Déploiement adaptatif d'applications orientées services sur environnements contraints

Amira Ben Hamida, Frédéric Le Mouël, Stéphane Frénot, Mohamed Ben Ahmed

Déploiement adaptatif d'applications orientées services sur environnements contraints

Amira Ben Hamida [*] [**]–Frédéric Le Mouël [*]–Stéphane Frénot [*]–Mohamed Ben Ahmed [**]

[*] *Université de Lyon, INRIA*
INSA-Lyon, CITI, F-69621, Villeurbanne, France
{amira.ben-hamida, frederic.le-mouel, stephane.frenot}@insa-lyon.fr

[**] *ENSI-Tunis*
RIADI, T-2010, Tunisie
{mohamed.benahmed}@riadi.rnu.tn

RÉSUMÉ. *L'installation et l'exécution d'applications sur les dispositifs mobiles est une tâche difficile en raison de la limite des ressources de ceux-ci. Dans cet article, nous proposons une approche de déploiement contextuel d'applications orientées services, basée sur la coloration de graphes selon des contraintes de ressources. Nous représentons l'application sous la forme d'un graphe bidimensionnel dynamique de dépendances entre services et composants. La coloration du graphe fournit la configuration optimale de déploiement de l'application à un instant et contexte donnés. Des mécanismes d'écoute du contexte capturent les changements et répercutent une recoloration et un redploiement dynamiques. L'architecture AxSeL (A conteXtual Service Loader) est validée par un prototype OSGi et illustrée par un cas d'utilisation réel.*

ABSTRACT. *While installing and executing applications on mobile devices, the issue of the limit of resources is quickly encountered. In this paper we describe how context-aware service-oriented applications deployment can be achieved. We base our approach on a service graph colouring process. Therefore, we represent an application as a bi-dimensional dynamic graph with services and components dependencies. The colouring decision provides an optimal deployment configuration of the application in a given context. Context listening mechanisms capture changes and propagate recolouring and redeployment processes. AxSeL (A conteXtual Service Loader) - an OSGi prototype - illustrates and validates the approach by a real use case.*

MOTS-CLÉS : *Intergiciel, applications orientées services, terminaux contraints, graphe de déploiement*

KEYWORDS: *Middleware, service-oriented applications, constraints devices, deployment graph*

1. Introduction

L'expansion de l'utilisation des terminaux mobiles offre une multitude de fonctionnalités aux utilisateurs. En effet, ces plates-formes rendent accessibles les services, applications et fonctionnalités n'importe où et n'importe quand. Par ailleurs, la conscience du contexte est un requis majeur dans les environnements intelligents principalement caractérisés par un constant changement dû à la mobilité des utilisateurs et à l'apparition de nouveaux périphériques électroniques constituant de nouveaux dépôts de services. Par conséquent, l'adaptation dynamique à ces changements est essentielle et permet de respecter les contraintes matérielles mais aussi les préférences de l'utilisateur qui est au centre des environnements omniprésents et intelligents.

Bien que la conscience du contexte soit fondamentale, les applications usuelles ne sont pas conçues pour s'adapter à un contexte donné. Elles nécessitent souvent l'installation de la totalité de leurs composants sans appliquer une quelconque décision. En partant de cette problématique, nous proposons d'intégrer la notion de contexte dans le déploiement des applications afin de les adapter lors du déploiement initial et de l'exécution, aux diverses contraintes de l'environnement omniprésent.

Dans cet article, nous décrivons AxSeL (*A conteXtual Service Loader*) une architecture de déploiement d'applications orientées services. AxSeL considère les applications orientées services composées par des éléments nécessaires (prioritaires) et d'autres optionnels. Nous proposons une adaptation structurelle de l'application au moment du déploiement et lors de l'exécution afin de rompre l'aspect figé des applications usuelles, mais aussi afin de donner la possibilité de toujours fournir la fonctionnalité dans des conditions contextuelles différentes.

Une application AxSeL est représentée sous la forme d'un graphe de dépendances global, bidimensionnel et flexible où les nœuds représentent les services et les composants. Ce graphe réunit deux niveaux : le niveau de déploiement (composants) et le niveau d'exécution (services). Nous assignons aux nœuds du graphe des données contextuelles et procédons à une décision de déploiement basée sur la coloration du graphe de dépendances sous certaines contraintes.

La coloration aboutit à choisir les nœuds à charger pouvant tenir dans les ressources disponibles du dispositif. Cette décision est réalisée grâce à des algorithmes de coloration à critères multiples qui rendent possibles la modélisation et la prise en compte des données provenant de l'environnement, nombreuses et non exhaustives. En fonction des données contextuelles AxSeL fournit une configuration instantanée de l'application à charger. Cette configuration n'est rien d'autre qu'une représentation dynamique et colorée parmi tant d'autres possibles. AxSeL propose également l'usage de plusieurs stratégies de déploiement selon le contexte considéré. Cet aspect dynamique confère à AxSeL la capacité d'adapter les applications à un contexte qui peut être hautement évolutif.

Notre approche propose trois contributions principales :

– la représentation d'une application sous la forme d'un graphe de dépendances bidimensionnel de services et de composants,

– un déploiement contextuel basé sur une coloration du graphe de dépendances entre services et composants avec prise en compte de plusieurs critères aboutissant à des configurations différentes d'une application donnée,

– une adaptation contextuelle au cours de l'exécution de l'application à travers une remise en cause de la coloration initiale suite à des notifications d'événements contextuels.

L'architecture est validée par un prototype OSGi appliqué sur un cas d'utilisation simple et illustrant. Des mesures de performances démontrent le surcoût observé lors de l'ajout de la couche d'optimisation.

Dans la suite de cet article, nous présentons un état de l'art en section 2. Une vue générale de l'architecture AxSeL ainsi que les modèles de conception proposés sont présentés dans la section 3. Ensuite, en section 4, nous détaillons les mécanismes mis en œuvre pour contextualiser le déploiement. Dans la section 5, nous présentons un prototype d'AxSeL et un cas d'utilisation réel. Enfin dans la section 6 nous concluons et donnons les perspectives de notre travail.

2. État de l'art : architectures adaptatives de déploiement de composants

Le déploiement logiciel (Carzaniga *et al.*, 1998) est le processus de mise en œuvre d'une application sur une machine hôte. Il se compose des étapes suivantes : lancement, installation, activation, désactivation, adaptation, mise à jour, désinstallation, suppression. Les architectures présentées traitent du déploiement de composants dans des environnements distribués, problème connu sous le nom de placement de composants (CPP).

(Hoareau *et al.*, 2008) considèrent le problème de fragmentation du réseau causé par la volatilité des hôtes dans les environnements omniprésents. Les auteurs présentent un déploiement de composants dirigé par les contraintes dans des réseaux dynamiques, en considérant un modèle hiérarchique déployé par propagation sur un ensemble de périphériques. Ils portent leur intérêt sur deux phases du déploiement : l'installation et l'activation. Les contraintes sur les ressources et les composants sont exprimées grâce à un ADL (*Architecture Description Language*), qui est pris en compte dans un algorithme décisionnel de déploiement par propagation.

