur le composant précédent a conduit à une solution moins bonne alors qu'il en existait de meilleures.

Avec la solution B, l'heuristique explore tout d'abord les bonnes solutions. S'il n'en existe pas, alors elle recommence en explorant cette fois-ci les moins bonnes solutions.

Nous pensons que la solution B donne un meilleur résultat et choisissons de la mettre en place pour la tentative de placement. Ainsi l'heuristique fonctionne de la façon suivante :

1. L'heuristique évalue les placements des composants, de façon récursive, avec les périphériques de l'intersection. Si elle aboutit, cela signifie qu'elle a trouvé un déploiement qui respecte les critères de QdS.
2. Si elle échoue, elle recommence en utilisant les périphériques de l'union. Si elle aboutit, cela signifie qu'elle a trouvé un déploiement qui respecte les critères de QdS. Sinon c'est un échec, aucun déploiement n'a été trouvé pour cette configuration. Nous sélectionnons alors la configuration suivante dans la sélection (4.7.1.1) et nous recommençons l'évaluation pour cette nouvelle configuration.

Ainsi, après avoir déterminé l'ensemble des chemins où apparaît le composant sélectionné, selon le mode dans lequel se trouve l'heuristique (intersection ou union), elle détermine l'ensemble des périphériques candidats pour une tentative de placement. Dans le cas du composant C_2 :
$H_{C_2} = \{CH_{H_1-H_2},\ CH_{H_1-H_3},\ CH_{H_2-H_3}\}$
donc l'heuristique sélectionne dans un premier temps l'intersection des routes $R_{H_1-H_2}$, $R_{H_1-H_3}$ et $R_{H_2-H_3}$:
$\mathrm{R_{H_1-H_2}} \cap \mathrm{R_{H_1-H_3}} = \{\mathrm{H_1, A, H_2, B}\}$
$\mathrm{R_{H_1-H_2}} \cap \mathrm{R_{H_2-H_3}} = \{\mathrm{H_2}\}$
$\mathrm{R_{H_1-H_3}} \cap \mathrm{R_{H_2-H_3}} = \{\mathrm{H_2, D, E, H_3}\}$
$\mathrm{R_{H_1-H_2}} \cap \mathrm{R_{H_1-H_3}} \cap \mathrm{R_{H_2-H_3}} = \{\mathrm{H_2}\}$

Le premier (et unique) périphérique à tester pour le placement de C_2 est H_2. Dans le cas où il existe plusieurs périphériques à tester, nous appliquons les 5 classements identifiés à l'étape 3 (section 4.7.1.2), à savoir par poids, par type, par énergie, CPU et mémoire disponible.

L'heuristique sélectionne comme périphérique candidat un périphérique de la liste classée, en commençant par le premier puis le suivant dans l'ordre de la liste. Si tous les périphériques de la liste ont été testés, l'heuristique se termine par un échec. Si nous sommes en mode intersection, alors l'heuristique recommence avec le mode union. Sinon, cela signifie qu'il n'y a aucun placement possible pour ce composant et donc, nous devons repartir à l'étape de sélection d'une configuration. L'heuristique sélectionne la configuration suivante dans la liste et démarre son évaluation.

L'heuristique tente de placer le composant sur le périphérique sélectionné. Pour cela elle calcule l'incidence de ce placement en termes de consommation des ressources. Tout comme pour les composants imposés, nous évaluons la possibilité de déploiement du composant C_i sur le périphérique sélectionné, en calculant sa note de *QdS Pérennité* de consommation des ressources (section 3.3.1.2).

Lorsque tous les composants ont été placés, nous devons à présent vérifier que les liaisons réseaux utilisées pour relier ces composants disposent d'un débit disponible suffisant pour que le déploiement obtenu lors de l'évaluation des ressources puisse fonctionner. Pour cela, nous utilisons la formule de QdS Pérennité de consommation du réseau (section 3.3.1.2). Si la note est supérieure ou égale au seuil de tolérance, cela signifie que le déploiement obtenu peut être mis en place. Il respecte les critères de QdS Utilité et Pérennité.

Lorsque l'heuristique échoue, cela signifie qu'aucun déploiement respectant les critères de QdS n'a été trouvé. L'heuristique retourne alors à la sélection d'une configuration et

recommence. Dans le cas où elle ne trouve plus de configuration à sélectionner, il existe deux alternatives :

- Soit il est possible de mettre à jour le seuil de QdS afin de pouvoir sélectionner de nouvelles configurations jusqu'alors non envisagées. Un nouveau classement est effectué avec les nouvelles configurations. L'évaluation est effectuée depuis la première configuration du classement jusqu'à trouver un déploiement qui satisfasse les critères de QdS.
- Soit la mise à jour des seuils ne permet pas de trouver de nouvelles configurations à tester. Toutes les configurations ont été envisagées et aucun déploiement ne satisfait les critères de QdS. La tentative de déploiement a totalement échouée, il n'existe aucune solution adaptée à la situation de contexte. L'application ne peut pas être reconfigurée.

Etape 5 : Déploiement Lorsque tous les composants de la configuration ont pu être placés, c'est qu'une solution a été trouvée et peut être déployée.

Pour chaque périphérique, nous comparons son déploiement actuel qui peut être représenté sous la forme d'un graphe orienté [42], avec le déploiement trouvé. Selon des règles de transformations de graphes, nous pouvons définir les modifications à apporter en termes de connexion et de déconnexion de connecteurs et d'ajout, de suppression ou de migration de composants.

4.8 Conclusion

L'adaptation dynamique dans les applications distribuées est une problématique largement étudiée, notamment dans le domaine de l'informatique pervasive. L'utilisation de périphérique mobiles et contraints a conduit à tenir compte de leurs ressources dans le choix des reconfigurations. Des travaux comme Music [64] ou AxSeL [30] proposent, dans ce sens, des middlewares de reconfiguration tenant compte de l'état des ressources des périphériques dans le choix de la configuration pour l'un, et du déploiement pour l'autre. Cependant, comme la plupart des travaux, ils ne tiennent pas compte du coût de la distribution des composants de la configuration. En effet, les communications réseau sont responsables de 90% de la consommation d'énergie sur un périphérique mobile. C'est pourquoi, afin de garantir un fonctionnement de qualité et en continu des applications, il est intéressant de prendre en compte le coût des liaisons réseau et de le minimiser.

Dans ce chapitre nous proposons Kalimucho, une plate-forme de gestion de la qualité de service des applications pour l'adaptation au contexte. Kalimucho résulte de la collaboration de cinq services : *Superviseur*, *Générateur de Reconfiguration*, *Usine à Conteneur*, *Usine à Connecteur* et *Routage*. Le service *Superviseur* est responsable de la gestion du contexte et de la prise de décision de reconfiguration. Lorsque la décision de reconfigurer l'application est prise, le service *Générateur de Reconfiguration* est chargé de trouver la première configuration offrant une QdS suffisante à l'aide d'une heuristique de choix d'une configuration. L'heuristique fait appel au service *Routage*, chargé de fournir toutes les routes possibles pour relier les périphériques utilisés par la configuration. Enfin, une fois le déploiement trouvé, les services *Usine à Conteneur* et *Usine à Connecteur* sont sollicités

afin d'encapsuler et de déployer tous les composants et les connecteurs intervenant dans le déploiement.

A chaque décision de reconfiguration, Kalimucho fait appel à une heuristique de choix d'une configuration. Cette heuristique évalue les déploiements possibles d'une configuration jusqu'à en trouver un satisfaisant. Elle se déroule en deux temps. Tout d'abord, à chaque itération, elle retient un périphérique d'accueil pour un composant et vérifie que les ressources dont dispose le périphérique satisfont les exigences du composant. Puis, lorsqu'un placement a été trouvé pour tous les composants, elle vérifie que les ressources réseau peuvent supporter toutes les liaisons requises. L'évaluation est guidée par deux principes permettant de satisfaire les critères de *QdS Pérennité* :

- minimiser les liaisons réseau.
  Pour cela, elle se base sur les notions de poids d'un composant et de poids d'un périphérique.
- choisir les périphériques déjà utilisés comme hôte d'accueil en priorité.
  Pour cela elle se base sur le classement des périphériques en fonction de leurs constraintes de ressources

Un prototype, décrit dans le chapitre suivant, permet de valider le fonctionnement de cette heuristique.

Chapitre 5

Prototype

Sommaire

Dans le chapitre 4 nous avons présenté Kalimucho : une plate-forme de reconfiguration et de déploiement contextuel et dynamique d'applications constituées de composants. Le fonctionnement de Kalimucho se base sur une collaboration de 5 services permettant de capturer le contexte, de l'interpréter et de choisir une configuration de QdS satisfaisante lorsqu'une reconfiguration est nécessaire. Le choix d'une configuration fait appel à une heuristique d'évaluation qui étudie la faisabilité du déploiement d'une configuration parmi les périphériques disponibles en tenant compte du contexte.

Dans ce chapitre, nous présentons deux prototypes. Le premier est un similateur d'évaluation qui permet de valider le fonctionnement de l'heuristique, et plus particulièrement, la validation de l'étape B (section 4.7.1.2) : l'évaluation d'une configuration. Il montre que l'heuristique permet de trouver le premier déploiement satisfaisant les critères

de *QdS Pérennité* en minimisant le nombre de liaisons réseau afin de minimiser la consommation d'énergie des composants et de garantir une durée de vie satisfaisante pour l'application. Pour cela il permet de vérifier que l'heuristique choisit bien d'utiliser le plus possible les périphériques qui supportent déjà les composants non déplaçables de la configuration, puis les périphériques les plus proches de ces derniers et en donnant la priorité aux moins contraints. Enfin il montre que l'heuristique obtient un résultat en un nombre de tentatives relativement faible.

Le second prototype est une implémentation de Kalimucho qui permet de valider le fonctionnement de la plate-forme Kalimucho, et plus particulièrement, la validation des actions nécessaires à la reconfiguration telles que la création et la suppression de composants (et leurs conteneurs), la création et la suppression de connecteurs, la migration de composants et la capture du contexte. Il montre que la plateforme est capable de proposer des conteneurs de composants et de connecteurs adaptés à chaque périphérique et que l'exécution de Kalimucho sur des périphériques très contraints tels que les SunSpots, présente des résultats de performances satisfaisants.

Dans les sections suivantes, nous allons présenter tout d'abord l'interface du premier prototype et les fonctionnalités qu'il propose, puis, nous décrirons son fonctionnement que nous illustrerons au travers de cas d'utilisation. Nous présentons ensuite le second prototype et les actions implémentées par la plate-forme, puis, nous analyserons ses performances en termes de temps de réponse aux actions demandées.

5.1 Prototype n° 1 : simulateur d'évaluation

Ce premier prototype que nous proposons est un simulateur d'évaluation d'une configuration. A partir des données relatives au contexte d'exécution telles que les cartes d'identité des composants et des périphériques, et les informations sur les débits des liaisons réseaux, il propose un placement des composants d'une configuration donnée sur les périphériques disponibles. Le prototype se présente sous la forme d'une fenêtre d'application divisée en deux parties (figure 5.1) : la partie supérieure où se trouvent les fonctionnalités et la partie inférieure où est affiché le déroulement de l'heuristique. Le prototype propose six fonctionnalités :

- Ajouter, modifier et consulter un composant
- Ajouter, modifier et consulter un périphérique
- Ajouter, modifier et consulter une configuration
- Créer et modifier une topologie de réseau
- Lancer une simulation
- Afficher le résultat des différentes étapes de l'heuristique.

Chacune de ces fonctionnalités sera décrite dans la suite de cette section.

5.1.1 Ajouter, modifier et consulter un composant

Ajouter un composant, dans le simulateur, signifie que l'utilisateur doit renseigner la carte d'identité du composant. Puisque la partie de l'heuristique que nous souhaitons tester avec ce simulateur ne concerne que la *QdS Pérennité*, nous proposons de réduire la carte

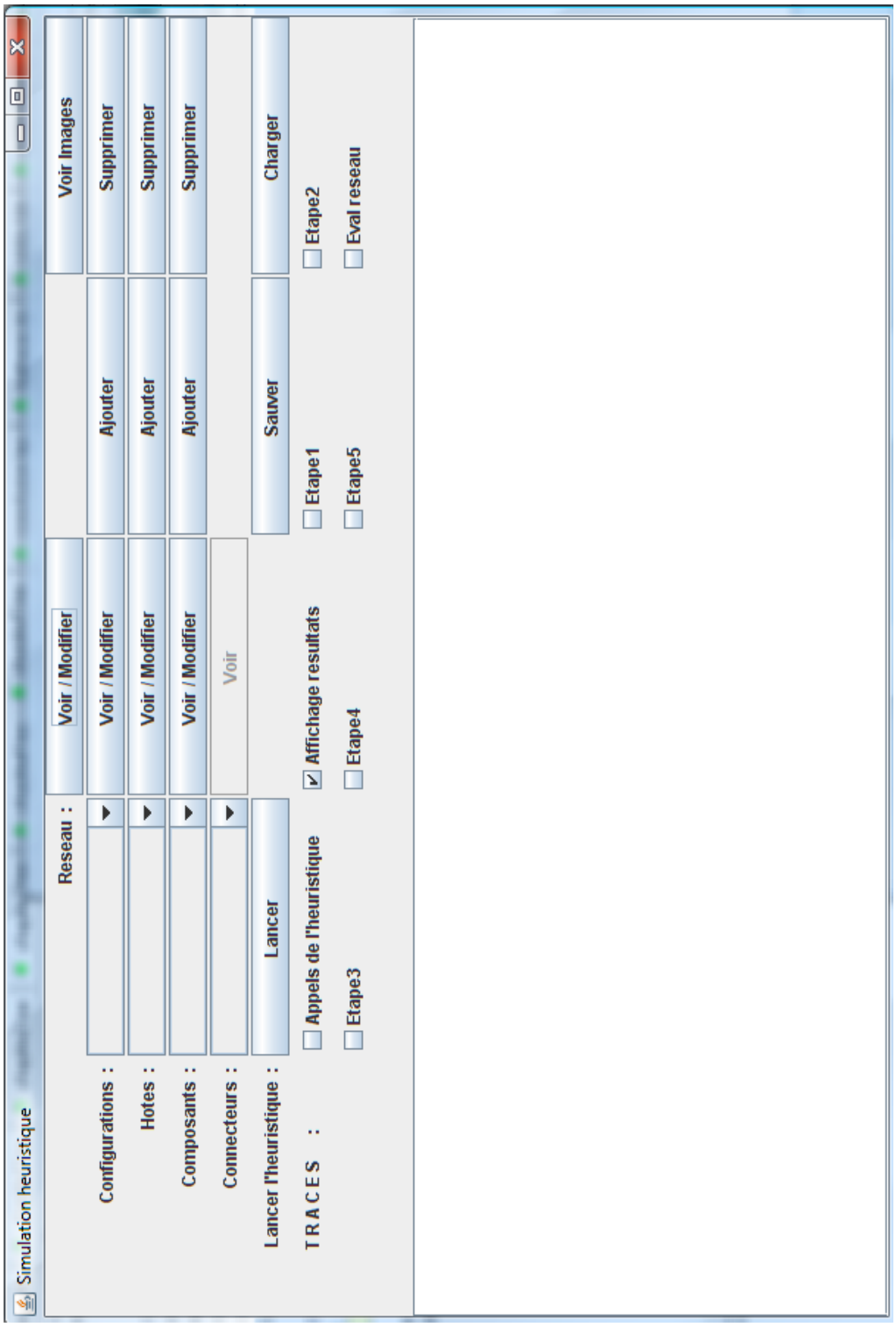

Figure 5.1 – Interface principale du simulateur

d'identité aux seules informations pertinentes. Nous avons défini une carte d'identité d'un composant de la façon suivante (listing 5.1) :

```
<Composant nom, fonction>
    <role>  </role>
    <type>  </type>
    <dependance>  </dependance>
    <Statique>
        <SE>  </SE>
        <CPU>  </CPU>
        <Mem>  </Mem>
        <Energie>  </Energie>
        <DebitS>  </DebitS>
    </Statique>
    <Dynamique>
        <etat>  </etat>
        <periph>  </periph>
        <prec> </prec>
        <suiv>  </suiv>
    </Dynamique>
    <Propriete>
    </Propriete>
</Composant>
```

Listing 5.1 – Carte d'identité d'un composant

Parmi ces caractéristiques, nous nous intéressons particulièrement à celles qui influencent la *QdS Pérennité* telles que la consommation de CPU, la consommation de mémoire, la consommation d'énergie et le débit réseau requis en sortie du composant (partie Statique du listing 5.1).

Conformément à la carte d'identité, pour chaque composant d'une configuration, l'utilisateur doit fournir son nom et les valeurs correspondant à chaque caractéristique (figure 5.2). Pour des raisons de facilité de manipulation, nous avons choisi d'indiquer des valeurs entières pour les cartes d'identité plutôt que des valeurs flottantes tel qu'il a été indiqué dans le modèle de QdS.

Concernant la consommation de CPU, de mémoire et d'énergie, les valeurs se situent entre 0 et 100, afin de correspondre avec les pourcentages de la formule de *QdS Pérennité de consommation de ressources*. Concernant le débit réseau, les valeurs correspondent à un débit en Ko/sec. La formule de *QdS Pérennité de consommation du réseau* comprend une étape de transformation en pourcentage. Ceci permet d'alléger le travail de saisie de toutes les valeurs.

Le bouton voir/modifier permet de consulter et mettre à jour les caractéristiques des composants. Par exemple, modifier un composant permet de simuler l'évaluation de configurations proches, c'est-à-dire qui possèdent des composants de fonction identique, mais avec des caractéristiques différentes.

Une configuration CF1 de diffusion d'images utilise un composant de compression C_j, dont la fonction (`<fonction>`) est de compresser au format *jpeg*. Ses caractéristiques sont les suivantes :

– `<CPU>` : 3

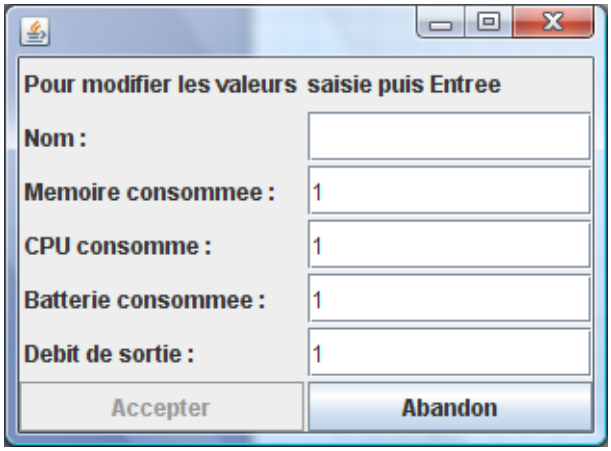

Figure 5.2 – Interface d'ajout, de modification et de consultation d'un composant

– `<Energie>` : 2
– `<Mem>` : 2

Une configuration CF2 de diffusion d'images est constituée des mêmes composants que CF1 sauf qu'à la place de C_j, elle utilise un composant C_t qui compresse au format *tiff*. Ses caractéristiques sont les suivantes :

– `<CPU>` : 4
– `<Energie>` : 2
– `<Mem>` : 2

5.1.2 Ajouter, modifier et consulter un périphérique

Après avoir ajouté tous les composants disponibles dans l'application, nous pouvons ajouter l'ensemble des périphériques disponibles.

Ajouter un périphérique, dans le simulateur, consiste à renseigner sa carte d'identité. Cette dernière (listing 5.2) se base sur le même modèle que celui de la carte d'identité d'un composant.

```
<Peripherique nom>
    <type>  </type>
    <Statique>
        <SE>  </SE>
    </Statique>
    <Dynamique>
        <CPU>  </CPU>
        <Mem>  </Mem>
        <Energie>  </Energie>
    </Dynamique>
    <propriete>
    </propriete>
</Peripherique>
```

Listing 5.2 – Carte d'identité d'un périphérique

Tout comme pour les composants, nous retenons particulièrement les informations qui influencent la *QdS Pérennité* qui sont le niveau de charge CPU (`<CPU>`), le niveau de mémoire disponible (`<Mem>`) et le niveau d'énergie disponible (`<Energie>`). Nous prenons en compte également le type des périphériques. En effet, l'heuristique utilise cette information pour déterminer l'ordre d'évaluation des périphériques lors des tentatives de placement d'un composant.

A partir de l'interface de la figure 5.3, nous allons pouvoir agir sur le contexte d'exécution de l'application. Par exemple pour pouvons simuler la consommation d'énergie d'un périphérique en modifiant la valeur du champ Puissance batterie et provoquer une reconfiguration suite à un évènement de ressource (E_r) ou un évènement d'urgence (E_u).

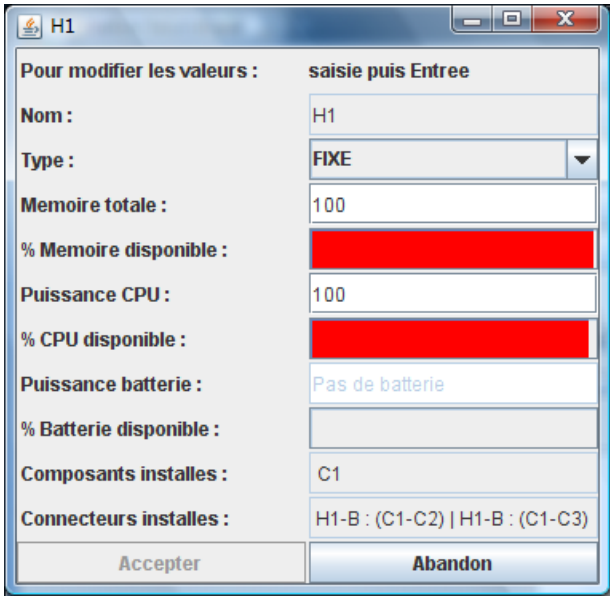

FIGURE 5.3 – Interface d'ajout, de modification et de consultation d'un périphérique

5.1.3 Ajouter, modifier et consulter une configuration

Cette interface (figure 5.4) permet de constituer une configuration que nous représentons sous la forme d'un graphe orienté. Pour cela, une configuration est identifiée par un nom (champ "service fourni"), un rôle (champ "famille") et une note de *QdS Utilité*, comprise en 0 et 1. L'interface propose une matrice où, pour chaque composant, nous indiquons par des cases à cocher les composants qui le suivent dans la configuration.

En conformité avec la carte d'identité des composants, nous pouvons imposer des contraintes de dépendance à un périphérique. Pour cela, une liste déroulante, à gauche du composant, permet de sélectionner le périphérique imposé.

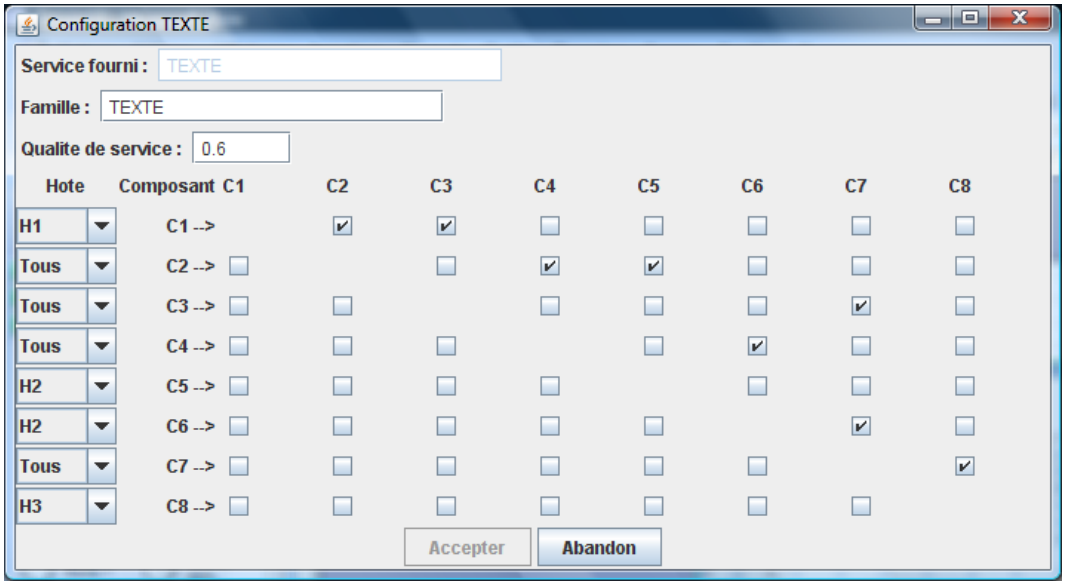

Figure 5.4 – Interface d'ajout, de modification et de consultation d'une configuration

5.1.4 Modifier une topologie réseau

L'interface de la figure 5.5 permet de simuler le réseau et la mobilité. Nous représentons le réseau par un graphe non orienté puisque les liaisons sont bidirectionnelles. Une matrice permet d'indiquer les voisins directs (un hop) de chaque périphérique. Pour cela, nous indiquons le débit disponible sur chaque liaison. Puisque le réseau est un graphe non orienté, lorsqu'une liaison $C_i - C_j$ est identifiée, le simulateur complète automatiquement la liaison $C_j - C_i$.

A partir de cette interface nous pourrons agir sur la mobilité des périphériques, et par association, sur celle des composants, pour simuler une reconfiguration en raison d'un évènement de mobilité (E_m).

Enfin, cette interface propose également un outil permettant de tester l'algorithme de recherche de routes utilisé par l'heuristique. Pour consulter les routes possibles entre deux périphériques, il suffit de les sélectionner au moyen des deux listes déroulantes puis de cliquer sur le bouton *Voir les routes*.

5.1.5 Lancer une simulation et afficher le résultat

Pour lancer une simulation, il suffit de choisir la configuration à tester puis de cliquer sur le bouton *Lancer*. Une fenêtre d'exécution de l'heuristique apparaît (figure 5.9), permettant de visualiser, en temps réel, les différentes tentatives de placement effectuées. Le titre de la fenêtre permet d'identifier si l'heuristique s'exécute en mode *intersection* (titre : exécution de l'heuristique en mode optimisation) ou en mode *union* (titre : exécution de l'heuristique en mode union des routes).

De plus, des cases à cocher permettent d'autoriser ou d'interdire l'écriture de traces à

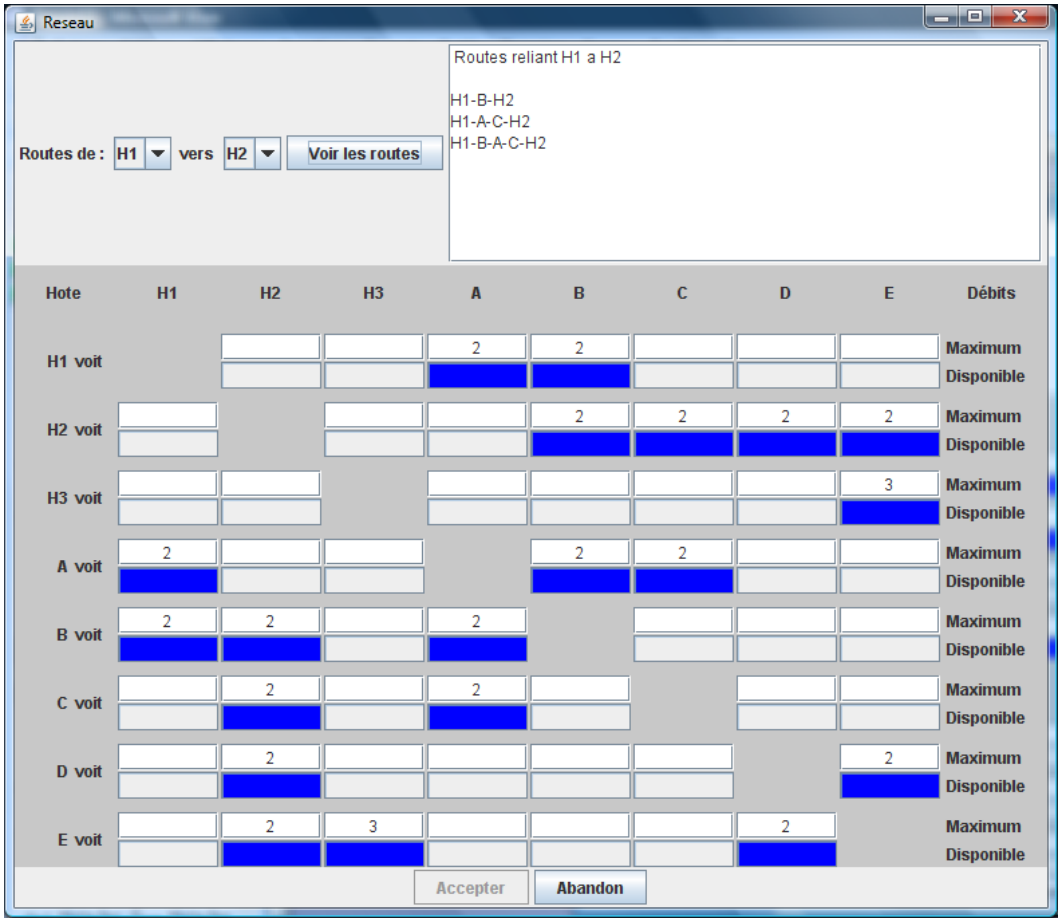

FIGURE 5.5 – Interface d'ajout, de modification et de consultation d'une topologie de réseau

chaque étape de l'heuristique dans la partie inférieure de la fenêtre principale. Ces traces permettent de voir comment l'heuristique a procédé et quels sont les tests qu'elle a effectués avant d'atteindre le résultat affiché. L'ensemble des traces est difficile à interpréter, c'est pourquoi il est intéressant de choisir de n'en afficher que certaines. Une étape intéressante est l'étape 5 qui montre toutes les tentatives de placement réalisées pour chaque composant.

Lorsque la simulation est terminée, un message s'affiche dans la partie inférieure (figure 5.1) indiquant si un placement a été trouvé ou si la simulation a échoué. Si la simulation a réussi, le placement trouvé s'affiche dans une fenêtre de résultat. Cette fenêtre indique le périphérique d'accueil de chaque composant, l'état de ses ressources physiques et les routes utilisées pour réaliser la configuration.

Il est alors possible de visualiser les connecteurs qui doivent être créés afin de mettre en place le déploiement (figure 5.6). Ils sont visibles d'une part sur les cartes d'identité des périphériques, puis grâce à la section Connecteurs de la fenêtre principale. Les connecteurs

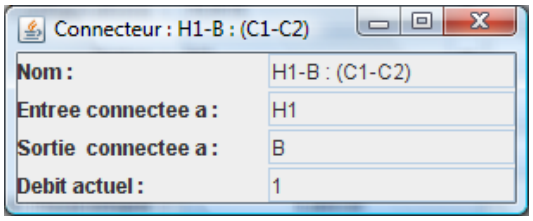

Figure 5.6 – Interface de consultation des connecteurs réseau

sont identifiés par le débit qu'ils transportent et par un nom de la forme

$$H_D - H_A : (C_S - C_E)$$

où :
- H_D est le périphérique de départ
- H_A est le périphérique d'arrivée
- C_S est le composant dont la sortie est connectée au connecteur
- C_A est le composant dont une entrée est connectée au connecteur

Enfin, à la fin de chaque simulation, un message affiche le nombre d'appels récursifs de l'heuristique pour atteindre le déploiement trouvé. Par hypothèse, plus il y aura de contraintes, plus le nombre d'appels récursifs sera élevé. En effet, l'heuristique teste toutes les solutions possibles avant de passer à une nouvelle configuration.

Il est possible de sauvegarder les paramètres d'une simulation dans un fichier. Pour cela il faut cliquer sur le bouton Sauvegarder, qui sérialise entièrement le graphe du réseau, les configurations définies, les composants et les périphériques dans un fichier *.kal* (pour kalimucho). La simulation pourra ainsi être chargée ultérieurement pour d'autres tests.

Les différentes fonctionnalités du simulateur permettent ainsi de simuler des changements au niveau du contexte d'exécution, principal acteur agissant sur la *QdS Pérennité*. Dans la section suivante, nous allons tester le fonctionnement de l'heuristique en simulant différentes contraintes de contexte comme la baisse du niveau d'énergie disponible puis celle de la charge CPU et enfin la mobilité.

5.2 Expérimentation

Nous allons à présent effectuer une série de tests permettant de simuler les fluctuations du contexte d'exécution d'une application. Nous illustrerons ces tests par une configuration appelée Texte (figure 5.7) dont les caractéristiques sont les suivantes :
- C_1 est placé sur H_1
- C_5 et C_6 sont placés sur H_2
- C_8 est placé sur H_3

Pour des raisons de facilité de manipulation, nous avons volontairement donné les mêmes caractéristiques à tous les composants à savoir :
- Mémoire consommée : 1
- CPU consommé : 3

– Energie consommée : 1
– Débit de sortie : 1

Les caractéristiques des hôtes sont décrites dans le tableau 12 :

Périphérique	Type	Mem	CPU	Energie
H_1	Fixe	100	100	Pas de batterie
H_2	CDC	70	70	50
H_3	CLDC	50	50	50
A	CDC	40	60	60
B	CLDC	20	40	20
C	CLDC	10	20	20
D	CLDC	20	20	20
E	CLDC	70	50	40

TABLE 12 – Caractéristiques des périphériques de la simulation

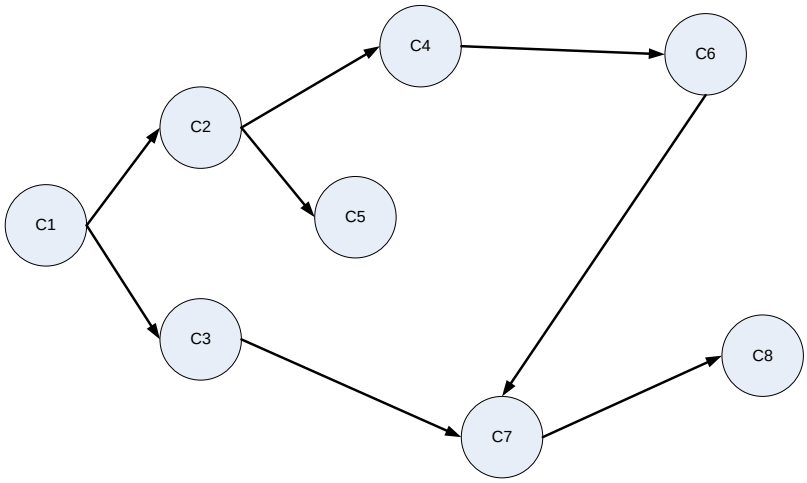

FIGURE 5.7 – Graphe de la configuration "Texte"

Le réseau défini pour ces tests est représenté par la figure 5.8.