(Dearle *et al.*, 2004) se basent également sur un langage déclaratif exprimant les contraintes et proposent un modèle pour le déploiement et la gestion autonome des applications distribuées orientées composants. Les contraintes telles que l'attribution des composants aux hôtes et la topologie de l'interconnexion des composants sont d'abord décrites, ensuite résolues pour trouver une configuration satisfaisant le déploiement souhaité. Un détecteur s'assure continuellement de la validité du déploiement.

Les contraintes abordées peuvent aussi être d'ordre matériel telles que l'énergie d'un terminal ou son espace mémoire. (Kichkaylo *et al.*, 2004b) ciblent l'économie de l'énergie des dispositifs disponibles, et résolvent le CPP au travers d'une approche basée sur l'intelligence artificielle. Ils étendent l'algorithme Sekitei (Kichkaylo *et al.*, 2004a) afin d'offrir un planificateur de déploiement de composants.

SDI (Taconet *et al.*, 2003) considère plutôt la contrainte d'optimisation de l'espace mémoire des dispositifs mobiles. Une architecture facilitant l'installation des applications distribuées sur des terminaux est fournie. Elle considère les applications comme étant un ensemble de composants distribués. SDI prend en compte les ressources disponibles et les capacités des terminaux des utilisateurs, afin de fournir une meilleure adaptation au contexte d'exécution. Lors de l'installation, un composant est soit chargé localement, soit utilisé à distance, et ce afin d'optimiser l'utilisation de l'espace mémoire. La position géographique d'un terminal est aussi prise en compte pour sélectionner les composants appropriés pour une application donnée.

(Poladian *et al.*, 2004) abordent les préférences utilisateurs et présentent un mécanisme d'adaptation des applications orientées services à l'exécution, dans un environnement omniprésent. L'adaptation se base sur la spécification de préférences utilisateurs et prend avantage des ressources matérielles disponibles. Pour réaliser cela, les auteurs font un choix préalable des applications et services pouvant fournir une tâche utilisateur particulière, et procèdent ensuite à l'allocation des ressources matérielles. Enfin, des reconfigurations sont possibles en cas de changement de situation. Un modèle analytique et un algorithme sont proposés afin de permettre une prise de décision de reconfiguration optimale.

(Preuveneers *et al.*, 2007) proposent une architecture de déploiement et d'adaptation des applications aux contraintes matérielles et notamment la batterie et le processeur. L'adaptation proposée est sur deux niveaux : structurel et comportemental. Les composants d'une application sont décrits dans un descripteur de déploiement qui mentionne si ceux-ci sont obligatoires ou optionnels. Une sonde sur l'interface de connexion à internet est mise en place et permet de remonter les données de celle-ci (type, disponibilité), l'application s'adapte ensuite selon la disponibilité de la ressource. Malgré l'adaptation structurelle proposée les auteurs abordent des applications décrites au préalable et ne remettent pas en cause l'extension ou le caractère dynamique de celles-ci.

L'adaptation au contexte peut être réalisée par la génération d'un comportement souhaité au niveau des services. CASM (Park *et al.*, 2005) permet le développement et le prototypage de services conscients du contexte. Cette architecture collecte et interprète les informations contextuelles et fournit aux services l'utilisant des tâches d'exécution en relation avec celles-ci.

SOCAM (Gu *et al.*, 2004) est également une architecture qui permet de construire et de prototyper des services mobiles et conscients du contexte. L'adaptation et la conscience au contexte sont réalisées avec un ensemble de règles prédéfinies déclenchant un comportement souhaité des services en question. La prise de décision du

comportement à avoir est réalisée grâce à un mécanisme de raisonnement sur le modèle du contexte.

Système	Déploiement	Unité	Décision
(Hoareau *et al.*, 2008)	installation et activation	composant	raisonnement sur ADL
(Kichkaylo *et al.*, 2004b)	installation	composant	planificateur de déploiement
(Dearle *et al.*, 2004)	configuration et mise à jour	composant	descripteur de déploiement
(Poladian *et al.*, 2004)	configuration et adaptation	service	modèle et algorithme analytique
(Taconet *et al.*, 2003)	installation	composant	descripteur de déploiement
(Park *et al.*, 2005)	adaptation	service	règles d'inférence
(Gu *et al.*, 2004)	adaptation	service	inférence sur modèle du contexte
(Preuveneers *et al.*, 2007)	adaptation	composant	descripteur de déploiement et algorithme de sélection

Tableau 1. *Tableau comparatif de différentes plates-formes de déploiement contextuel*

Les architectures décrites réalisent du déploiement de composants logiciels sur des plates-formes locales ou distribuées. La phase de déploiement, l'unité de déploiement et la décision prise lors du déploiement varient selon les travaux.

Nous axons la comparaison faite au tableau 1 sur ces trois critères. Le critère de décision renseigne sur le mécanisme considéré pour choisir les composants à déployer sur un ensemble de machines hôtes selon les contraintes de celles-ci. Certaines décisions sont le fruit d'un raisonnement logique établi selon des règles prédéfinies au préalable, alors que d'autres sont dictées par des descripteurs de déploiement ou des mécanismes s'appuyant sur l'intelligence artificielle.

Ces architectures traitent du problème de placement des composants sur des machines hôtes dans un environnement donné en considérant aussi bien les contraintes des composants que celles des machines, mais ne proposent pas cependant des stratégies dynamiques qui pourraient s'adapter selon le contexte. Les critères considérés sont définis d'une manière statique et les applications à déployer ne sont pas remises en question en cas de changement dans les dépôts de composants ou services.

AxSeL opère au niveau local (terminal) et propose selon les contraintes rencontrées (matérielles et logicielles) une configuration instantanée de l'application à charger. Celle-ci représente une vue dynamique de l'application variant selon le contexte

d'exécution (capacité du terminal, services disponibles). La composition de l'application est mise à jour en cas de changement dans le dépôt des services.

3. AxSeL, une architecture de déploiement contextuel de services

Dans cette section, nous présentons d'abord une vue générale illustrant le comportement d'AxSeL. Ensuite, nous proposons les modèles de conception d'application et de contexte sur lesquels s'appuie notre architecture.

3.1. *Vue générale*

Dans un environnement omniprésent, les services sont hébergés sur des dépôts distants ou locaux décrits par des descripteurs. Les descripteurs de services incluent des données sur les services (dépendances, emplacements, taille mémoire). Les terminaux mobiles accèdent à ces dépôts à travers leur descripteur de services, et y puisent les fonctionnalités requises. Une fois trouvé, le composant implantant le service est chargé localement. AxSeL effectue la prise en compte du contexte par une vue unifiée des différents dépôts de services au travers d'un unique descripteur. Le descripteur offre aux dispositifs une vision contextuelle des services disponibles dans l'environnement. La figure 1 présente une vue générale de l'architecture.