La fonction de découverte de routes proposée par le prototype identifie les routes suivantes entre les périphériques imposés de la configuration "Texte" :

Routes reliant H_1 à H_2 :
– $H_1 - B - H_2$
– $H_1 - A - C - H_2$
– $H_1 - A - B - H_2$
– $H_1 - B - A - C - H_2$

Routes reliant H_1 à H_3
– $H_1 - B - H_2 - E - H_3$

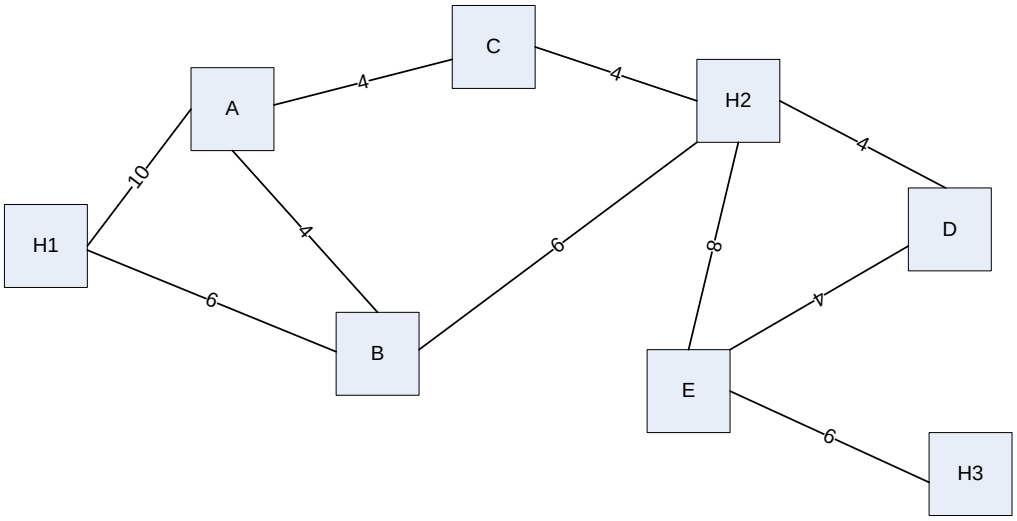

Figure 5.8 – Graphe de réseau de la simulation

- $H_1 - B - H_2 - D - E - H_3$
- $H_1 - A - C - H_2 - E - H_3$
- $H_1 - B - A - C - H_2 - E - H_3$
- $H_1 - A - C - H_2 - D - E - H_3$
- $H_1 - B - A - C - H_2 - D - E - H_3$

Routes reliant H_2 à H_3
- $H_2 - E - H_3$
- $H_2 - D - E - H_3$

L'heuristique commence toujours son exécution en utilisant l'intersection des routes pour étudier les placements des composants. Si aucun placement n'est trouvé en mode intersection, alors elle recommence en utilisant l'union des routes.

5.2.1 Test 1 : Conditions normales d'exécution

La configuration est tout d'abord évaluée dans un contexte d'exécution favorable afin de servir de base pour les tests suivants.

Dans cette configuration, l'intersection des routes $H_1 - H_2$, $H_1 - H_3$ et $H_2 - H_3$ est un unique périphérique : H_2. D'après nos réglages, H_2 dispose d'assez de ressources pour héberger l'ensemble des composants non encore placés. La solution proposée par l'heuristique est représentée sur la figure 5.9.

Le déploiement est trouvé en 5 appels récursifs qui sont les suivants :

1. étude du placement de C_2 sur H_2.

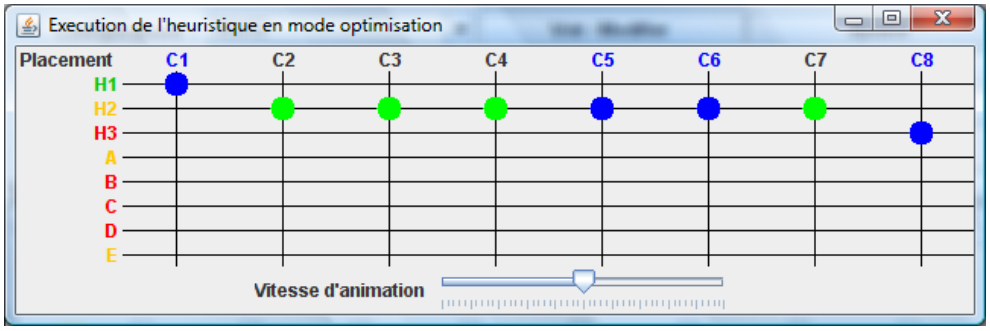

FIGURE 5.9 – Visualisation du résultat du test 1

2. étude du placement de C_3 sur H_2.
3. étude du placement de C_4 sur H_2.
4. étude du placement de C_7 sur H_2.
   L'évaluation des ressources de H_2 et du réseau respecte les critères de QdS Pérennité, les placements sont retenus.
5. tous les composants sont placés. Un déploiement a été trouvé.

La fenêtre de résultat de la figure 5.10 nous indique que cette solution utilise les routes $H_1 - B - H_2$ et $H_2 - E - H_3$.

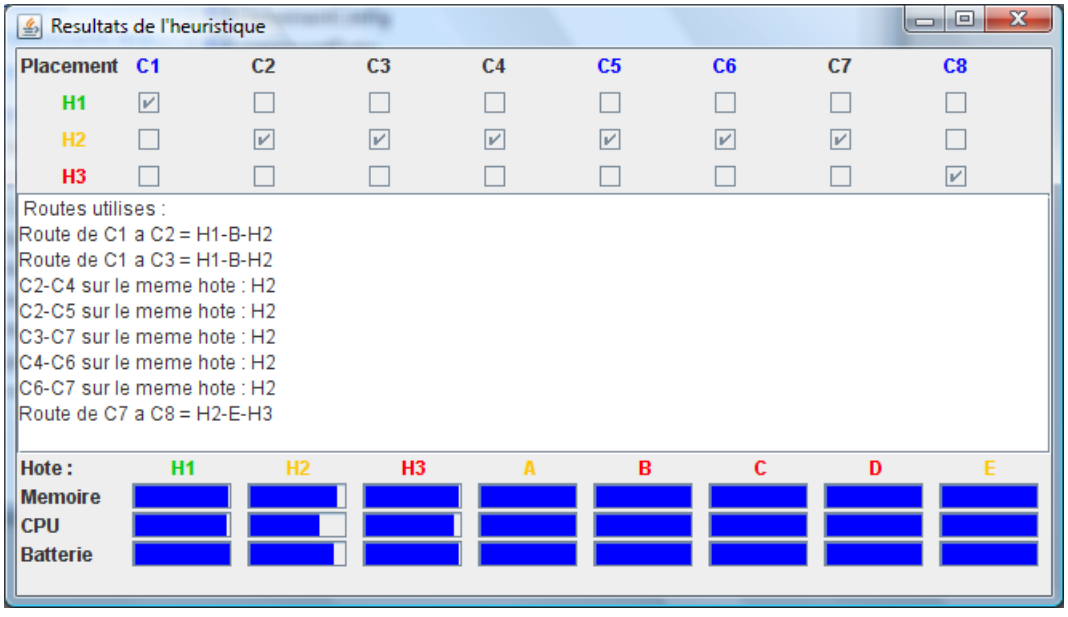

FIGURE 5.10 – Résultat du placement des composants de la configuration "Texte" dans les conditions d'exécution favorables

Sur la figure 5.11, nous pouvons remarquer que les routes utilisées (en pointillés) pour ce déploiement sont les routes les plus courtes entre H_1, H_2 et H_3.

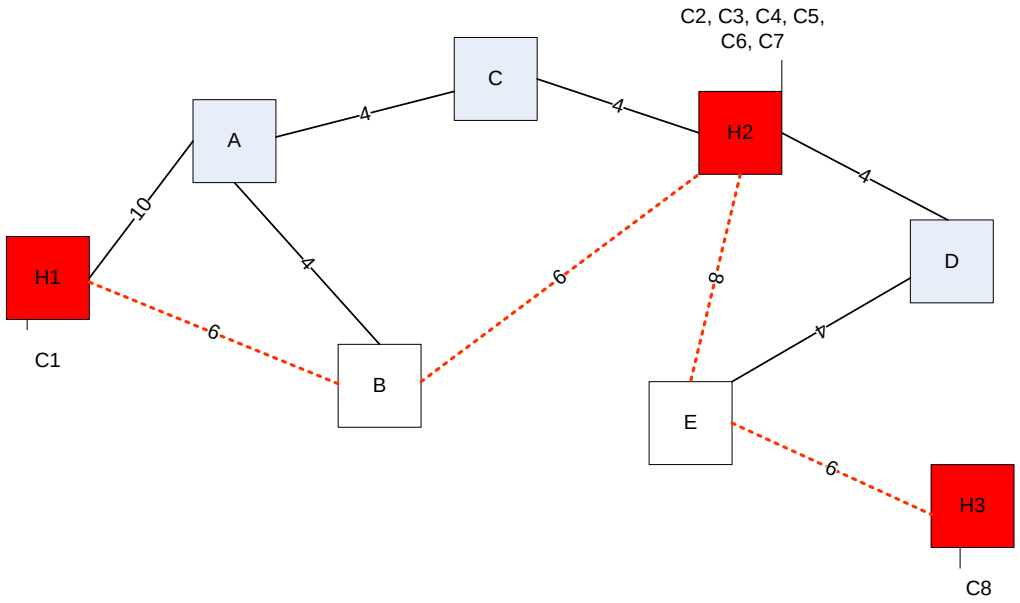

Figure 5.11 – Résultat du déploiement du test 1

5.2.2 Test 2 : Evènement de ressource sur H_2

Le deuxième test consiste à provoquer une migration suite à un évènement de ressources sur le périphérique H_2. Pour cela nous réduisons la puissance de la batterie de H_2 à la valeur 3. D'après les conditions initales de la configuration, il supporte déjà deux composants, C_5 et C_6, qui consomment chacun 1 de batterie. H_2 ne peut donc supporter qu'un seul composant supplémentaire. Parmi C_2, C_3, C_4 et C_7, un seul pourra être hébergé sur H_2, les autres doivent être migrés.

Puisque l'intersection des routes ne comprend que H_2 et que celui-ci ne peut accueillir qu'un seul composant supplémentaire, l'heuristique en mode *intersection* échoue. Elle recommence en mode *union* des routes. Après 7 appels récursifs, elle trouve le déploiement suivant :

- C_1 placé sur : H_1
- C_2 placé sur : H_1
- C_3 placé sur : H_1
- C_4 placé sur : H_1
- C_5 placé sur : H_2
- C_6 placé sur : H_2
- C_7 placé sur : H_1
- C_8 placé sur : H_3

Pour soulager H_2, tous les composants qui étaient sur H_2 dans le test 1, ont été migrés sur H_1 (figure 5.12), qui dispose de plus de ressources.

Nous pouvons remarquer que contrairement au test 1 où l'intersection apporte une

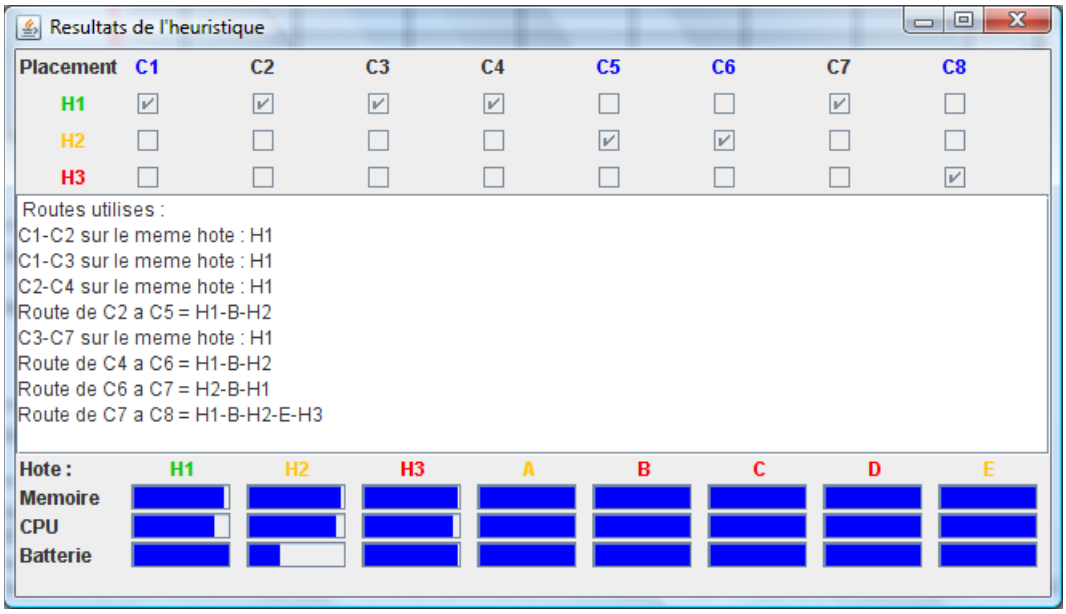

FIGURE 5.12 – Interface du résultat du déploiement du test 2 : Evènement de ressource sur H_2

solution ne comprenant que 6 connecteurs, ici, l'utilisation de l'union propose une solution qui comporte 10 connecteurs. De plus, cette solution " fait un détour " par H_1 pour relier $C_6 - C_7 - C_8$ (connecteurs $H_1 - B$, $B - H_2$, $H_2 - E$ et $E - H_3$) (figure 5.13).

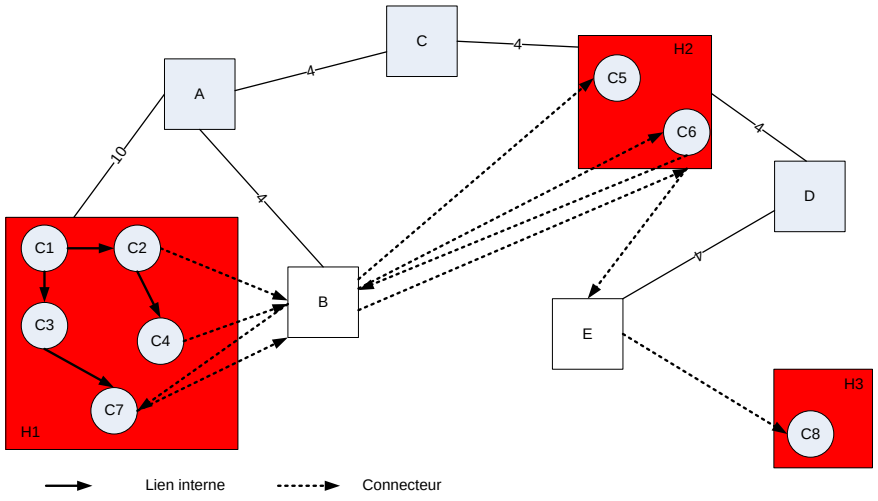

FIGURE 5.13 – Résultat du déploiement du test 2

5.2.3 Test 3 : Evènement de ressource sur H_1

Le troisième test consiste à provoquer une reconfiguration suite à un évènement de ressources sur H_1. Ce test permet de voir quelles sont les solutions que l'heuristique trouve lorsque les périphériques imposés sont contraints.

Par hypothèse, tous les composants ont une valeur de CPU consommé égale à 3. Nous modifions par conséquent la valeur de puissance CPU de H_1 afin qu'il ne puisse supporter que deux composants au maximum. Nous lui donnons la valeur 8.

A présent, les contraintes sont telles que H_1 peut supporter au maximum 2 composants et H_2 peut en supporter 3.

Tout comme pour le test 2, l'heuristique ne trouve pas de solution en mode *intersection* et propose une solution en mode *union*.

Voici un extrait du déroulement de l'évaluation :

```
*** Début de l'étape 4 ***
composant a placer : C2
Résultat étape 4 = Composant retenu : C2
    avec la liste classée d'hôtes :
        {H2}

*** Début de l'étape 5 ***
Etude du placement de C2 sur H2

*** Début de l'étape 4 ***
composant a placer : C3
Résultat étape 4 = Composant retenu : C3
    avec la liste classée d'hôtes :
        {H2}

*** Début de l'étape 5 ***
Etude du placement de C3 sur H2
Tentative de placement de C3 sur H2 -->
                                manque de ressources sur l'hôte
Tentative de placement de C2 sur H2 ne donne pas de solution ==>
                                autre essai pour C2


************** FIN DE L'HEURISTIQUE ****************************
Aucun placement n'a été trouvé en version avec intersection des
routes

On essaye avec la version de l'heuristique avec union des routes

*** Début de l'étape 4 ***
composant a placer : C3
*** Début de l'étape 4 ***
composant a placer : C2
Résultat étape 4 = Composant retenu : C2
    avec la liste classée d'hôtes :
        {H1, H2, H3, A, E, B, C, D}
```

```
*** Début de l'étape 5 ***
Etude du placement de C2 sur H1

*** Début de l'étape 4 ***
composant a placer : C3
Résultat étape 4 = Composant retenu : C3
    avec la liste classée d'hôtes :
        {H1, H2, H3, A, E, B, C, D}

*** Début de l'étape 5 ***
Etude du placement de C3 sur H1
Tentative de placement de C3 sur H1 -->
                                manque de ressources sur l'hôte
Etude du placement de C3 sur H2

*** Début de l'étape 4 ***
composant a placer : C4
Résultat étape 4 = Composant retenu : C4
    avec la liste classée d'hôtes :
        {H1, H2, H3, A, E, B, C, D}

*** Début de l'étape 5 ***
Etude du placement de C4 sur H1
Tentative de placement de C4 sur H1 -->
                                manque de ressources sur l'hôte
Etude du placement de C4 sur H2
Tentative de placement de C4 sur H2 -->
                                manque de ressources sur l'hôte
Etude du placement de C4 sur H3

*** Début de l'étape 4 ***
composant a placer : C7
Résultat étape 4 = Composant retenu : C7
    avec la liste classée d'hôtes :
        {H1, H2, H3, A, E, B, C, D}

*** Début de l'étape 5 ***
Etude du placement de C7 sur H1
Tentative de placement de C7 sur H1 -->
                                manque de ressources sur l'hôte
Etude du placement de C7 sur H2
Tentative de placement de C7 sur H2 -->
                                manque de ressources sur l'hôte
Etude du placement de C7 sur H3
```

Nous pouvons remarquer dans ce test l'utilisation du back-tracking lors du premier essai de l'heuristique en mode intersection. Malgré ce retour en arrière, il n'existe pas d'autre périphérique possible dans l'intersection, l'heuristique a donc échoué puis recommencé en mode union.

Comme prévu dans l'hypothèse, nous pouvons voir sur la figure 5.14 que le périphérique

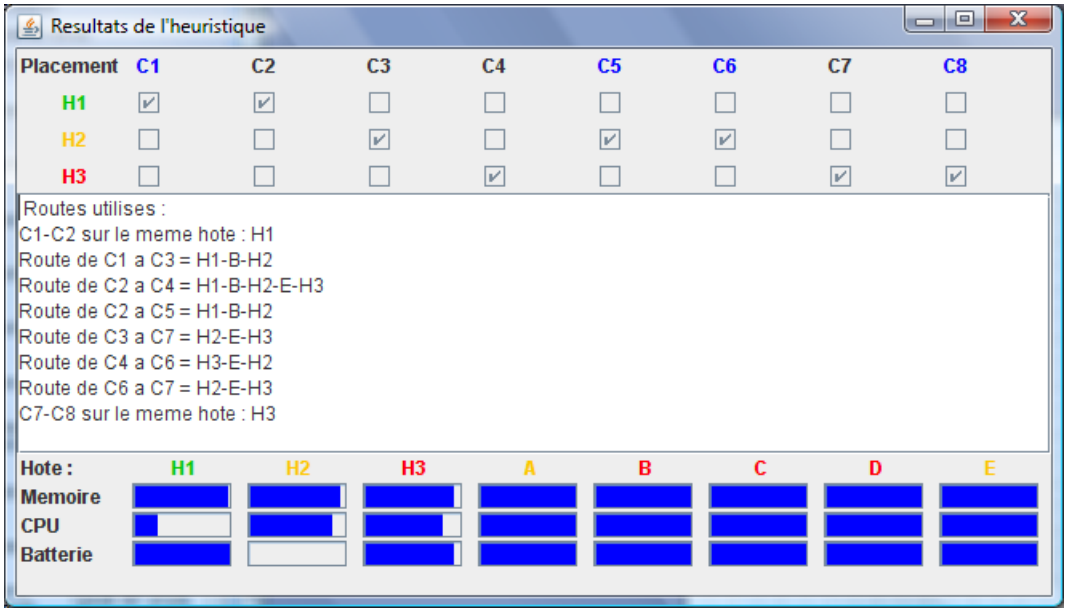

FIGURE 5.14 – Interface du résultat du déploiement du test 3 : Evènement de ressource sur H_1

H_1 héberge seulement deux composants, C_1 et C_2, et H_2 héberge trois composants, C_3, C_5 et C_6. C_4 et C_7 sont logiquement placés sur H_3 puisque l'heuristique tente tout d'abord d'utiliser les périphériques imposés avant d'utiliser d'autres périphériques.

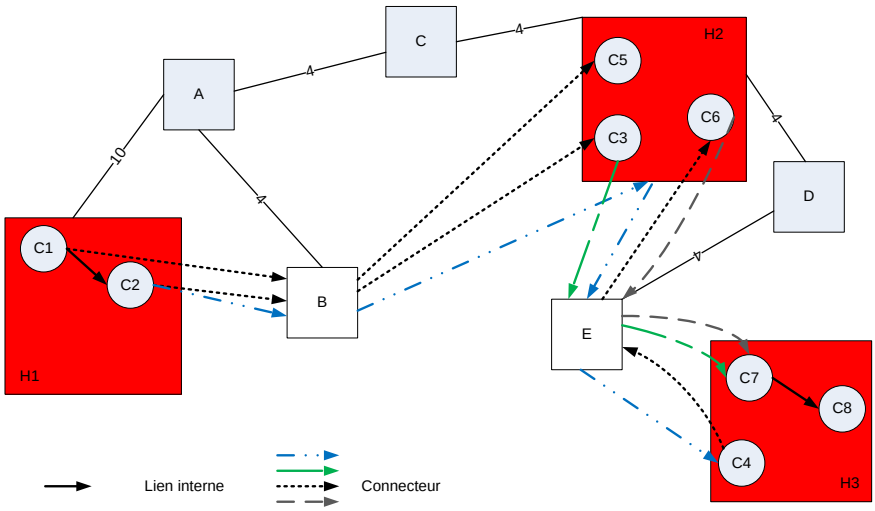

FIGURE 5.15 – Résultat du déploiement du test 3

Malgré la contrainte supplémentaire sur H_1, le déploiement a été trouvé en 7 tentatives (identique au test 2).

Nous pouvons faire la même remarque que pour le test 2 concernant le manque d'optimisation de la solution. En effet, cette solution place C_4 sur H_3 et implique de faire un aller-retour entre H_2 et H_3 pour relier $C_4 - C_6 - C_7 - C_8$ alors que placer C_4 sur B économiserait deux connecteurs (figure 5.15).

Ce test confirme notre hypothèse que l'heuristique ne permet pas d'obtenir un déploiement optimal mais uniquement un déploiement de qualité suffisante.

5.2.4 Test 4 : Evènement de mobilité sur H_3

Le quatrième test consiste à vérifier que malgré un changement de topologie, la solution trouvée par l'heuristique propose un nombre minimum de modifications en termes de connecteurs et de migration de composants.

Pour cela nous déplaçons H_3 afin qu'il ne voie plus E mais D.

Le déploiement reste inchangé par rapport au test 3, et conformément à notre hypothèse, les connexions qui utilisaient E, utilisent maintenant D comme relais (figure 5.16).

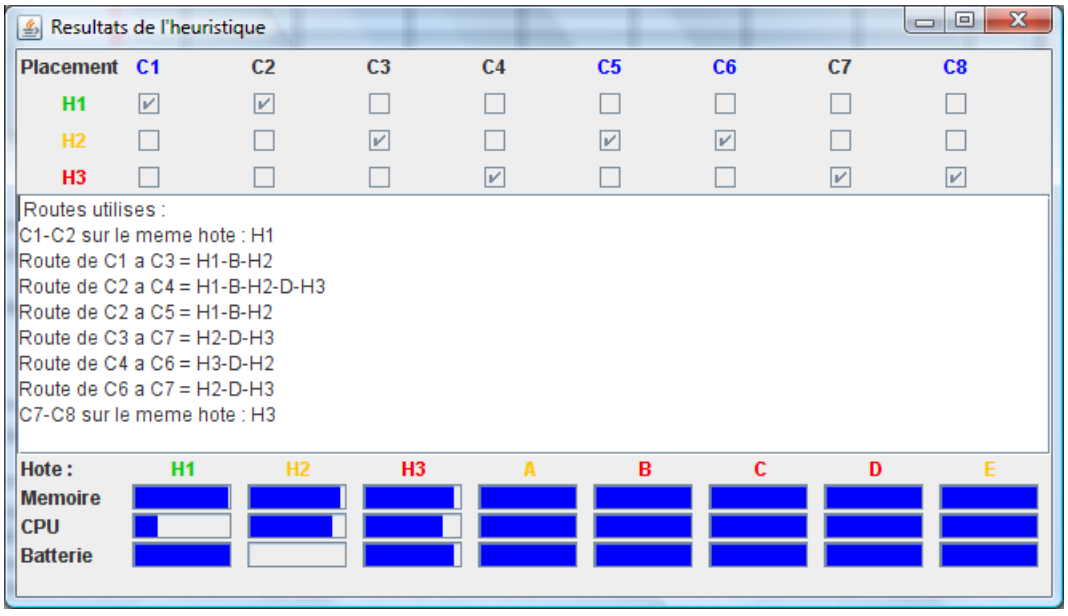

FIGURE 5.16 – Interface du résultat du déploiement du test 4 : Evènement de mobilité sur H_3

En effet, E n'étant plus en liaison directe avec H_3, selon la définition du poids d'un périphérique, son poids est devenu supérieur à celui de D, donc D a été testé en priorité.

5.2.5 Test 5 : Cas d'un environnement très contraint

Le cinquième test consiste à montrer l'importance de la prise en compte du réseau dans la recherche d'un déploiement. En effet, comme les liaisons réseau sont bidirectionnelles, lors de l'utilisation de l'application, la charge réseau peut impliquer des difficultés de

circulation de l'information à double sens entre les périphériques. Il faut alors pouvoir trouver une autre liaison afin d'assurer la continuité de service et une QdS suffisante.

Dans ce test, nous réduisons volontairement les débits et les ressources des périphériques afin de simuler un environnement très contraint. Aucun périphérique, mis à part H_2, ne peut héberger plus d'un composant. Nous réduisons également les débits sur le graphe de réseau afin de ne pouvoir faire passer qu'un ou deux connecteurs par liaison réseau. Nous obtenons le déploiement suivant (figure 5.17) :

- C_1 est placé sur H_1
- C_2 est placé sur A
- C_3 est placé sur B
- C_4 est placé sur C
- C_5 et C_6 sont placés sur H_2
- C_7 est placé sur D
- C_8 est placé sur H_3

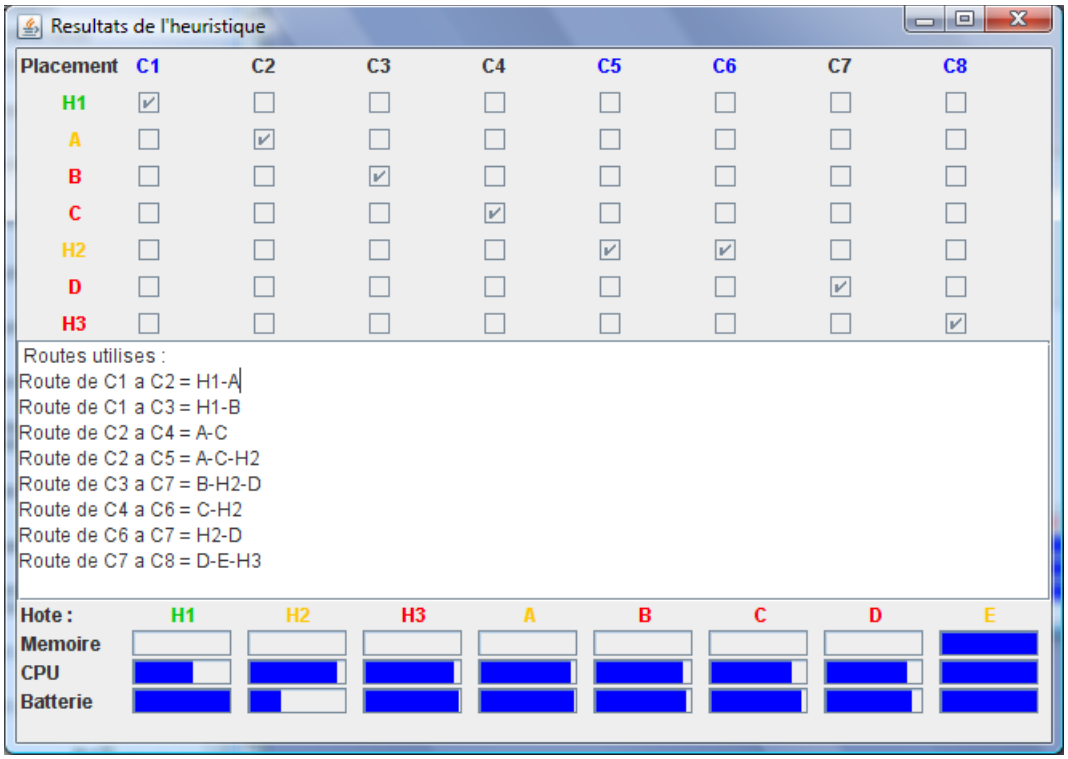

Figure 5.17 – Interface du résultat du déploiement du test 5 : Environnement très contraint

Sur la figure 5.18, illustrant le résultat du test 5, nous pouvons remarquer que l'heuristique a respecté le principe de minimisation des liens réseau, puisqu'il existe au maximum deux liens entre deux composants qui se suivent dans la configuration.

Le cas précédent présente une configuration peu complexe et un réseau très maillé, ce qui permet d'avoir un grand nombre de routes à disposition pour des tentatives de placement. Cependant, bien que nous ayons réduit les ressources des périphériques et les

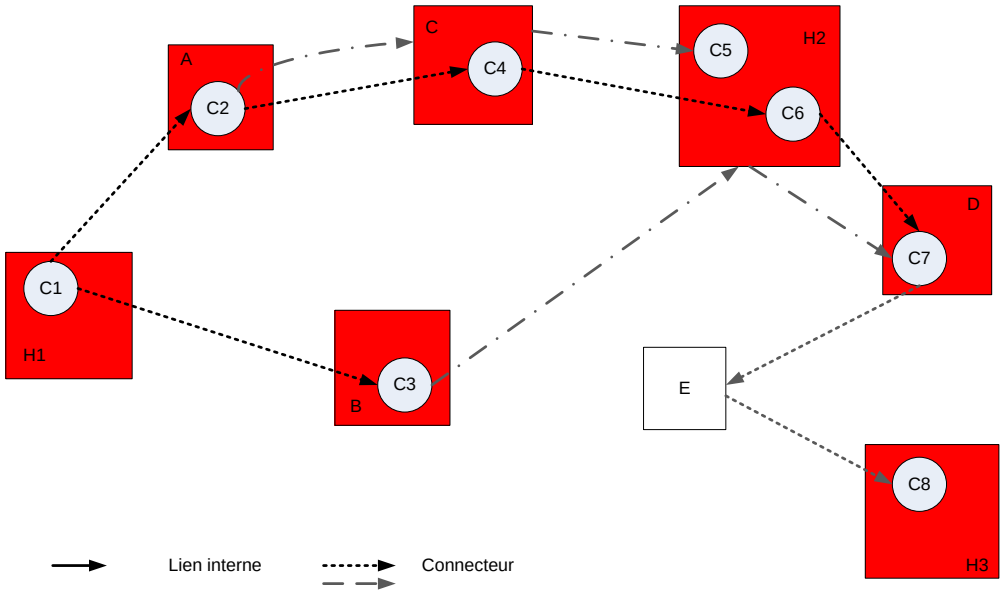

FIGURE 5.18 – Résultat du test 5

débits du réseau, cet exemple ne reflète pas la complexité des réseaux mobiles. Afin de montrer cette complexité, nous effectuons un autre test à l'aide de la configuration de la figure 5.19 et du réseau de la figure 5.20.