Pour effectuer le chargement de services, AxSeL suit quatre étapes :

1) l'*extraction des dépendances* est le processus par lequel les dépendances d'un service sont extraites à partir d'un descripteur de services. Ce processus aboutit à un graphe de dépendances incluant les propriétés du service et celles de ses dépendances. Les nœuds du graphe sont les services et composants, et les arcs sont les dépendances extraites. Les services et composants sont décrits dans le dépôt d'une manière unitaire. Nous réalisons donc l'extraction récursivement en parcourant les dépendances de chaque nœud et en les incluant au fur et à mesure dans le graphe de dépendances globales. Lorsque l'ensemble des dépendances est tracé, la construction du graphe est alors finie (descripteur commun de services, figure 1),

2) la *décision du chargement* est l'opération de parcours du graphe de dépendances du service en confrontant le contexte requis par celui-ci avec le contexte fourni par les terminaux mobiles. Pour chaque service du graphe de dépendances, une décision de chargement ou non, selon les contraintes, est prise (étape a, figure 1),

3) le *chargement/déchargement* est l'étape qui applique la décision précédente en effectuant techniquement le téléchargement des composants, réalisant son installation, le démarrant et publiant le service sur la plate-forme de services (étape b, figure 1).

Le déchargement est également possible, par exemple après modification du contexte, et correspond à une désinstallation du composant de la plate-forme (étape c, figure 1). Un déchargement peut s'accompagner, si le service est indispensable ou

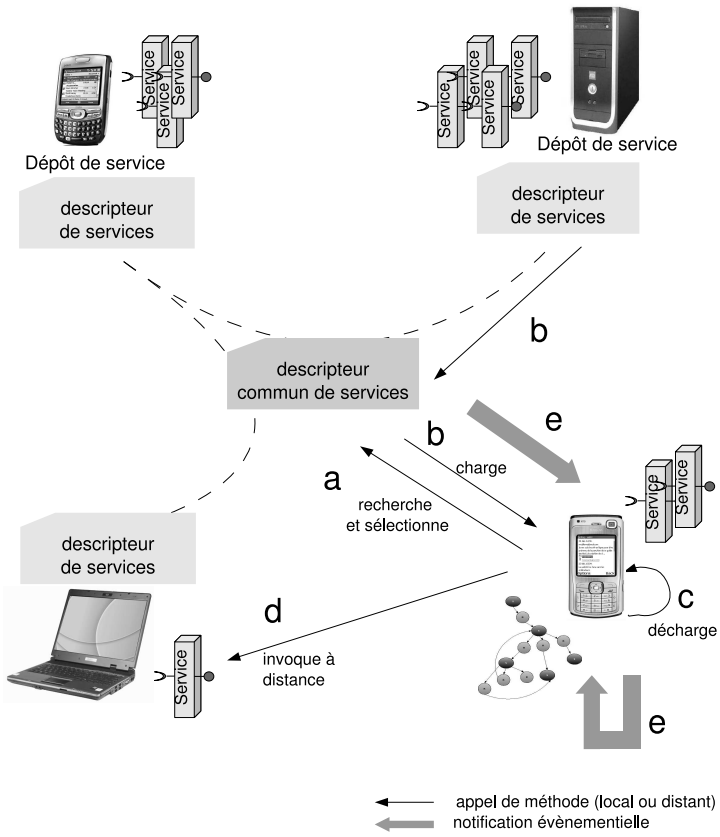

Figure 1. *Comportement de la plate-forme AxSeL : (a) le dispositif recherche et décide du service à charger, (b) il le charge localement, ainsi que ses dépendances, à partir du dépôt source du service, (c) en cas de changement (notifications provenant du contexte), le service peut être déchargé et (d) les appels peuvent éventuellement être redirigés vers un service distant*

en cours d'utilisation, d'une redirection des appels vers un service distant (étape d, figure 1),

4) l'*adaptation contextuelle* est réalisée lorsque des changements sont observés dans le contexte (étape e, figure 1). Ces notifications peuvent provenir du dispositif lui-même à partir des moniteurs sur le matériel (diminution ou augmentation de la mémoire, activation ou désactivation d'interfaces réseau, etc.) et également à partir des descripteurs de dépôt avec des notifications de nouvelles versions, des ajouts ou des suppressions d'un ou plusieurs services ou composants. Cette étape déclenche une nouvelle prise de décision avec comme conséquences d'éventuels chargements/ déchargements (étapes a, b, c, figure 1).

Dans cet article, nous présenterons essentiellement l'étape 4 qui consiste en l'écoute du contexte et l'adaptation du chargement selon les événements recueillis. Les autres étapes sont détaillées dans un autre article (Ben Hamida *et al.*, 2008).

3.2. *Vers une modélisation d'applications contextuelles orientées services*

Dans ce qui suit, nous détaillons les modèles proposés dans AxSeL pour construire une application. Dans un premier temps, la section 3.2.1 décrit les briques de base : services, composants et dépendances nécessaires pour élaborer la représentation plus complexe d'une application comme un graphe bidimensionnel de dépendances. Dans un second temps, nous décrivons, dans la section 3.2.2, le modèle de contexte et la façon dont nous enrichissons le graphe pour sa prise en compte.

3.2.1. *D'un dépôt de services à un graphe de dépendances*

3.2.1.1. Service, composant et dépendance

Service, 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 dynamique des fonctionnalités.

Dépendance La notion de dépendance entre services et composants est une notion fondamentale pour les architectures orientées services car elle permet d'importer de nouvelles fonctionnalités sans avoir à les implanter. Selon le moment dans le cycle de vie du service ou du composant, nous distinguons deux types de dépendances : dépendances de déploiement et dépendances d'exécution. Lors du déploiement un composant peut dépendre de zéro ou de plusieurs autres composants. Les dépendances de services apparaissent à l'exécution, ainsi, un service peut dépendre de zéro ou plusieurs services et obligatoirement du composant qui l'implante. Les dépendances entre services et composants sont renseignées dans les descripteurs de dépôts.

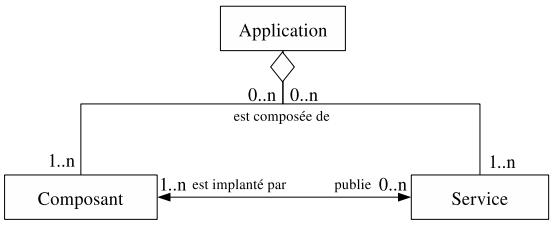

Figure 2. *Représentation d'une application orientée services*

Application Dans AxSeL, nous considérons une application comme l'ensemble des services et des composants qui l'implantent (figure 2). Nous n'avons pas consi-

déré les dépendances comme des éléments logiciels à part entière (comme par exemple dans (Li *et al.*, 2008)) pour ne pas restreindre les possibilités de communications directes entre services. L'implantation d'une approche à connecteur avec notre modèle est toutefois possible en spécialisant des composants pour l'interconnexion et en les proposant sous forme de composants composés (Ibrahim *et al.*, 2010).

3.2.1.2. Graphe de dépendances de services et de composants

Dans un environnement omniprésent, chaque dispositif électronique est une source éventuelle de services et de composants. La mobilité des utilisateurs implique une découverte constante de nouveaux dépôts alimentés par de nouveaux fournisseurs de services. Ceci amène des aspects multidépôts et multifournisseurs qu'il est pertinent d'intégrer pour optimiser les décisions de déploiement.