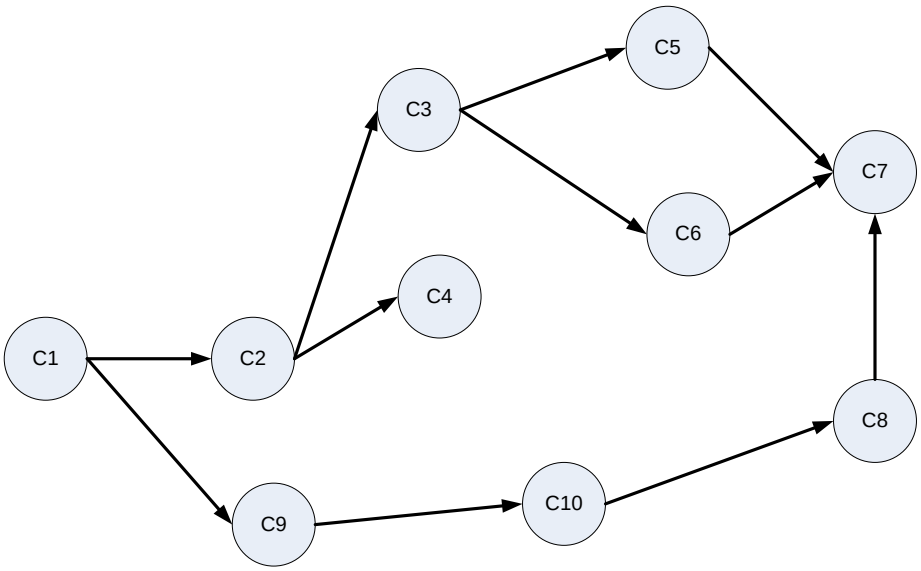

FIGURE 5.19 – Graphe d'une configuration complexe

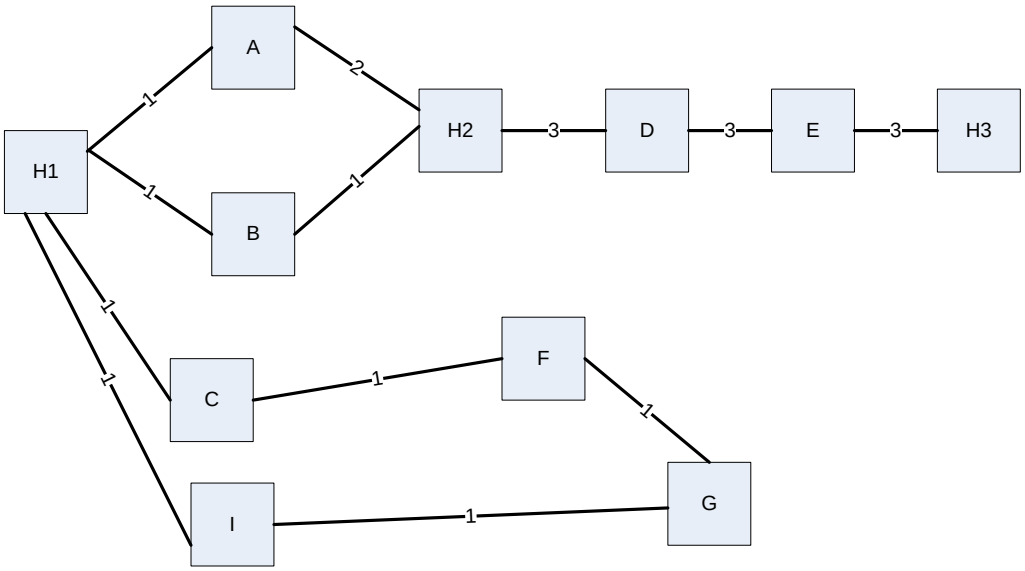

Figure 5.20 – Graphe d'un réseau constraint

Le placement de départ est le suivant :
- C_1 est placé sur H_1
- C_2 et C_4 sont placés sur H_2
- C_7 est placé sur H_3
- C_8 est placé sur G

Au bout de 222 tentatives, l'heuristique obtient un résultat qui respecte tous les principes que nous avons définis jusqu'à présent. En effet, afin de minimiser les liaisons réseau, les composants sont placés de façon à être le plus proche possible de leurs suivants (figure 5.21).

De plus, nous pouvons remarquer l'importance de considérer la configuration comme un graphe non orienté lors de la recherche des chemins entre périphériques imposés afin d'élargir le nombre de solutions à tester. Un exemple est le cas de la liaison entre le composant C8 et le composant C7, illustrée dans la figure 5.22. Bien qu'augmentant de façon considérable le nombre de tentatives (222 contre moins de 20 pour les tests précédents), ce principe permet de trouver une solution satisfaisante et de garantir la continuité de service même dans des cas très contraints comme celui-ci.

Ce test montre comme il est parfois difficile d'établir des connexions entre des périphériques très contraints tels que les périphériques mobiles. Il montre l'importance de la prise en compte du réseau lors des reconfigurations afin de pouvoir garantir une continuité de service et une QdS suffisante.

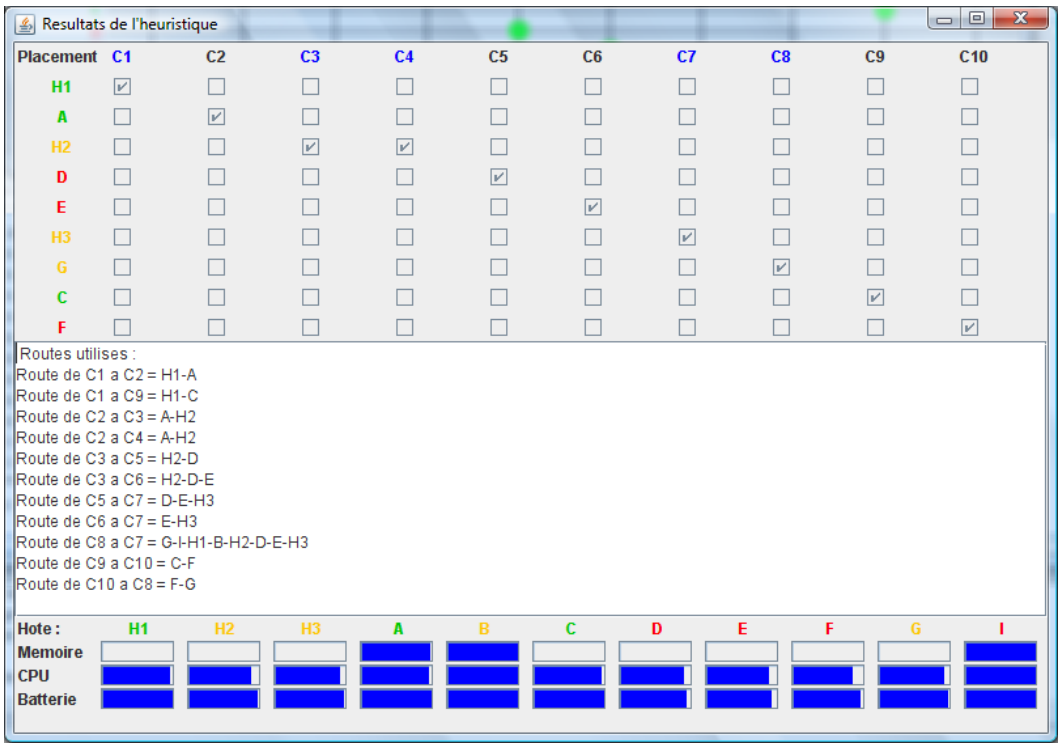

FIGURE 5.21 – Interface du résultat du déploiement d'une configuration complexe en environnement très contraint

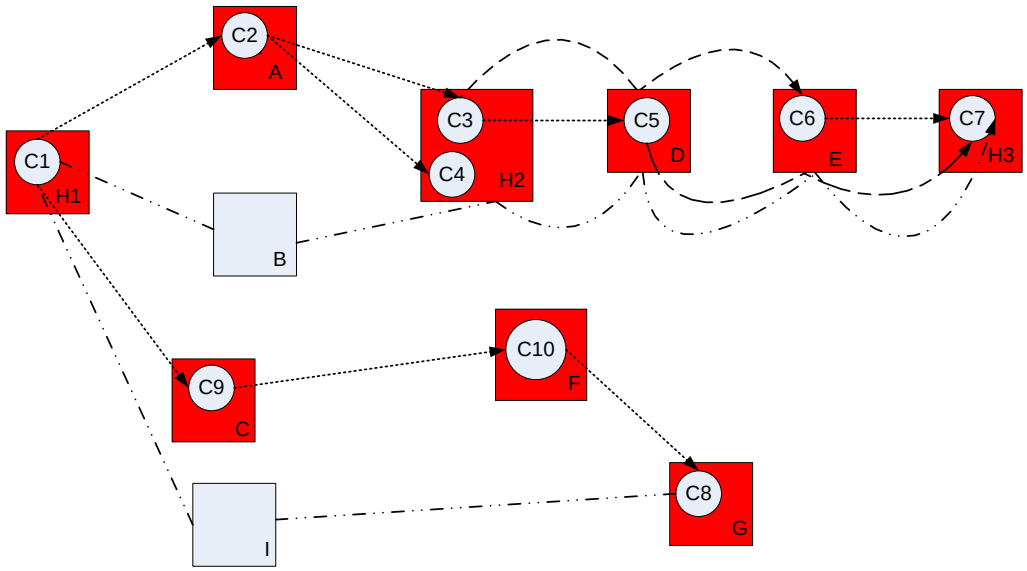

FIGURE 5.22 – Résultat du déploiement d'une configuration complexe en environnement très contraint

5.3 Conclusion

L'optimisation de la qualité de service dans les applications distribuées se heurte au problème de la complexité algorithmique d'une telle tâche.

Face à ce problème, nous avons proposé une heuristique de choix d'une configuration à déployer permettant d'obtenir rapidement un déploiement satisfaisant les critères de QdS. Cette heuristique se concentre particulièrement sur les critères de *QdS Pérennité*. Elle permet d'obtenir un déploiement contextuel prenant en compte à la fois les contraintes de ressources physiques et réseau. Elle repose sur une approche récursive guidée par la minimisation des liaisons réseaux et le souci de préserver les ressources des périphériques.

L'heuristique débute sa recherche par une tentative de placement parmi les périphériques à l'intersection des routes entre composants non-déplaçables. Les tests ont montré que, lorsqu'une solution existe, elle utilise un faible nombre de liaisons réseau, principales responsables de la consommation d'énergie des périphériques mobiles. Si aucune solution n'est possible, elle recherche alors des solutions parmi les périphériques de l'union des routes.

L'efficacité de l'heuristique repose également sur le classement des périphériques candidats à un placement et sur le principe du back-tracking. Le classement, tout comme le back-tracking, permettent de réduire le nombre de connexions réseau en essayant de regrouper les composants sur un même périphérique, ou à défaut, sur un périphérique situé à un hop du composant suivant ou précédent et d'éviter les " détours " sur le réseau.

Nous avons pu vérifier également que l'heuristique obtient une solution satisfaisante en un nombre de tentatives raisonnables dans les cas d'utilisation peu contraints.

En revanche ce prototype ne permet pas de valider le principe de la migration, telle que nous l'avons définie, lors des reconfigurations. Pour cela, nous devrions améliorer l'heuristique en ajoutant un critère supplémentaire de classement qui prend en compte la position précédente du composant dans le réseau.

Enfin, nous avons pu remarquer l'importance de considérer la configuration comme un graphe non orienté lors de la recherche des chemins entre périphériques imposés afin d'élargir le nombre de solutions à tester. Bien qu'augmentant de façon considérable le nombre de tentatives, ce principe permet de trouver une solution satisfaisante et de garantir la continuité de service même dans des cas très contraints. Il montre l'importance de la prise en compte du réseau lors des reconfigurations afin de pouvoir garantir une continuité de service et une QdS suffisante.

5.4 Prototype n° 2 : implémentation de Kalimucho

Ce prototype est une mise en œuvre de Kalimucho dans un environnement constitué d'un ordinateur fixe, d'un ordinateur portable, d'un PDA et de capteurs Sun Spot (figure 5.23). Cette implémentation de Kalimucho permet de déployer et de reconfigurer des applications basées composants à partir de fichiers de commandes. Elle permet également de capturer les éléments de contexte nécessaires à la prise de décision. En revanche, actuellement, cette implémentation n'intègre pas le *générateur de reconfiguration* dans sa

totalité. En effet, ce dernier comporte uniquement les fichiers de description des configurations et les fichiers de commandes de reconfiguration. L'heuristique du prototype n° 1 n'a pas encore été intégré. Actuellement, le gestionnaire de contexte rapporte, sous forme d'une alarme, les changements de contexte et l'utilisateur choisit l'adaptation que la plate-forme doit appliquer. Pour cela, le *gestionnaire de contexte* est capable de :

- Lire l'état d'un hôte (mémoire, CPU, batterie)
- Lire l'état d'un composant ou d'un connecteur (QdS, activité, connexions, ...)
- Relayer des états vers d'autres plate-formes
- Recevoir des alarmes de l'infrastructure

Le *déployeur* quant à lui, est capable de :

- Créer des composants
- Créer des connecteurs
- Supprimer des composants
- Supprimer des connecteurs
- Migrer des composants
- Connecter/Déconnecter une entrée de composant
- Supprimer/Dupliquer une sortie de composant
- Envoyer des commandes à d'autres plate-formes

Comme décrit dans la section 4.6, Kalimucho est distribuée sur l'ensemble des périphériques participants à l'application. Chaque implémentation est spécifique au type de périphérique. Ainsi nous trouvons dans notre prototype :

- Kalimucho pour les périphériques fixes tels que les PC et les ordinateurs portables ;
- Kalimucho pour les périphériques CDC tels que les PDA ;
- Kalimucho pour les périphériques CLDC tels que les capteurs Sun Spot.

De plus, ces périphériques n'utilisent pas tous le même réseau de communication. La figure 5.23 représente tous les réseaux utilisés par notre prototype. Nous pouvons constater que le PDA et l'ordinateur portable communiquent via le réseau WIFI, que le PC et l'ordinateur portable sont reliés par un câble et que les capteurs communiquement via le réseau Zigbee. Pour pouvoir communiquer avec le reste des périphériques, une station base reliée à un ordinateur permet de faire le relais entre le réseau Zigbee et le réseau filaire ou Wifi. Ce relais est également équipé de Kalimucho.

Dans ce prototype, nous testons le service *Superviseur* de Kalimucho. Pour cela, nous proposons un scénario permettant d'effectuer toutes les actions de la plate-forme :

- Déployer une configuration
- Remplacer un composant
- Ajouter un composant
- Migrer un composant
- Reconfigurer un service

Une application d'affichage de l'inclinaison d'un capteur a été développée. Nous déployons initialement, sur deux capteurs Sun Spot, une configuration composée de trois composants : un composant d'acquisition de l'inclinaison, un composant de traitement permettant de mettre la valeur capturée dans un objet de type « vecteur », et un composant d'affichage (figure 5.24).

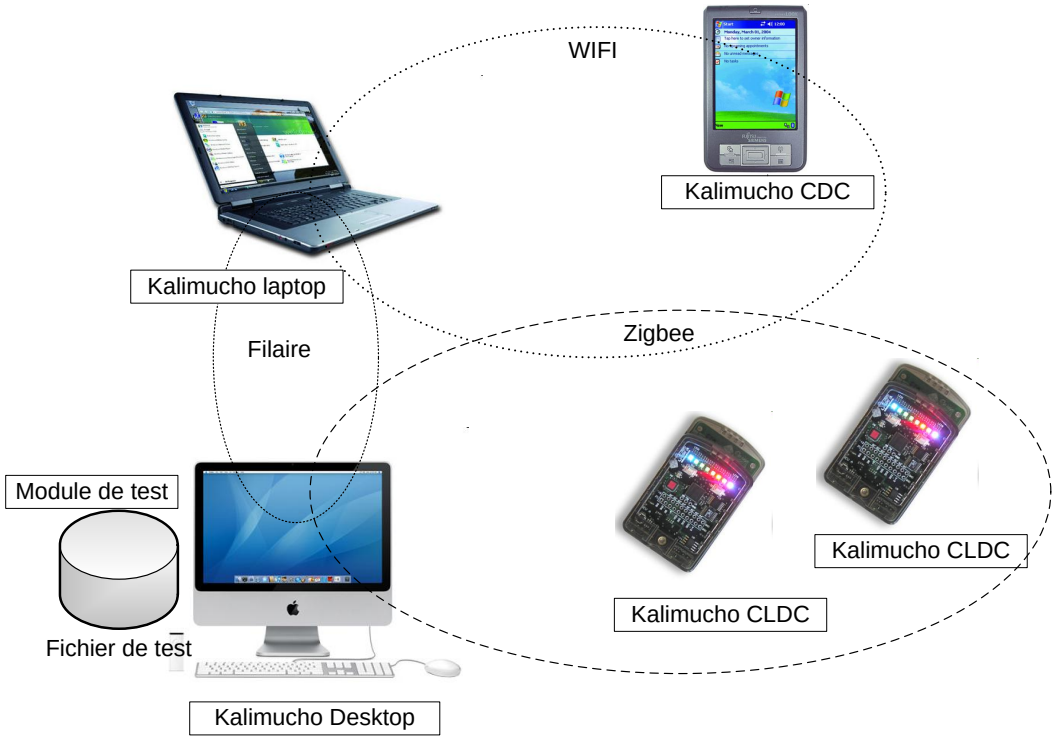

Figure 5.23 – Déploiement de Kalimucho dans un environnement hétérogène

Pour cette application, il n'y a qu'un seul service. Par conséquent il n'y a pas d'élément de contexte d'utilisateur à surveiller. De la même façon, pour des raisons de facilité d'expérimentation, nous n'avons spécifiée aucune contraintes d'utilisation. Les éléments de contexte à surveiller sont donc les éléments du contexte d'exécution à savoir le débit réseau, le niveau des ressources matérielles et les ressources logicielles.

La capture du contexte est assurée de la manière suivante :

- A partir de l'application, les conteneurs de composants métiers capturent l'état de chaque composant qu'ils encapsulent. Ainsi ils surveillent :
  - l'activité du composant (cycle de vie)
  - l'état des connexions (entrées/sorties)
  - la valeur personnalisée de QdS
- A partir de l'infrastructure, les conteneurs de connecteurs capturent l'état de chaque connecteur qu'ils encapsulent. Ils surveillent le niveau des buffers d'entrée et de sortie des connecteurs et déclenchent une alarme, transmise à la plate-forme, lorsque le niveau est bas (famine) ou haut (saturation). Les hôtes sont également surveillés afin de capturer l'évolution du niveau de leurs ressources matérielles : mémoire utilisée, batterie, charge CPU. Des alarmes sont déclenchées lorsque le niveau des ressources atteint les seuils prédéfinis lors de la conception.