Afin de prendre cette décision de manière optimale, AxSeL se base sur une vue globale et flexible de l'application à charger. D'abord, la vue globale est construite à partir des descripteurs de dépôts en intégrant aussi bien les paramètres de déploiement et d'exécution, que les aspects multidépôts et multifournisseurs. L'ensemble des services, des composants et des dépendances d'une application orientée services est représenté sous la forme d'un graphe de dépendances pondéré, orienté et bidimensionnel. De cette manière nous bénéficions des approches déjà implémentées pour les graphes, mais également de l'expressivité apportée en termes de représentation des dépendances, d'assignation de poids sur les nœuds et les arcs.

Ensuite, pour permettre l'adaptation de la structure de l'application selon les contraintes contextuelles, il est nécessaire que cette vue globale soit flexible permettant ainsi l'extensibilité et la dynamique des éléments la constituant. Ceci est réalisé grâce à un ensemble d'opérations sur le graphe de dépendances de l'application.

Modélisation du graphe de dépendances Un graphe d'applications AxSeL comporte les éléments suivants, illustrés dans la figure 3 :

– *point d'entrée* : dans le cas des applications orientées services, nous distinguons le service ou le composant d'entrée des autres de la même application. Le point d'entrée correspond au service ou composant qu'il va falloir déployer et exécuter en premier, et qui entraîne également le déploiement de ses dépendances. AxSeL fournit un graphe extensible qui supporte l'existence de plusieurs points d'entrées,

– *niveau* : le graphe que nous proposons regroupe dans une même vue deux concepts différents : les services qui font partie des environnements d'exécution et les composants qui appartiennent plutôt aux environnements de déploiement. Dans le graphe, nous distinguons ces deux niveaux : le niveau d'exécution et celui du déploiement. Les nœuds correspondant aux services sont ajoutés au niveau d'exécution et ceux correspondant aux composants au niveau de déploiement. La représentation globale nous fournit une manière simple de gérer les deux niveaux simultanément. Par ailleurs, la distinction entre les niveaux permet d'assigner aux éléments appartenant à chacun les propriétés qui leurs sont intrinsèques,

– *nœuds* : les services et les composants constituant une application représentent des ressources logicielles. Ils possèdent de la même manière des propriétés fonctionnelles et non fonctionnelles qu'il est pertinent de représenter en vue de prendre une décision de déploiement. Ainsi, nous représentons communément les services et les composants par des nœuds du graphe. La distinction entre les nœuds services et les nœuds composants se fait par leur type et leur appartenance à un niveau différent. Chaque nœud est caractérisé par sa désignation, son identifiant et la ressource qu'il représente. L'expressivité du graphe permet également d'associer un contexte à chaque nœud service ou composant,

– *arcs* : nous représentons les dépendances dans le graphe par des arcs. Par analogie à une dépendance un arc lie deux nœuds. Un arc est caractérisé par le nœud de départ et les nœuds de destination. Nous distinguons entre dépendances d'exécution et dépendances de déploiement. Les services dépendent des composants qui les implantent, les composants peuvent importer d'autres composants. Enfin, les services importent d'autres services. Un arc est orienté et va d'un service à un autre, ou d'un composant à un autre ou d'un composant à un service. Les arcs possèdent des annotations spécifiques et un contexte.

– *opérateurs logiques* : dans un même dépôt, il est possible de trouver des représentations, des versions et des descriptions contextuelles différentes d'un même composant ou service offert par des fournisseurs différents. Pour intégrer cet aspect multi-fournisseur, AxSeL rompt le déterminisme des dépendances et offre un choix multiple entre plusieurs chemins possibles. Pour représenter l'alternative entre une dépendance et une autre, nous étendons le graphe avec les opérateurs logiques ET et OU entre les arcs. Un arc portant l'opérateur ET renseigne sur la nécessité de charger les nœuds de cette dépendance (arc avec un nœud source et un seul un nœud destination). Un arc portant l'opérateur OU permet la multiplicité des choix (arc avec un nœud source et plusieurs nœuds destination au choix). Le choix des services et composants à déployer est réalisé automatiquement et par adéquation aux contraintes contextuelles.

Flexibilité du graphe de dépendances Les descriptions des applications à déployer sont puisées dans des descripteurs de dépôts. Cependant, ces données peuvent changer lors de la modification des dépôts. La modification peut consister en l'apparition ou la disparition de nouveaux fournisseurs de services ou de composants ou la livraison de nouvelles mises à jour. Il est pertinent de répercuter dynamiquement les modifications observées sur la représentation de l'application pour garder, d'une part, une vue actualisée et d'autre part, pour améliorer la prise de décision du chargement en intégrant de nouveaux paramètres. Pour offrir une représentation supportant la dynamique de l'environnement un ensemble d'opérations de manipulation du graphe est fourni :

– *opérations sur les nœuds* : elles sont relatives aux éventuelles mises à jour réalisées dans le dépôt. Elles couvrent l'*ajout*, la *modification* et la *suppression* d'un nœud. Lorsqu'un service ou un composant est ajouté à l'application ceci est pris en compte dynamiquement par l'ajout du nœud correspondant au niveau adéquat selon

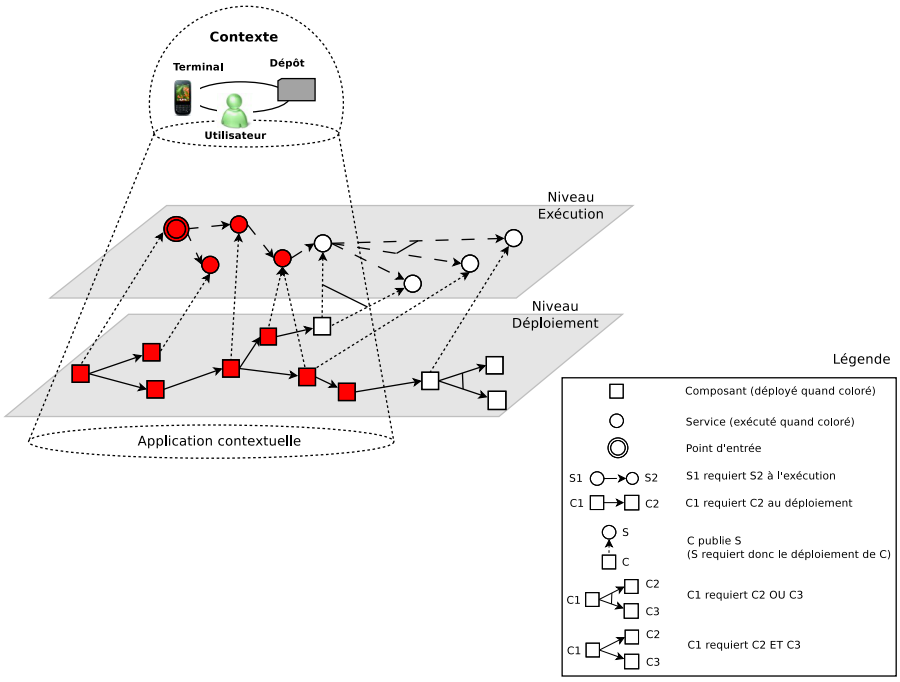

Figure 3. *Modélisation du déploiement contextuel des applications par un graphe bidimensionnel de composants et services*

son type. Cette opération recherche dans le graphe les dépendances du nœud et les relie ensemble. De la même manière lors du retrait d'un service ou d'un composant, cela entraîne la suppression du graphe du nœud correspondant et de ses dépendances,

– *opérations sur les arcs* : elles sont aussi relatives aux mises à jour entraînant la modification de la structure de l'application. Elles couvrent l'*ajout*, la *modification* et la *suppression* d'un arc. Lorsqu'une nouvelle dépendance apparaît dans le dépôt de services composants - par exemple lors d'une nouvelle version d'un composant, celui-ci peut dépendre d'un nouveau composant librairie - celle-ci est répercutée par la création et l'ajout d'un arc entre les deux nœuds correspondants. La modification des donnés contextuelles des arcs est également possible. Enfin lorsqu'une mise à jour du dépôt supprime une dépendance cela entraîne la suppression de l'arc correspondant du graphe,

3.2.2. *D'un graphe de dépendances à un déploiement contextuel d'applications*

La section 3.2.2.1 présente le modèle hiérarchique très classique que nous utilisons. Le point-clef, dans la section 3.2.2.2, est l'intégration de ce contexte par projection sur le graphe bidimensionnel.

3.2.2.1. Modèle hiérarchique du contexte

Le modèle de contexte doit satisfaire les besoins inhérents aux environnements ambiants tels que la limite matérielle des terminaux, l'expressivité des applications et la représentation des données relatives à l'utilisateur. De par leur mobilité et leur petite taille, les terminaux mobiles sont contraints en ressources matérielles. La connaissance de l'état actuel d'usage en ressource au sein d'un dispositif est une information pertinente pour prendre une décision optimale de chargement. Ensuite, les applications considérées sont composées de services et de composants qui possèdent eux-mêmes des données caractéristiques contextuelles qu'il est pertinent d'intégrer dans le processus de décision. Enfin, l'adéquation aux préférences utilisateurs est un objectif important dans les environnements ambiants pour fournir des services personnalisés. AxSeL considère donc ces trois sources d'informations contextuelles : le terminal, le dépôt de services et de composants et l'utilisateur. Ceux-ci sont décrits ci-dessous.

Plusieurs modèles de représentation du contexte ont été proposés dans des approches intergicielles contextuelles (Baldauf *et al.*, 2007). Dans AxSeL, nous réutilisons un modèle hiérarchique de classes (Rossi *et al.*, 2005) où chaque source d'information représentée dans la figure 4 définit ses éléments de données, un ensemble de méthodes et une interface de manipulation exportant les méthodes d'affectation et de récupération des données.

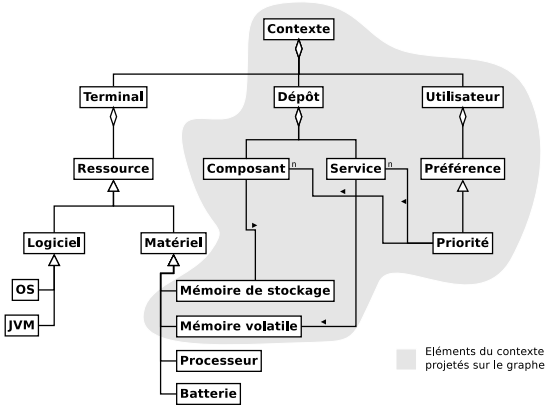

Figure 4. *Modèle hiérarchique du contexte*

Le terminal Un terminal ou dispositif est une machine hôte destinée à l'usage de l'utilisateur. Un terminal est composé d'un ensemble de ressources matérielles et logicielles. Les ressources logicielles décrivent la plate-forme logicielle d'exécution sous-jacente : le système d'exploitation, les librairies, les machines virtuelles installées. Dans notre contexte, cette plate-forme logicielle est fixée (Linux+JVM+OSGi, cf. section 5) et ne sera donc pas sujette à variations. Les ressources matérielles - le processeur, le disque dur, la mémoire et la batterie - sont celles supportées par le ter-

minal mais aussi celles qui sont réellement mises à disposition de l'utilisateur ou de la plate-forme logicielle. Par exemple, un terminal peut avoir 256 Mo de RAM et ne laisser à la disposition de la plate-forme d'exécution que 100 Mo. Dans cet article, nous prenons en compte uniquement les contraintes sur les mémoires disponibles virtuelle et de stockage.

Le dépôt de services et de composants Les dépôts publient une description des composants et services qu'ils mettent à disposition. Cette description inclut des caractéristiques techniques sur ceux-ci, comme la mémoire utilisée. Cette caractéristiques peut être soit ajoutée par le développeur du composant ou service, soit calculée automatiquement par le dépôt, soit calculée et surveillée par notre plate-forme AxSeL. Composant ou service peuvent être rattachés à des éléments différents du contexte (cf. figure 4) : composant rattaché à la mémoire de stockage pour le déploiement, service rattaché à la mémoire volatile pour l'exécution. En cas d'absence totale de description, nous supposons que la quantité mémoire estimée nécessaire est égale à la taille de stockage de l'unité de déploiement même (l'archive jar par exemple).

L'utilisateur Le profil utilisateur peut inclure un grand nombre d'informations complexes. Par soucis de clarté et de simplicité, nous considérons un contexte utilisateur simple dans lequel celui-ci renseigne ses préférences applicatives sous forme de priorité (cf. figure 4). Ainsi, l'utilisateur peut renseigner une liste de composants ou de services qu'il désire avoir sur son terminal en priorité. Cette liste de priorité nous permet de prendre en compte les désirs de l'utilisateur lors du chargement concomitant de plusieurs composants et services.

3.2.2.2. Projection du contexte sur le graphe bidimensionnel

Comme le montre la figure 3, un point-clef pour la prise en compte du contexte est que celui-ci est projeté sur le graphe afin d'obtenir une vue contextuelle du déploiement des composants et services. Tout les éléments du contexte ne sont pas projetés sur le graphe (cf. figure 4) : les aspects contextuels liés aux composants et services, comme leurs priorités définies par l'utilisateur ou leurs occupations mémoire surveillées sur le terminal, sont projetés sur le graphe ; d'autres éléments contextuels, comme par exemple l'occupation mémoire globale ou la batterie du terminal, restent externalisés et surveillés par le terminal.

Cette surveillance du contexte se base sur des mécanismes de capture des événements provenant des sources définies d'abord, ensuite sur le traitement des informations collectées et enfin par un mécanisme de notification.

Afin d'éviter qu'AxSeL ne gère des données hétérogènes recueillies à partir de chaque source d'information nous utilisons un mécanisme qui masque les sondes. En effet, pour chaque source nous associons un adaptateur (*Wrapper*) permettant à travers l'interaction avec des API spécifiques de récupérer les données à partir des sondes. L'adaptateur fournit des méthodes qui permettent de récupérer et traiter les données

courantes. Pour les ressources matérielles, les API utilisées sont celles fournies par la plate-forme d'exécution, dans notre cas nous reposons sur une machine virtuelle permettant la récupération des valeurs de la mémoire. Les données relatives aux services et aux composants sont extraites à partir de l'API correspondante du dépôt.