Le capteur A supporte le composant d'acquisition et le composant de traitement, tandis que le capteur B supporte le composant d'affichage par leds. Nous remplaçons ensuite le

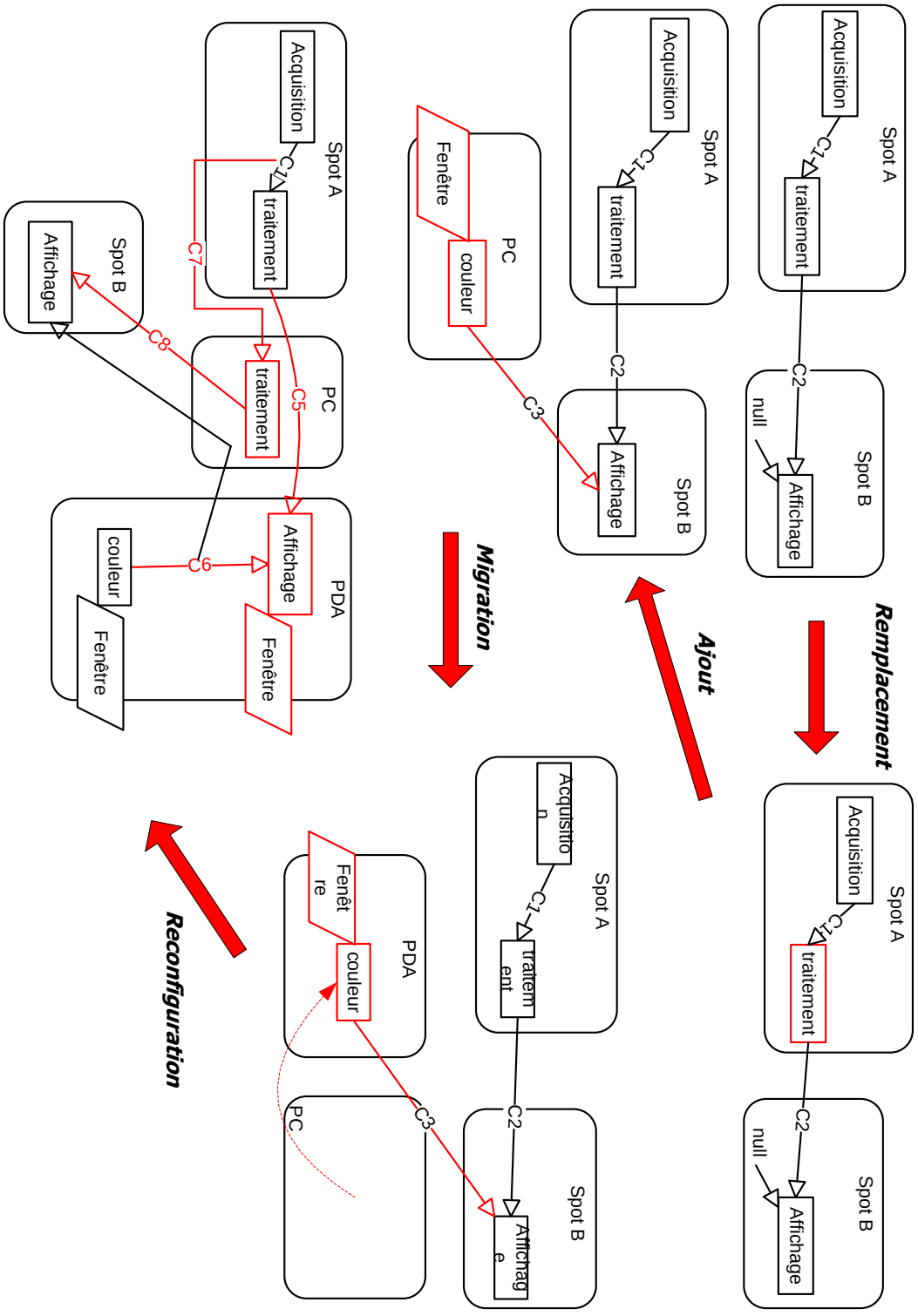

FIGURE 5.24 – Scénario de reconfiguration d'un service d'affichage

composant de traitement par un composant qui inverse la valeur de l'inclinaison. Puis nous ajoutons un composant permettant de choisir la couleur des leds qui affichent l'inclinaison. Puisque le capteur ne disposent pas d'interface utilisateur, ce composant est déployé sur un PC, ce qui ajoute un périphérique au réseau. L'utilisateur souhaitant se déplacer, il décide d'utiliser un PDA et non plus le PC. Nous migrons alors le composant couleur vers le PDA. La migration permet de déplacer un composant avec son contexte ce qui évite la perte de données lors des reconfigurations. Nous pouvons constater que le connecter $C3$ reliant les composants couleur et affichage utilise le PC en tant que relais. En effet, le PDA et les capteurs n'utilisent pas le même réseau, le PC permet de faire le lien entre les deux grâce à la station base qui y est reliée. Enfin, nous reconfigurons l'application afin que le PDA affiche également l'inclinaison. Pour cela nous dupliquons le flux en sortie du composant couleur afin que les deux composants affichage reçoivent l'information. Le listing 5.3 illustre un exemple de commandes de reconfigurations.

```
# names et addresses of the hosts
spotA   0014.4F01.0000.0AE0
spotB   0014.4F01.0000.0A6C
PC 192.168.000.027
PDA 192.168.000.030

# First deployment:
#(          spotA           )(        spotB          )
# (acquisition) -[c1]- (treatment1) -[c2]-------> (display)
spotA CreateComponent acquisition application.TiltAcquisition [null]
 [c1]
spotA CreateComponent treatment application.Treatment1 [c1] [c2]
spotB CreateComponent display application.TiltColorDisplayOnSunspot
 [c2 not_used]  null
spotA CreateConnector c1 internal internal
spotA CreateConnector c2 internal spotB
...
# Third reconfiguration: move color selection component to the PDA
# (    spotA          )(       PC                )(     SpotB   )
#  (acquisition) -[c20]-> treatment2) -[c3]-> (display)
# (    PDA         )(       PC         )(      spotB      )
#     color -------------->[c1]------------> (display)
PC RemoveComponent color
spotB RemoveConnector c4
PDA CreateComponent color application.ColorSelector [null] c1
PC CreateConnector c1 PDA spotB
spotB ReconnectInputComponent  display   1  c1
...
```

Listing 5.3 – Exemple de commandes de reconfiguration

Le paragraphe suivant présente les expérimentations que nous avons menées à partir de ce scénario.

5.5 Expérimentations

L'objectif de cette expérimentation consiste d'une part à mesurer les performances de Kalimucho en termes de temps d'exécution lors d'une reconfiguration et, d'autre part, à montrer la limite de notre plate-forme lors des transferts de données, puisque l'originalité de notre proposition réside en partie dans la minimisation des communications réseau.

5.5.1 Test d'exécution d'une commande de reconfiguration sur un capteur

Ce premier test consiste à mesurer les performances de la plate-forme en termes de temps. Nous avons mesuré le temps d'exécution de toutes les commandes que la plate-forme utilise pour reconfigurer une application telles que la création et la suppression de composant et de conteneurs.

Nous avons également mesuré le temps d'exécution des commandes nécessaires à la prise en compte du contexte dans les reconfigurations. Ce sont les commandes de lecture de QdS et de lecture d'état des composants et des périphériques. Le tableau 13 présente un récapitulatif des mesures effectuées.

Commande	**Durée d'exécution en ms**
Création d'un composant	70 à 170
Suppression d'un composant	minimum 20 ms, lié au temps nécessaire au CM pour se terminer (limité par paramétrage de la PF)
Création d'un conteneur	Interne : 70 à 110 Réparti : 100 à 190 sur l'hôte qui reçoit la commande, 30 à 120 sur l'autre hôte
Supression d'un conteneur	Interne : 60 à 80 Réparti : 100 à 260 sur l'hôte qui reçoit la commande, 30 à 120 sur l'autre hôte
Déconnexion ou reconnexion d'une entrée	20 à 60
Duplication d'une sortie	20 à 80
Lecture de la QdS d'un conteneur	80
Lecture de l'état d'un conteneur	70 à 80
Lecture de l'état d'un hôte	70 à 90

TABLE 13 – Temps d'exécution d'une commande de déploiement/reconfiguration sur un capteur SunSpot

Nous pouvons constater que toutes les mesures sont de l'ordre de la milliseconde, ce qui est acceptable pour un périphérique très contraint tel qu'un capteur. Concernant la création et la suppression de conteneur, nous remarquons que les temps d'exécution sont plus longs lorsqu'il s'agit d'un conteneur réparti. Ceci vient du fait qu'il est nécessaire

d'établir une connexion réseau pour chaque communication. A partir de cette constatation, nous avons procédé à un deuxième test afin de déterminer le temps de transfert de données entre deux périphériques CLDC.

5.5.2 Test de transfert de données entre capteurs

Ce test consiste à déterminer le temps de transfert de données entre deux capteurs et de déterminer la limite de charge réseau permettant aux périphériques de communiquer.

Pour cela, nous constituons des boucles de communication (figure 5.25) : un composant CM1 envoie des données à un composant CM2 qui en envoie à CM1 à son tour. Nous mesurons alors la durée d'un tour de boucle. Au fur et à mesure du test, nous ajoutons une nouvelle boucle, jusqu'à 8 boucles maximum, afin de charger le CPU du périphérique et le réseau.

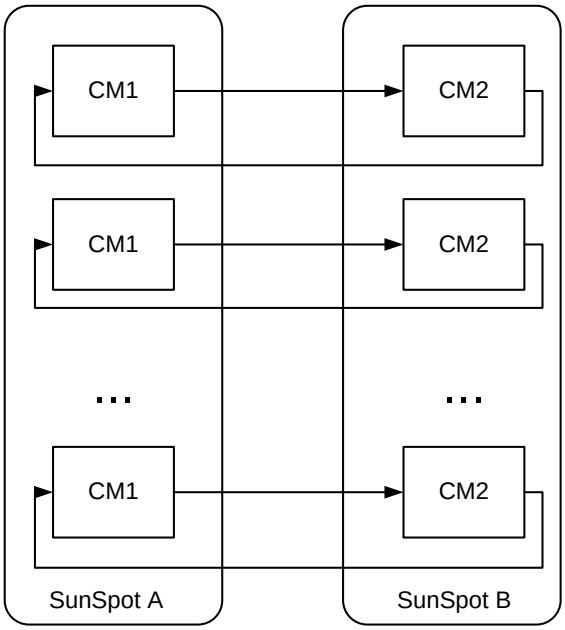

Figure 5.25 – Procédure de test de transfert de données

Nous effectuons deux séries de mesures avec deux tailles de données différentes permettant de refléter au mieux des conditions réelles d'utilisation. La première série de mesures (figure 5.26) évalue le transfert d'un objet de 52 octets tandis que la seconde (figure 5.27) évalue le transfert d'un object de 294 octets.

Sur les figures 5.26 et 5.27, nous constatons, par les courbes repérées par des carrés, que le temps de transfert augmente logiquement avec le nombre de boucles. Nous remarquons en revanche que le temps par boucle diminue jusqu'à atteindre la saturation du CPU ou du réseau. Ainsi le débit n'est pas seulement limité par la charge réseau, il est également limité par la charge du CPU. Cette limitation justifie le choix de de vérifier le CPU disponible des périphériques, immédiatement après l'énergie, dans le classement des périphériques de

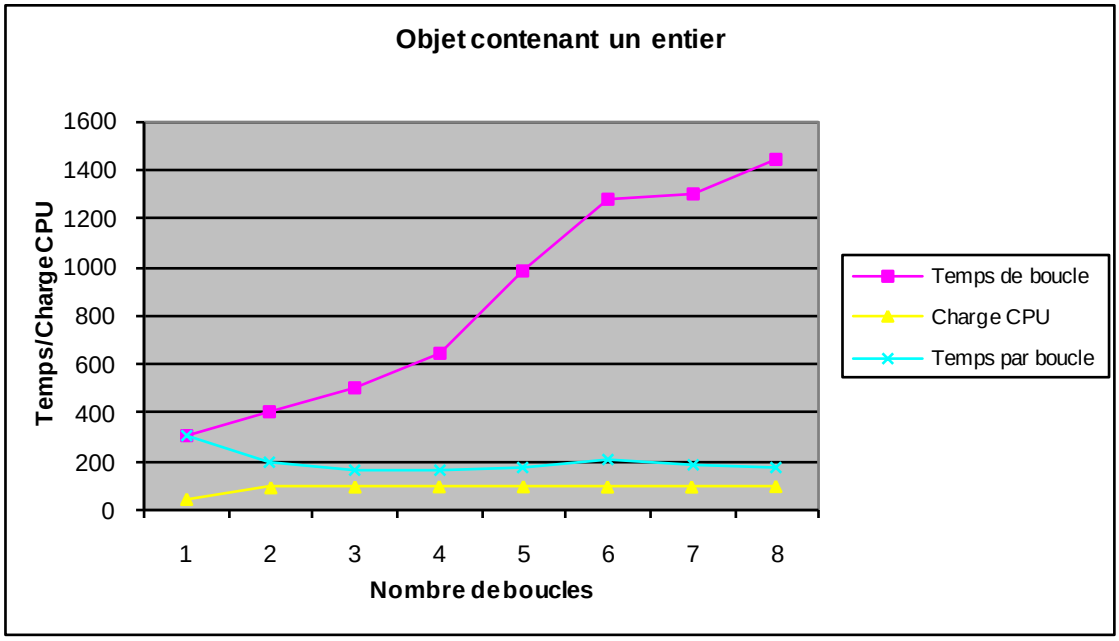

FIGURE 5.26 – Analyse du test de transfert d'un objet de 52Ko

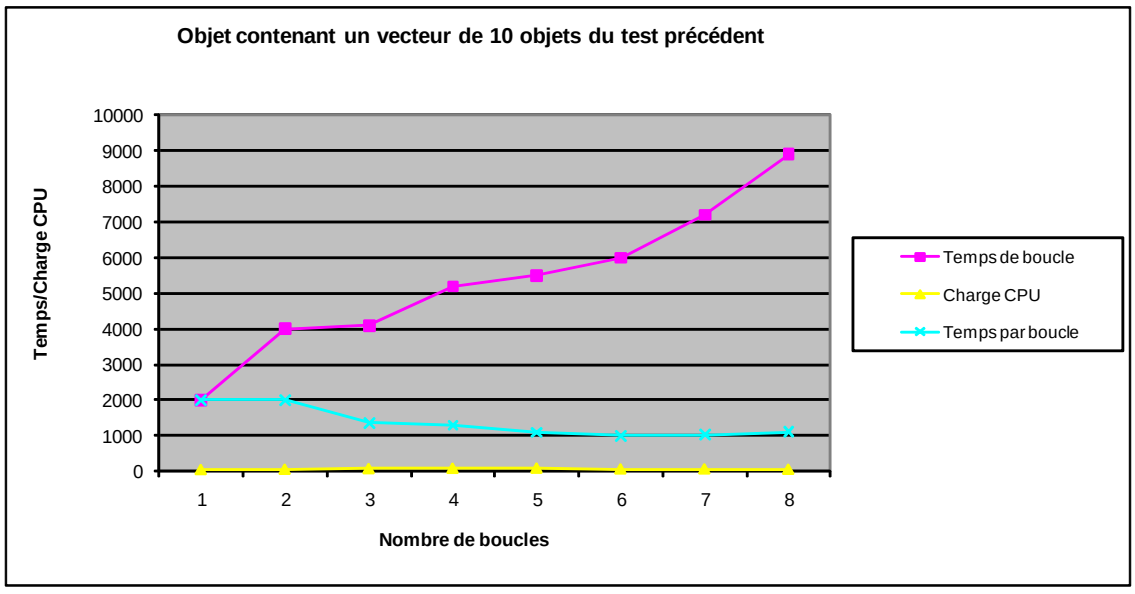

FIGURE 5.27 – Analyse du test de transfert d'un objet de 294 Ko

l'heuristique.

Enfin, nous mesurons un temps de transfert minimal de 80 ms dans la première série et de 500 ms dans la seconde.

Ces temps de transfert, tout comme pour les temps d'exécution des commandes de

reconfiguration, restent de l'ordre de la milliseconde et reste donc acceptable pour des périphériques très contraints.

5.6 Conclusion

En conclusion, ces mesures de performances montrent que le temps de réponse de la plate-forme face à une demande de reocnfiguration est acceptable.

Le temps de transfert d'informations est essentiellement dû à l'absence de sérialisation dans la machine virtuelle java Squawk des capteurs Sun Spots. Pour palier à ce manque, une méthode de sérialisation a été écrite spécialement pour les Sun Spots. Cependant, elle utilise une bibliothèque écrite en java ; la virtualisation rend l'exécution d'une telle méthode assez lente. En effet la sérialisation dans la première série prend 56% du temps et 90% pour la seconde série.