Au moment de la projection de contexte, chaque nœud du graphe - composant ou service - enregistre alors un mécanisme d'écoute sur les adaptateurs pour être notifié en cas de changement. Il est important de noter que chaque nœud enregistre individuellement des écouteurs sur les éléments du contexte qui les intéressent. Dans notre cas, les composants enregistrent un écouteur sur les adaptateurs de mémoire de stockage ; les services enregistrent un écouteur sur les adaptateurs de mémoire volatile et composants et services s'enregistrent tous sur l'adaptateur de priorité utilisateur. Un service peut très bien enregistrer un écouteur sur l'adaptateur de la batterie et, lorsqu'il reçoit une notification de niveau bas, déclencher un redéploiement sur ce critère.

4. Déploiement contextuel dynamique

En se basant sur les modèles de contexte et d'application préalablement définis, AxSeL propose un déploiement multicritère, adaptatif et extensible. La section 4.1 présente l'algorithme de coloration du graphe déterminant la faisabilité du déploiement. La section 4.2 présente ensuite comment ce déploiement est adapté selon des variations dans le contexte. Enfin la section 4.3.1 présente les mécanismes qui permettent le changement dynamique des stratégies de coloration du graphe.

4.1. *Faisabilité du déploiement*

L'algorithme 2 détaille le processus de prise de décision opéré par AxSeL (les variables utilisées par les algorithmes 2 et 3 sont décrites dans l'algorithme 1). L'évaluation de la faisabilité du déploiement est réalisée d'une manière récursive sur tous les nœuds du graphe de dépendances grâce à la fonction *colourGraph()* qui propage l'application de la stratégie de chargement à l'ensemble des nœuds du graphe. Au niveau de chaque nœud les propriétés non fonctionnelles sont évaluées et comparées aux contraintes de déploiement. Pour réaliser cela nous proposons une fonction glouton qui prend une décision locale à chaque nœud du graphe de dépendances en vue d'obtenir un chemin global à coût optimal. Chaque élément de la liste des nœuds à traiter *totagNodes* est visité. La possibilité d'installation de chaque nœud est évaluée à travers la fonction *isLoadable()* qui compare les critères contextuels des nœuds aux contraintes définies.

Le résultat de cette fonction amène l'algorithme à appliquer un code de couleur pour différencier les nœuds installables des autres. Ainsi, lorsqu'un nœud obéit aux contraintes de chargement il est colorié en rouge et a un statut *Loadable*, sinon il est colorié en blanc et est *Unloadable*. Lorsqu'un nœud est blanc il est enlevé de la liste des nœuds à parcourir, ajouté à la liste des nœuds visités et à la liste des nœuds à traiter

Algorithme 1 variables

print $graph$: le graphe à parcourir
print $taggedNodes$: les nœuds déjà visités (initialisé à vide)
print $totagNodes$: les nœuds à visiter (initialisé avec les points d'entrée)
print $restartNodes$: les nœuds à visiter en premier lors d'un prochain passage (initialisé à vide)
print $totagEdgeNodes$: les nœuds à visiter dans le cas d'un opérateur OU (initialisé à vide)

Algorithme 2 colourGraph()

Require: $taggedNodes$, $totagNodes$, $restartNodes$, $totagEdgeNodes$

```
sort(totagNodes)
node ← totagNodes.getFirstElement()
if node ∈ taggedNodes then
  taggedNodes.remove(node)
else
  if !node.isLoadable() then
    node.setColour(WHITE)
    totagNodes.remove(node)
    taggedNodes.add(node)
    restartNodes.add(node)
  else
    node.setColour(RED)
    totagNodes.remove(node)
    taggedNodes.add(node)
    edges ← node.getEdges()
    while edges.hasElements() do
      if edges.getElement().getTo().size() == 1 then
        totagNodes.add(edges.getElement().getTo())
      else
        totagEdgeNodes ← sort(totagNodes) {cas d'un opérateur OU}
        while edges.getElement().getTo().hasElements() do
          totagEdgeNodes.add(edges.getElement().getTo().getElement())
        end while
        totagNodes.add(totagEdgeNodes.getFirstElement())
      end if
    end while
  end if
end if
colourGraph()
```

en premier lors des prochains passages. Grâce à cette technique nous le mettons en priorité par rapport aux nouveaux nœuds. Lorsqu'un nœud est rouge il est également enlevé de la liste des nœuds à traiter et ajouté à la liste des éléments parcourus. Ensuite, nous procédons au parcours de leurs dépendances. Ainsi, nous parcourons les nœuds destinations de chaque arc et l'ajoutons après tri à la liste des nœuds à traiter.

Lorsqu'un arc portant l'opérateur logique OR est rencontré l'algorithme peut choisir aléatoirement son chemin parmi les options qui se présentent à lui ou opérer une évaluation sur les critères non fonctionnels des différents chemins. L'algorithme procède d'une manière récursive jusqu'au traitement de tous les nœuds de la liste *totagNodes*. Cette liste est obligatoirement finie car elle recense d'une manière unique les nœuds à traiter qui sont également puisés dans une liste finie.

Une décision découle de la couleur attribuée à chaque nœud. Si un nœud respecte les contraintes, alors il est installable (couleur rouge) et sera installé sur la plateforme. Sinon il ne l'est pas (couleur blanche). Dans le cas échéant où le nœud ne peut être chargé il est déclaré non installable (couleur blanche). Dans le cas où aucune des contraintes contextuelles n'est respectée, l'installation des composants et services d'une application n'est pas possible, l'algorithme aboutit à un résultat nul n'entraînant aucun chargement. L'évaluation de la faisabilité du déploiement est multicritère et supporte l'ajout dynamique de nouveaux critères contextuels, cela ne nécessite pas de configuration particulière.

4.2. *Adaptation du déploiement initial selon les variations du contexte*

Compte tenu de l'hétérogénéité des sources et des événements qui peuvent prendre place, AxSeL déploie un ensemble de mécanismes d'écoute pour capturer les événements du contexte. Le gestionnaire du contexte illustré dans la figure 5 est chargé de la collecte des événements provenant des écouteurs placés sur le dépôt et le terminal. Il est également possible de déployer des écouteurs de découverte des terminaux environnants grâce à des protocoles de découverte.

Sur notification des événements de changement, un ensemble d'actions peuvent être entreprises. Les sources et événements considérés sont récapitulés dans le tableau 2.

Sources	Événements
Dépôt/Graphe	Ajout/Suppression/Modification d'un nœud
Terminal	Libération/Occupation de la mémoire
Utilisateur	Modification de ses préférences (Priorités)

Tableau 2. *Sources et événements considérés*

Les actions correspondantes sont effectuées par un ensemble d'algorithmes détaillés dans les sections suivantes.