Enfin il n'existe pas non plus de méthode de clonage dans Squawk. Par conséquent, tout comme pour la sérialisation, une méthode de clonage a été écrite pour chaque type d'objet qui peut être transféré.

En résumé, bien que la taille des données à transférer est un facteur limitant le débit, dans le cas des périphériques CLDC tels que les capteurs, la charge CPU est la principale responsable de la limitation du débit. Cette constatation conforte notre proposition de classement des périphériques lors du choix d'une configuration afin d'obtenir une application de qualité suffisante.

Chapitre 6

Conclusion

Conclusion

L'informatique pervasive est devenue une réalité. L'arrivée massive de périphériques nomades permet d'envisager des applications utilisant ces dispositifs en même temps que des machines plus classiques. En effet les utilisateurs souhaitent disposer, sur leurs PDA et téléphones portables, de services équivalents à ceux qu'ils ont l'habitude d'utiliser. En outre, en raison de la mobilité naturellement induite par ces dispositifs, la qualité du service offert ne peut pas faire abstraction du contexte environnemental. La récente mise sur le marché de capteurs sans-fil dotés de processeurs et de moyens de communication permet d'imaginer l'intégration de ces dispositifs aux applications proposées afin de disposer d'informations sur l'environnement d'exécution de ces applications. L'introduction de périphériques légers dans de telles applications se heurte aux problèmes liés à leur mobilité, à leurs faibles ressources, aux débits limités de communication qu'ils offrent et enfin, pour la majorité d'entre eux, aux contraintes d'énergie des batteries.

La grande majorité des travaux liés aux périphériques contraints concernent l'optimisation des ressources matérielles (capacité de calcul, énergie) et réseau (contrôle de congestion, agrégation des données, etc.). Toutefois il s'agit généralement de solutions ad-hoc ne pouvant pas s'étendre à l'ensemble des composants de l'application. De façon analogue, la plupart des travaux traitant de l'adaptation dynamique des applications et leur qualité de service sont centrées essentiellement autour de l'utilisateur. Peu de travaux prennent en compte simultanément plusieurs de ces aspects et particulièrement peu s'intéressent à l'aspect applicatif lui-même, ce que nous appelons les spécifications de l'application. De plus, avec l'arrivée et la démocratisation des capteurs sans-fil, il est possible d'obtenir des informations sur le contexte environnemental qui permettent d'améliorer l'application en proposant de nouvelles informations. Néanmoins, les capteurs y sont utilisés pour leurs fonctions propres de mesure de l'environnement et leur capacité à transmettre et relayer l'information, en veillant à en maximiser la durée de vie. Il existe actuellement peu de recherches sur l'intégration de tels dispositifs dans des environnements hétérogènes où collaborent composants logiciels et capteurs.

En effet, afin d'accroître l'offre des possibilités d'adaptation nous pensons qu'il est intéressant de ne plus considérer les capteurs simplement comme des dispositifs capables

d'effectuer des mesures. Lorsqu'ils n'utilisent pas la totalité de leur mémoire et de leur capacité de calcul, il leur est possible d'héberger d'autres composants logiciels en relation ou non avec leurs fonctions propres. Ainsi les capteurs participent à l'infrastructure matérielle des applications en offrant de nouvelles possibilités d'hébergement de fonctionnalités et d'adaptation. Ceci permet de proposer de nouvelles configurations pour les applications et, par conséquent, d'accroître les possibilités d'offrir une qualité de service suffisante.

Partant de ce constat, nous avons proposé dans cette thèse, une plate-forme de supervision des applications, utilisant la mesure de la qualité de service pour les adapter aux changements du contexte. Cette plate-forme se base sur un modèle de QdS qui regroupe plusieurs aspects habituellement utilisés dans le domaine des applications pervasives : la prise en compte des besoins de l'utilisateur, la prise en compte des ressources matérielles et réseau et enfin la prise en compte des contraintes d'utilisation de l'application. Ce modèle de QdS est ensuite utilisé par le mécanisme d'adaptation. Ce dernier se base sur une heuristique de choix de la configuration à déployer pour mettre en œuvre les reconfigurations.

Un modèle d'évaluation de la qualité de service pour les applications pervasives

Dans cette thèse nous avons abordé la problématique de la gestion de la qualité de service (QdS) des applications pour l'adaptation au contexte dans le domaine des applications pervasives. Pour cela nous nous sommes intéressés aux différentes définitions de la qualité de service ainsi qu'à l'adaptation structurelle des applications. La plupart des travaux concernant la qualité de service se focalisent sur un aspect en particulier. Par exemple dans les applications distribuées c'est l'aspect réseau comme le temps de latence qui est pris en compte. Dans les applications multimédias c'est plutôt l'aspect utilisateur qui est pris en compte, et plus particulièrement sa satisfaction face au service rendu, alors que dans les applications pour périphériques contraints c'est l'aspect matériel concernant la consommation de ressources qui est privilégié. Concernant l'adaptation structurelle, nous pouvons constater que l'utilisation des intergiciels et plate-formes de supervision est très répandue.

Chacun de ces aspects de qualité de service est influencé par différents éléments regroupés sous le nom commun contexte : type de connexion, nombre de couleurs, débit de transfert, langue, température, localisation, mémoire disponible, etc. Chaque élément n'influence qu'une partie de la QdS de l'application. Par exemple, le nombre de couleurs pour une vidéo influence la satisfaction de l'utilisation alors qu'un manque de mémoire disponible gène le fonctionnement de l'application. Pour mieux traiter les changements de contexte et mieux cibler les réactions du système, nous proposons trois catégories de contexte :
Le contexte utilisateur qui concerne les souhaits de l'utilisation par rapport aux fonctionnalités fournies par l'application.
Le contexte d'utilisation qui est une représentation des spécifications des règles de fonctionnement de l'application en fonction de situations données.
Le contexte d'exécution qui concerne les préoccupations matérielles et réseau des périphériques.

A partir de ce constat, nous proposons dans cette thèse une définition originale de la qualité de service qui englobe les différents aspects que nous trouvons dans la littérature,

et plus particulièrement, la prise en compte de l'application elle-même. Notre définition est la suivante :
« *La qualité de service d'une application est tout d'abord la garantie de sa continuité malgré les défaillances matérielles et réseaux. La qualité de service d'une application doit également refléter le bon fonctionnement de cette dernière par son adéquation avec les spécifications de l'application et les souhaits de l'utilisateur. Enfin, la qualité de service d'une application est fortement influencée par son infrastructure. La structure et la répartition de l'application doit permettre un fonctionnement le plus long possible.* »

Afin de gérer la qualité de service d'une application telle que nous l'avons définie, nous avons distingué deux types de QdS : la *QdS Utilité* et la *QdS Pérennité*.

La *QdS Utilité* est principalement influencée par le contexte utilisateur et le contexte d'utilisation. Elle mesure l'adéquation de l'application fournie avec les souhaits de l'utilisateur et le cahier des charges de l'application.

La *QdS Pérennité* est principalement influencée par l'infrastructure de l'application, c'est-à-dire le contexte d'exécution. Elle mesure la durée de vie de l'application par rapport aux ressources disponibles sur les périphériques et au débit réseau utilisé. En effet, nos applications visent à être déployée sur des périphériques autonomes énergétiquement, nous devons par conséquent optimiser les dépenses d'énergie.

Cette définition est accompagnée d'une méthode de conception afin d'aider le concepteur de l'application à décrire les configurations utilisées par le mécanisme d'adaptation. La méthode aide également à décrire le contexte à savoir :

- Les règles de contexte utilisateur
- Les règles de contexte d'utilisation.
  Ces règles permettent notamment de mettre à jour les notes de QdS des configurations dans le but de respecter la *QdS Utilité*.
- Les règles de contexte d'exécution.
  Ces règles permettent de définir les seuils de QdS et de définir les mises à jour des seuils.

Après avoir déterminé un modèle d'évaluation de la QdS, la deuxième contribution de cette thèse est de fournir une plate-forme de déploiement contextuel des applications : Kalimucho.

Kalimucho : la gestion de la QdS pour l'adaptation au contexte

Les plate-formes de supervision sont des solutions répandues dans le domaine de l'adaptation dynamique d'applications distribuées. La plupart des solutions se basent sur une description détaillée des composants afin de construire à la volée une configuration possédant les caractéristiques nécessaires pour satisfaire les exigences de QdS. Des solutions comme Kalinahia [43] [44], Qua [23] et Qinna [77] [76] utilisent la mesure de la QdS pour engendrer des reconfigurations. Le mécanisme d'adaptation de Kalinahia repose sur une heuristique permettant de construire une configuration à partir des exigences de l'utilisateur et du contexte d'exécution. Qua utilise une fonction d'utilité tandis que Qinna fait correspondre une observation de QdS, par exemple bonne qualité ou mauvaise qualité, à un contrat que doit respecter le système. Cette dernière solution contrairement aux autres, prend en

compte le caractère contraint des périphériques en termes de ressources physiques telles que le CPU, la mémoire et l'énergie. Cette contrainte est également prise en compte par des solutions comme MUSIC [64][65][59] pour la sélection d'une configuration, ou AxSel [30] pour le choix d'un déploiement. En revanche aucune ne prend en compte le coût des liens réseaux dû à la répartition.

L'originalité de la plate-forme Kalimucho est l'intégration de la prise en compte du coût des liaisons réseau dans la recherche d'une configuration à déployer lors d'une reconfiguration. En effet, les transmissions réseaux sont très coûteuses en énergie pour des périphériques contraints, nous considérons alors qu'il est nécessaire de les prendre en compte dans la mesure de la QdS d'une application.

Pour cela nous avons proposé une heuristique de recherche d'une configuration à déployer qui repose dans un premier temps sur le respect de la *QdS Utilité*, puis dans un deuxième temps sur le respect de la *QdS Pérennité*. Cette heuristique ne permet pas d'obtenir une solution optimale puisque le problème d'optimisation est connu comme étant NP-complet. Cependant, elle est guidée par une série de classement en fonction de la connectivité, de la disponibilité des ressources réseau, CPU, mémoire et énergie, qui permet d'obtenir une solution viable rapidement. De plus nous proposons que Kalimucho soit répartie sur les différents périphériques de l'application. Compte tenu de l'hétérogénéité de ces périphériques, nous avons proposé plusieurs déploiements possibles de Kalimucho.

Perspectives

La méthode de conception que nous avons proposée permet de définir, pour chaque application, les critères significatifs de prise en compte du contexte (matériel, environnemental, géographique, utilisateur, etc.) ainsi que de décomposer ces applications en composants autonomes, mobiles, communiquant entre eux. Cependant cette méthodologie utilise des modèles et un langage propres à ces travaux. Pour pouvoir développer de nouvelles applications et intégrer de nouveaux services et de nouveaux matériels rapidement, nous devons proposer une approche suffisamment générale au niveau des modèles. Une telle approche peut être aidée par l'ingénierie des modèles par exemple. Il serait ainsi possible de proposer une représentation indépendante de la plate-forme de l'application, ses services et ses périphériques, et obtenir un modèle approprié à partir de règles de transformations lors du choix de l'implémentation et de la plate-forme. Une telle approche permettrait alors la génération automatique de code et la vérificiation des architectures avant l'implantation telle que le propose [5].

Le prototype que nous proposons permet de valider le fonctionnement de l'heuristique et de montrer qu'elle trouve une solution viable rapidement bien qu'elle ne soit pas optimale. Il permet également de valider l'utilisation des cartes d'identité des périphériques et des composants pour l'étude d'un déploiement ainsi que le modèle de QdS que nous avons défini pour le choix d'un déploiement. Cependant ce prototype simule uniquement la recherche d'une configuration depuis une machine classique. Il ne permet pas de valider la distribution de la plate-forme sur les différents périphériques mobiles et la collaboration entre les services. Dans cet objectif, une implémentation de la plate-forme a été réalisée. Elle comprend une partie du service *Superviseur*, les services *Usine à Conteneur* et *Usine*

à Connecteur, le service *Routage* et le registre de composants. Elle permet de déployer une configuration et de recueillir les informations contextuelles provenant des conteneurs de composants et de connecteurs. En revanche, elle ne possède pas le service *Générateur de Reconfiguration* chargé de la prise de décision de reconfiguration. Les perspectives à moyen termes sont de proposer un prototype complet de la plate-forme, déployé sur des périphériques mobiles, et, intégrant l'heuristique de choix de la configuration dans la plate-forme existante, afin de pouvoir tester le schéma complet de l'adaptation dynamique.

Dans cette thèse nous nous sommes concentrés sur l'adaptation structurelle et le déploiement contextuel des applications en environnement contraint. Les nouvelles générations de périphériques mobiles tels que les téléphones intelligents, permettent à présent aux utilisateurs d'échanger des données multimédias telles que des photos, des musiques et des vidéos. En plus de l'hétérogénéité des périphériques et les limitations de leurs ressources, les concepteurs d'applications sont également confrontés aux problèmes liés à l'hétérogénéité des données, des formats (codec), des types (images, vidéos, audio) et des caractéristiques des périphériques (taille de l'écran, nombre de couleurs, etc.). Des travaux sont actuellement en cours afin de proposer un méta-modèle pour les architectures logicielles multimédias permettant l'adaptation de formats de flux de données. Cette approche se base sur la description des flux de données conformément à une ontologie et propose des stratégies d'adaptation mises en œuvre par des connecteurs. Il peut être intéressant d'intégrer ces travaux à la plate-forme Kalimucho afin de pouvoir proposer des reconfigurations encore mieux adaptées aux applications actuellement utilisées.

Enfin, dans cette thèse, nous avons choisi de considérer les périphériques mobiles, non plus comme de simples terminaux personnels, mais comme des supports éventuels pour héberger des services en relation ou non avec leurs fonctions propres (comme le cas des capteurs). Ainsi ils participent à l'infrastructure matérielle des applications, en offrant de nouvelles possibilités d'hébergement de fonctionnalités. Par ailleurs, tous ces périphériques ne sont pas dotés des mêmes dispositifs. Ainsi, par exemple, tous ne disposent pas d'un GPS. La possibilité pour la plate-forme de déployer des services, peut permettre à un utilisateur dépourvu de GPS, de se localiser géographiquement du seul fait qu'il se situe à proximité d'un dispositif qui en est doté et sur lequel la plate-forme a installé un service à cet effet. De plus, la composition de services peut permettre la réalisation de services, non initialement prévus par le concepteur de l'application, mais nécessaires sur le moment à l'utilisateur. Actuellement, la plate-forme Kalimucho ne permet pas une telle modularité. Les services sont figés à la conception dans des configurations pour plus de facilité lors de la recherche de la configuration à déployer en cas de reconfiguration. Proposer une telle plate-forme permettrait de proposer de nouvelles fonctionnalités aux utilisateurs et, par conséquent, d'accroître les possibilités d'offrir une qualité de service suffisante à ces derniers sans figer les configurations. L'idée est de proposer un modèle de plate-forme, modulable, à la façon d'un puzzle, qui permettrait d'une part, de déployer une plate-forme adaptée aux capacités du périphérique et, d'autre part, de construire à la volée la plate-forme adaptée au contexte d'utilisation.

Publications

Journaux

Cyril Cassagnes, Philippe Roose, Marc Dalmau, **Christine Louberry** *KALIMUCHO : software architecture for limited mobile devices*, ACM SIGBED Review, Volume 6, Number 3, October 2009, Special Issue on the 2nd Workshop on Adaptive and Reconfigurable Embedded System (IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design), 11/10/2009, Grenoble, France, ISSN 978-0-615-32386-2.

Christine Louberry, Philippe Roose, Marc Dalmau *Heterogeneous component interactions : Sensors integration into multimedia applications*, Journal of Networks, Vol.3 Issue 4, Academy Publisher, ISSN : 1796-2056, 2008.

Christine Louberry, Marc Dalmau, Philippe Roose *Intégration de périphériques contraints par reconfiguration dynamique d'applications*, Numéro Spécial Revue ISI, Objets Composants Modèles dans l'ingénierie des Systèmes d'Informations, Hermès 3/2008 (juin 2008).

Conférences et ateliers internationaux

Christine Louberry, Philippe Roose, Marc Dalmau *QoS Based Design Process for Pervasive Computing Applications*, short paper, ACM Mobility 2009, 2 - 4 september 2009, Nice, France.

Christine Louberry, Philippe Roose, Marc Dalmau *QoS-Based Design Method for Constraint Device Based Applications*, 4th ACM International Workshop on Services Integration in Pervasive Environments (SIPE 2009), July 13-17, 2009, London, UK.

Christine Louberry, Philippe Roose, Marc Dalmau *Towards Sensor Integration into Multimedia Applications*, 4th European Conference on Universal Multiservice Networks ECUMN'2007, Toulouse, France, 12-14 February, 2007, IEEE Computer Society Press, ISBN : 0-7695-2768-X, pp 355-363.

Roose Philippe, Dalmau Marc, **Louberry Christine** *A Unified Component Model for Sensor Integration into Multimedia Applications*, ACM SIGPLAN, Workshop at OOPSLA Building Software for Sensor Networks, Portland, Oregon (USA), 22-26 October 2006.