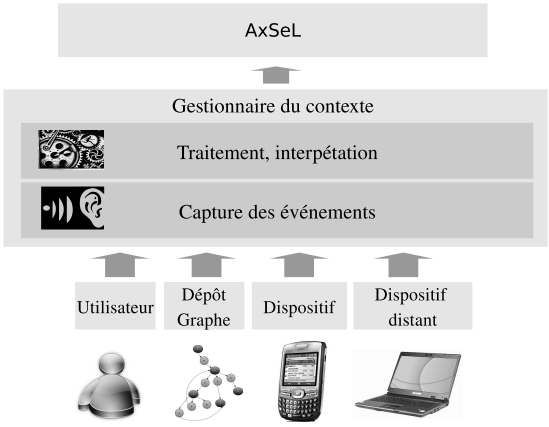

Figure 5. *Gestionnaire du contexte*

4.2.1. *Quand le dépôt change*

L'algorithme 3 est appliqué sur notification des événements de changement du dépôt de services. Les actions varient selon les changements notifiées :

– *ajout d'un nœud* : lorsqu'un service est rajouté au dépôt et par conséquent au descripteur du dépôt, il est ajouté dynamiquement au graphe de dépendances de services tracé par AxSeL. Lorsque le nœud ajouté est un service il est placé au niveau service *ServiceLayer*, puis relié à ses dépendances. Il en est de même dans le cas où le nouveau nœud représente un composant, il est ajouté au niveau des composants *ComponentLayer*. Le nouveau nœud est ajouté à la liste des nœuds à colorier en premier *restartNodes*. La recoloration prend en compte les contraintes contextuelles prédéfinies ainsi que les critères du nouveau nœud,

– *modification des propriétés d'un nœud* : les données contextuelles relatives aux nœuds peuvent subir des changements en raison d'une nouvelle livraison des services ou autre. Lorsqu'un nœud est modifié, dans le cas où il existe dans le graphe, nous modifions ses critères contextuels et vérifions sa couleur, si celle-ci est blanche, il est ajouté à la liste à colorier en premier et une recoloration du graphe est lancée. Sinon, s'il est rouge mais qu'il n'est plus installable nous lui attribuons la couleur blanche et décolorons récursivement ses dépendances,

– *suppression d'un nœud* : elle correspond à la décoloration du nœud en question ainsi que ses dépendances immédiates pour sa non-adéquation aux contraintes de chargement ou sa disparition du contexte. Pour ce faire, nous désignons le niveau auquel il appartient (service ou composant). Une fois positionné sur ce nœud l'algorithme lance une décoloration récursive *uncolor(List)* des dépendances de ce nœud en prenant soin de ne pas décolorer ceux qui sont impliqués dans d'autres dépendances. Un nœud traité n'est pas repris en compte. Lorsqu'un nœud est supprimé les ressources matérielles qu'il exploitait sont libérées.

Algorithme 3 adapt(event e)

Require: $graph, totagNodes, restartNodes$

```
node ← e.getSourceNode()
if e.getType() == NewNode then
  graph.add(node, nodeLayer)
  node.linkdependencies()
  restartNodes.add(node)
  totagNodes.add(node)
  colourGraph()
else
  if e.getType() == ChangeNode then
    if node ∈ graph and node.Colour == WHITE then
      Change node properties by new ones
      restartNodes.add(node)
      colourGraph()
    else if node ∈ graph and node.Colour == RED then
      Change nodei properties by new ones
      if !node.isLoadable() then
        nodei.Colour(WHITE)
        recurseUnColour(children(nodei))
        colourGraph()
      else if node.isLoadable() then
        Change nodei properties by new ones
      end if
    end if
  end if
else if e.getType() == RemoveNode then
  node.Colour(WHITE)
  recurseUnColour(node.getDependency())
end if
```

4.2.2. *Quand le terminal et les préférences utilisateur changent*

Les changements observés au niveau du terminal ou des préférences utilisateurs entraînent des mécanismes similaires au sein d'AxSeL. En effet, dans les deux cas, cela entraîne une nouvelle prise de décision basée sur des paramètres contextuels mis à jour tels qu'une nouvelle quantité mémoire ou une liste de préférences modifiée.

4.2.2.1. Dispositif

La quantité de la mémoire virtuelle disponible sur le dispositif est un critère significatif dans l'adaptation du déploiement. Nous prenons en compte celle-ci en plaçant une sonde sur la mémoire disponible dans la machine virtuelle à travers l'API java

Runtime[1]. Nous couplons cela avec un écouteur qui nous renseigne à une fréquence régulière des changements (accroissement ou décroissement) de la quantité de mémoire disponible. Si les changements sont significatifs (plus de 10 %), AxSeL lance une recoloration du graphe en prenant en compte la nouvelle valeur de la mémoire. La recoloration fournit une image contextuelle de l'application selon les contraintes actuelles d'exécution des services.

La plate-forme AxSeL place des écouteurs sur le dispositif et en abonne un à la mémoire de telle sorte que lorsqu'il y a une modification de la mémoire, la plate-forme est notifiée pour adapter le chargement de services. AxSeL se base également sur le design pattern *Adapter* (Gamma *et al.*, 1995) afin de modéliser les variations de la mémoire et de répercuter ces changements sur le déploiement (figure 6).

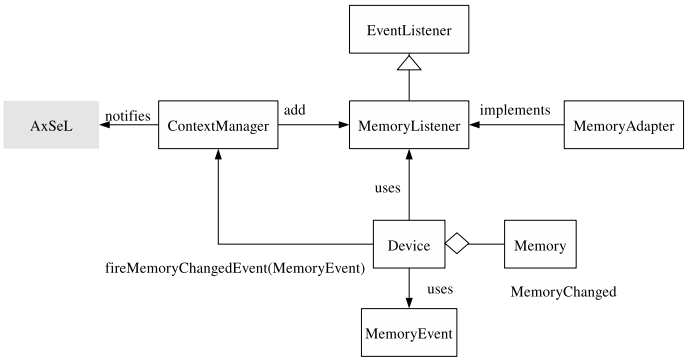

Figure 6. *Écouteur abonné à la mémoire du terminal*

Lors de la réception d'un événement mémoire AxSeL relance une recoloration du graphe de l'application en prenant en compte les nouvelles valeurs enregistrées. Dans le cas de la libération de la mémoire la recoloration peut entraîner l'installation de nouveaux services sur le dispositif. La diminution de la mémoire disponible peut engendrer la désinstallation de certains services.

1. http ://java.sun.com/j2se/1.5.0/docs/api/java/lang/Runtime.html. Nous avons utilisé une sonde interne plutôt qu'externe car une sonde externe au niveau du système d'exploitation permet de mesurer la taille mémoire globale de la machine virtuelle Java pré-allouée au départ mais pas la mémoire allouée lors de l'exécution d'une application à l'intérieur de la machine virtuelle. L'utilisation d'une sonde interne pose également des problèmes, notamment celui du ramasse-miettes qui peut perturber les mesures. Le ramasse-miettes étant impossible à désactiver dans la machine virtuelle utilisée, nous avons particulièrement prêté attention à effectuer les mesures lorsqu'il n'est pas actif, tout simplement en l'activant avant de mesurer.