Conférences et ateliers nationaux

Cyril Cassagnes, Philippe Roose, Marc Dalmau, **Christine Louberry** *KALIMUCHO : Architecture logicielle pour périphériques mobiles contraints*, Toulouse'09 (Rempar), Papier court, 9-11 septembre 2008, Toulouse, France.

Christine Louberry, Marc Dalmau, Philippe Roose *Software Architecture for Dynamic Adaptation of Heterogeneous Applications*, NOTERE 2008, ACM Digital Library, 23-27 juin 2008.

Christine Louberry, Philippe Roose, Marc Dalmau *Modèle Unifié de Composants pour l'Intégration de Capteurs dans des Applications Multimédia*, Inforsid 2006, Atelier OCM-SI, Hammameth, Tunisie, 01-03 Juin 2006.

Autres Communications

Christine Louberry, Philippe Roose, Marc Dalmau, Architecture logicielle pour la gestion de la qualité de service en environnement contraint, Ingénierie des modèles et Adaptation dynamique, Journée de l'action ADAPT, GDR ASR, 13 novembre 2008, Brest, France.

Christine Louberry, Marc Dalmau et Philippe Roose, Architecture logicielle pour des applications hétérogènes, distribuées et reconfigurables, Journée de l'action ADAPT, GDR ASR, 14 Février 2008, Fribourg, Suisse.

Christine Louberry *Gestion de la Qualité de Service par Déploiement et Adaptation dynamique de Composants hétérogènes en Environnement mouvant*, Inforsid 2007, Forum Jeunes Chercheurs, Perros-Guirec, France.

Bibliographie

[1] OSGi Service Platform Service Compendium Realease 4. Technical report, version 4.1, OSGi Alliance, 2007.

[2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks : a survey. *Computer Networks*, 38(4) :393–422, 2002.

[3] Mourad Alia, Viktor S. Wold Eide, Nearchos Paspallis, Frank Eliassen, Svein O. Hallsteinsen, and George A. Papadopoulos. A utility-based adaptivity model for mobile applications. *Advanced Information Networking and Applications Workshops, International Conference on*, 2 :556–563, 2007.

[4] Mourad Alia, Viktor S. Wold Eide, Nearchos Paspallis, Frank Eliassen, Svein O. Hallsteinsen, and George A. Papadopoulos. A utility-based adaptivity model for mobile applications. In *AINA Workshops (2)*, pages 556–563. IEEE Computer Society, 2007.

[5] Adel Alti and Adel Smeda. Integration of architectural design and implementation decisions into the mda framework. In José Cordeiro, Boris Shishkov, Alpesh Ranchordas, and Markus Helfert, editors, *ICSOFT (PL/DPS/KE)*, pages 366–371. INSTICC Press, 2008.

[6] Sten Lundesgaard Amundsen, Ketil Lund, Carsten Griwodz, and Pål Halvorsen. Qos-aware mobile middleware for video streaming. In *EUROMICRO-SEAA*, pages 54–61. IEEE Computer Society, 2005.

[7] Vogel Andreas, Kerhervé Brigitte, von Bochmann Gregor, and Gecsei Jan. Distributed multimedia and qos : A survey. *IEEE MultiMedia*, 2(2) :10–19, 1995.

[8] Dhouha Ayed, Chantal Taconet, Guy Bernard, and Yolande Berbers. Cadecomp : Context-aware deployment of component-based applications. *J. Network and Computer Applications*, 31(3) :224–257, 2008.

[9] E. Bouix, P. Roose, and M. Dalmau. The korrontea data modeling. In *Ambi-Sys '08 : Proceedings of the 1st international conference on Ambient media and systems*, pages 1–10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[10] Emmanuel Bouix, Marc Dalmau, Philippe Roose, and Franck Luthon. Multimedia oriented component model. In *AINA*, pages 3–8. IEEE Computer Society, 2005.

[11] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et science informatiques*, 20(4) :289–512, 2001.

[12] T. Chaari, D. Ejigu, F. Laforest, and V.-M Scuturici. Modeling and using context in adapting applications to pervasive environments. *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 111–120, 2006.

[13] Tarak Chaari, Frédérique Laforest, and André Flory. Adaptation des applications au contexte en utilisant les services web. In *UbiMob '05 : Proceedings of the 2nd French-speaking conference on Mobility and ubiquity computing*, pages 111–118, New York, NY, USA, 2005. ACM.

[14] Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stephane Lavirotte, and Michel Riveill. Wcomp : a multi-design approach for prototyping applications using heterogeneous resources. In *IEEE International Workshop on Rapid System Prototyping*, pages 119–125. IEEE Computer Society, 2006.

[15] Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stephane Lavirotte, and Michel Riveill. Self-adaptation of event-driven component-oriented middleware using aspects of assembly. In Sotirios Terzis, Steve Neely, and Nitya Narasimhan, editors, *MPAC*, ACM International Conference Proceeding Series, pages 31–36. ACM, 2007.

[16] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Cosmos, composition de noeuds de contexte. *Technique et Science Informatiques*, 27(9-10) :1189–1224, 2008.

[17] Marc Dalmau. *Plates-formes et virtualisation pour gérer la complexité*. Habilitation à diriger des recherches, Université de Pau et des Pays de l'Adour, 2008.

[18] Stéphane Boyera Dave Raggett. Composite capabilities/preference profiles. Technical report, W3C (http ://www.w3.org/Mobile/CCPP/).

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

[20] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *In DAIS03, volume 2893 of LNCS*, pages 1–14. Springer-Verlag, 2003.

[21] Umeshwar Dayal, Eric N. Hanson, and Jennifer Widom. Active database systems. In *Modern Database Systems*, pages 434–456. 1995.

[22] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1) :4–7, 2001.

[23] Frank Eliassen, Richard Staehli, Gordon S. Blair, and Jan Øyvind Aagedal. Qua : building with reusable qos-aware components. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 154–155. ACM, 2004.

[24] Bounaas F. Using eca rules for object and schema evolution in an object-oriented system. In *Technology of Object-Oriented Languages and Systems*, 1995.

[25] David Fernández-Baca. Allocating modules to processors in a distributed system. *IEEE Trans. Software Eng.*, 15(11) :1427–1436, 1989.

[26] Jacqueline Floch, Svein O. Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjørven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2) :62–70, 2006.

[27] Leonard J. N. Franken and Boudewijn R. Haverkort. Quality of service management using generic modelling and monitoring techniques. *Distributed Systems Engineering*, 4(1) :28–37, 1997.

[28] Bieber G and Carpenter J. Openwings : A service-oriented component architecture for self-forming, self-healing. Technical report, openwings document, 2003.

[29] M. R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[30] Amira Ben Hamida, Frederic Le Mouel, Stéphane Frénot, and Mohamed Ben Ahmed. A graph-based approach for contextual service loading in pervasive environments. In Meersman and Tari [50], pages 589–606.

[31] Amira Ben Hamida, Frédéric Le Mouël, Stéphane Frénot, and Mohamed Ben Ahmed. Une approche pour un chargement contextuel de services dans les environnements pervasifs. *Ingénierie des Systèmes d'Information*, 13(3) :59–82, 2008.

[32] George T. Heineman and William T. Council. Component-based software engineering, putting the pieces together. *Addison-Wesley*, 2001.

[33] Vincent Hourdin, Jean-Yves Tigli, Stephane Lavirotte, Gaëtan Rey, and Michel Riveill. Slca, composite services for ubiquitous computing. In Jason Yi-Bing Lin, Han-Chieh Chao, and Peter Han Joo Chong, editors, *Mobility Conference*, page 11. ACM, 2008.

[34] Jadwiga Indulska, Ricky Robinson, Andry Rakotonirainy, and Karen Henricksen. Experiences in using cc/pp in context-aware systems. In Ming-Syan Chen, Panos K. Chrysanthis, Morris Sloman, and Arkady B. Zaslavsky, editors, *Mobile Data Management*, volume 2574 of *Lecture Notes in Computer Science*, pages 247–261. Springer, 2003.

[35] Michael Jeronimo and Jack Weast. *UPnP Design by Example : A Software Developer's Guide to Universal Plug and Play*. Intel Press, may 2003.

[36] Inc. Jon Courtney, Sun Microsystems. Connected device configuration. Technical report, Sun Microsystems, Inc. (http ://-jcp.org/aboutJava/communityprocess/mrel/jsr218/index.html).

[37] Inc. Jon Courtney, Sun Microsystems. Connected limited device configuration. Technical report, Sun Microsystems, Inc. (http ://-jcp.org/aboutJava/communityprocess/mrel/jsr218/index.html).

[38] Holger Karl, Andreas Willig, and Adam Wolisz, editors. *Wireless Sensor Networks, First European Workshop, EWSN 2004, Berlin, Germany, January 19-21, 2004, Proceedings*, volume 2920 of *Lecture Notes in Computer Science*. Springer, 2004.

[39] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[40] Fernando A. Kuipers and Piet Van Mieghem. Mamcra : a constrained-based multicast routing algorithm. *Computer Communications*, 25(8) :802–811, 2002.

[41] Robert Laddaga, Paul Robertson, and Howard E. Shrobe, editors. *Self-Adaptive Software, Second International Workshop, IWSAS 2001, Balatonfüred, Hungary, May*

17-19, 2001 Revised Papers, volume 2614 of *Lecture Notes in Computer Science*. Springer, 2003.

[42] Sophie Laplace. *Conception d'Architectures Logicielles pour intégrer la qualité de service dans les applications multimédias réparties*. Thèse de doctorat, Université de Pau et des Pays de l'Adour, 2006.

[43] Sophie Laplace, Marc Dalmau, and Philippe Roose. Kalinahia : Considering quality of service to design and execute distributed multimedia applications. In *NOMS*, pages 951–954. IEEE, 2008.

[44] Sophie Laplace, Marc Dalmau, and Philippe Roose. Kalinahia : Considering quality of service to design and execute distributed multimedia applications. *CoRR*, abs/0812.2529, 2008.

[45] Sung-Ju Lee, Elizabeth M. Belding-Royer, and Charles E. Perkins. Ad hoc on-demand distance-vector routing scalability. *Mobile Computing and Communications Review*, 6(3) :94–95, 2002.

[46] Chunhung Richard Lin and Jain Shing Liu. Qos routing in ad hoc wireless networks. *IEEE Journal On Selected Areas In Communications*, 17(8) :1426–1438, 1999.

[47] Christine Louberry, Philippe Roose, and Marc Dalmau. Towards sensor integration into multimedia applications. In *ECUMN*, pages 355–363. IEEE Computer Society, 2007.

[48] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference model for service oriented architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, 2006.

[49] V. Maurin, N. Dalmasso, B. Copigneaux, Stephane Lavirotte, Gaëtan Rey, and Jean-Yves Tigli. Simplyengine-wcomp : plate-forme de prototypage rapide pour l'informatique ambiante basée sur une approche orientée services pour dispositifs réels/virtuels. In David Menga and Florence Sedes, editors, *UbiMob*, volume 394 of *ACM International Conference Proceeding Series*, pages 83–86. ACM, 2009.

[50] Robert Meersman and Zahir Tari, editors. *On the Move to Meaningful Internet Systems : OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, November 9-14, 2008, Proceedings, Part I*, volume 5331 of *Lecture Notes in Computer Science*. Springer, 2008.

[51] Bertrand Meyer. The grand challenge of trusted components. In *Software Engineering, International Conference on Software Engineering*, page 660, 2003.

[52] Sun Microsystems. Technical report, SunSpot World (http ://www.sunspotworld.com).

[53] Rex Min, Manish Bhardwaj, Seong-Hwan Cho, Eugene shih, Amit Sinha, Alice Wang, Anantha Chandrakasan, and Eugene Shih Amit Sinha. Low-power wireless sensor networks. In *In VLSI Design*, pages 205–210, 2001.

[54] F. J. Moomena. *Modélisation des architectures logicielles dynamiques : application à la gestion de la qualité de service des applications à base de services Web*. Thèse de doctorat, Institut National Polytechnique de Toulouse, 2007.

[55] Brian Noble, Morgan Price, and Mahadev Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. *Computing Systems*, 8(4) :345–363, 1995.

[56] Brian Noble, Mahadev Satyanarayanan, and Morgan Price. A programming interface for application-aware adaptation in mobile computing. In *Symposium on Mobile and Location-Independent Computing*, pages 57–66. USENIX, 1995.

[57] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing : State of the art and research challenges. *Computer*, 40(11) :38–45, 2007.

[58] Jason Pascoe, Nick Ryan, and David R. Morse. Using while moving : Hci issues in fieldwork environments. *ACM Trans. Comput.-Hum. Interact.*, 7(3) :417–437, 2000.

[59] Nearchos Paspallis, Romain Rouvoy, Paolo Barone, George A. Papadopoulos, Frank Eliassen, and Alessandro Mamelli. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In Meersman and Tari [50], pages 553–570.

[60] Charles Perkins, Elizabeth Belding-Royer, Samir Das, and Ian Chakeres. Ietf rfc 3561. Technical report, MANET (http ://www.ietf.org/rfc/rfc3561.txt), 2003.

[61] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *SIGCOMM*, pages 234–244, 1994.

[62] Victor J. Rayward-Smith. The complexity of preemptive scheduling given interprocessor communication delays. *Inf. Process. Lett.*, 25(2) :123–125, 1987.

[63] Gaëtan Rey and Joëlle Coutaz. Le contexteur : une abstraction logicielle pour la réalisation de systèmes interactifs sensibles au contexte. In Patrick Girard, Thomas Baudel, Michel Beaudouin-Lafon, Eric Lecolinet, and Dominique L. Scapin, editors, *IHM*, volume 32 of *ACM International Conference Proceeding Series*, pages 105–112. ACM, 2002.

[64] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein O. Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 164–182. Springer, 2009.

[65] Romain Rouvoy, Mikaël Beauvois, Laura Lozano, Jorge Lorenzo, and Frank Eliassen. Music : an autonomous platform supporting self-adaptive mobile applications. In Oriana Riva and Luís Veiga, editors, *Mobile Middleware*, page 6. ACM, 2008.

[66] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit : Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[67] Bill N. Schilit and Marvin M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5) :22–32, 1994.

[68] Holger Schmidt and Franz J. Hauck. Samproc : middleware for self-adaptive mobile processes in heterogeneous ubiquitous environments. In *Middleware (Doctoral Symposium)*, page 11, 2007.

[69] J. Shlimmer and J. Thelin. Device profile for web services. Technical report, http ://-schemas.xmlsoap.org/ws/2006/02/devprof/, Février 2006.

[70] K. P. Smith. *Managing Rules in Active Databases*. Ph. d. thesis, Champaign, 1992.

[71] Katayoun Sohrabi, Jay Gao, Vishal Ailawadhi, and Gregory J Pottie. Protocols for self-organization of a wireless sensor network. *IEEE Personal Communications*, 7 :16–27, 2000.

[72] João Pedro Sousa and David Garlan. Aura : an architectural framework for user mobility in ubiquitous computing environments. In Jan Bosch, W. Morven Gentleman, Christine Hofmeister, and Juha Kuusela, editors, *WICSA*, volume 224 of *IFIP Conference Proceedings*, pages 29–43. Kluwer, 2002.

[73] Clemens Szyperski. Component software - beyond object-oriented programming. *Addison-Wesley*, 1998.

[74] J. Y. Tigli, D. Cheung-Foo-Wo, S. Lavirotte, and M. Riveill. Adaptation au contexte par tissage d'aspects d'assemblage de composants déclenchés par des conditions contextuelles. *RSTI Série ISI - Adaptation et Gestion du Contexte*, 11(5) :89–114, 2006.

[75] Jean-Yves Tigli, Stephane 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. *Annales des Télécommunications*, 64(3-4) :197–214, 2009.

[76] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. An evaluation of qinna, a component-based qos architecture for embedded systems. In Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright, editors, *SAC*, pages 998–1002. ACM, 2005.

[77] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. Qinna, a component-based qos architecture. In George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2005.

[78] UIT. X.902 : Technologies de l'information, traitement réparti ouvert, modèle de référence : fondements. Recommandation x.902 (11/95), Union Internationale des Télécommunications (http ://www.itu.int/net/home/index-fr.aspx), 1995.

[79] Bart Veltman, B. J. Lageweg, and Jan Karel Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2-3) :173–182, 1990.

[80] Patrice Vienne, Jean-Louis Sourrouille, and Mathieu Maranzana. Modeling distributed applications for qos management. In Thomas Gschwind and Cecilia Mascolo, editors, *SEM*, volume 3437 of *Lecture Notes in Computer Science*, pages 170–184. Springer, 2004.

[81] T. w. Chen, J. T. Tsai, and M. Gerla Y. Qos routing performance in multihop, multimedia, wireless networks. In *In Proceedings of IEEE International Conference on Universal Personal Communications (ICUPC*, pages 557–561, 1997.

[82] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *ICAC*, pages 70–77. IEEE Computer Society, 2004.

[83] Zheng Wang and Jon Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7) :1228–1234, 1996.

[84] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3) :66–75, September 1991.

[85] Mohamed F. Younis, Kemal Akkaya, Mohamed Eltoweissy, and Ashraf Wadaa. On handling qos traffic in wireless sensor networks. In *HICSS*, 2004.

[86] Chenxi Zhu and M. Scott Corson. Qos routing for mobile ad hoc networks. In *INFOCOM*, 2002.