4.2.2.2. Utilisateur

Les changements des préférences de l'utilisateur influencent sur le déploiement. Lorsque l'utilisateur change sa liste de services prioritaires, AxSeL prend en compte ces modifications et les intègre dans la stratégie de déploiement. Cela entraîne une nouvelle prise de décision et un nouveau parcours du graphe amenant au chargement ou déchargement éventuels de services. Le mécanisme d'adaptation actionné est exactement le même que quand une donnée contextuelle d'un service change, puisque les priorités utilisateur ont été projetés sur le graphe, et l'algorithme 3 est donc appliqué.

4.3. *Stratégies de déploiement extensibles*

4.3.1. *Stratégies de parcours extensibles*

Le parcours d'un graphe de dépendances peut s'effectuer de différentes manières. Dans l'algorithme, toute l'intelligence de ce parcours réside dans la première ligne :

Algorithme 4 colourGraph()

```
{...}
sort(totagNodes)
{...}
```

totagNodes contenant les prochains nœuds à visiter, établir une relation d'ordre sur cet ensemble permet de définir le parcours du graphe. La fonction *sort* établit cette relation et trie les nœuds. Dans AxSeL, cette fonction est implantée dans un objet Java respectant le design pattern *Strategy* (Gamma *et al.*, 1995). Cela permet, en plein parcours du graphe, de changer l'implantation du parcours et de pouvoir répondre à des besoins de parcours en largeur, profondeur, etc. (cf. figure 7).

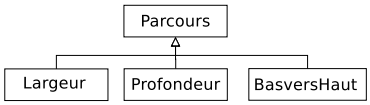

Figure 7. *Stratégies de parcours extensibles*

4.3.2. *Critères de déploiement extensibles*

Dans un environnement omniprésent, plusieurs paramètres contextuels peuvent entrer en jeu pour décider si un composant ou un service est approprié au chargement. Dans l'algorithme, ce choix est réalisé dans la ligne testant *isLoadable()*.

Cette fonction introspecte le nœud pour y trouver les caractéristiques contextuelles qui peuvent lui être utiles pour décider du chargement ou non. Cette fonction peut également interagir avec le gestionnaire du contexte pour compléter ces informations utiles - par exemple, pour calculer que la taille mémoire du nœud est inférieure à

Algorithme 5 colourGraph()

```
{...}
if !node.isLoadable() then
   {...}
end if
{...}
```

la taille restante de mémoire du terminal. Dans AxSeL, comme pour le parcours, cette fonction est implantée dans un objet Java respectant le design pattern *Strategy* (Gamma *et al.*, 1995). Cela permet, en plein parcours du graphe, de changer les critères d'évaluation du chargement et de pouvoir répondre à la dynamique d'approches multicritères (cf. figure 8).

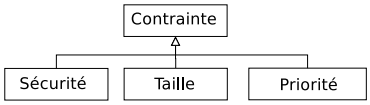

Figure 8. *Critères de parcours extensibles*

Par contre, à la différence du parcours de graphe qui respecte un design pattern *Singleton* (Gamma *et al.*, 1995), l'objet évaluant les critères de chaque nœud ne respecte pas forcément ce design pattern et n'est pas obligatoirement unique. Dans le cas où celui-ci est unique, le terminal respecte donc une politique de chargement multicritère cohérente et optimale pour toutes les applications. Par contre, chaque nœud ou des groupes de nœuds peuvent très bien avoir leur propre fonction d'évaluation du chargement. Cela peut permettre d'avoir une décision très fine et proche des besoins de l'application mais dans ce cas AxSeL ne peut garantir la cohérence et l'optimalité du déploiement. Les tests ci-après sont réalisés avec un parcours *Singleton* multicritère priorité et mémoire.

5. Réalisation d'AxSeL

Le prototype d'AxSeL est développé avec le langage Java. Nous avons utilisé la machine virtuelle Java comme plate-forme d'exécution et la plate-forme Felix (Apache Software Foundation, 2010), implantation de la spécification OSGi (OSGi Alliance, 2010), pour le déploiement et l'exécution des services. Nous faisons correspondre les éléments de notre modèle avec ces plates-formes. Un composant est implanté par un composant OSGi appelé *bundle* (unité de déploiement composée de classes et d'un descripteur appelé *Manifest*). Un service est implanté par un service OSGi représenté par une interface Java qui est enregistré dans le *ServiceRegistry* de Felix afin d'être visible et accessible.

L'ensemble des services compilés de notre intergiciel a une taille mémoire globale de stockage de 65,5 Ko, ce qui lui permet d'être déployé sur des machines contraintes. Le prototype est disponible dans la gforge INRIA[2] sous licence Cecill. Le développement et les tests d'évaluation ont été réalisés sur une machine Dell Lattitude D610 Intel Pentium Microsoft processeur 2,13 Ghz sous le système d'exploitation Linux. Bien que puissant, cet environnement de test a été redimensionné au niveau logiciel en restreignant la machine virtuelle Java aux capacités d'un Slug (Frénot *et al.*, 2010).

Dans cette section, nous présentons un scénario de cas d'utilisation réel et les tests de performances réalisés sur celui-ci. Enfin, nous évaluons le surcoût de l'adaptation lors du passage à l'échelle.

5.1. *Scénario d'un cas d'utilisation réel : une application de visualisation de documents PDF*

Nous illustrons le fonctionnement d'AxSeL à travers une application simple orientée services qui permet la visualisation de document PDF (*Portable Document Format*). Nous avons créé cette application et donc décidé de la granularité de ses services et de ses composants. L'application est composée de :

– un service PDF qui fournit le document en format PDF (S1),

– un service de navigation dans le document PDF (S2), la navigation peut se faire :

- à travers le clavier (keyNavigator) (A), ou bien

- à travers une interface qui s'affiche sur l'écran avec des boutons de navigation et qui facilite la navigation (F) (widgetNavigator).

Bundles	Taille mémoire	Désignation
`pdfkeynavigator-0.0.1.jar`	5135 o	A
`pdfnavigatorservice-0.0.1.jar`	3093 o	B
`pdfserver-0.0.1.jar`	236837 o	C-S1
`pdfservice-0.0.1.jar`	2923 o	D
`pdfviewer-0.0.1.jar`	10250 o	E
`pdfwidgetnavigator-0.0.1.jar`	14102 o	F-S2

Tableau 3. *Composants et services de l'application pdfviewer*

Pour des raisons de lisibilité sur les schémas nous avons assigné à chacun des composants et services de l'application des désignations simplifiées (cf. tableau 3). Les *bundles* C et F exportent respectivement les services S1 et S2.

2. http ://amazones.gforge.inria.fr/

5.2. *Du descripteur XML de dépôt OSGi au graphe de dépendances*

Les services et composants OSGi sont hébergés dans un dépôt. AxSeL exploite le descripteur de dépôt du *Bundle Repository* de Felix ainsi que l'API correspondante fournie par OSGi pour extraire le graphe de dépendances de services et de composants.

5.2.1. *Le descripteur du dépôt*

Le descripteur de dépôt du *Bundle Repository* décrit un ensemble de services en intégrant leurs dépendances en termes de services et de *bundles*. Cependant, il n'intègre pas de notion d'applications et ne décrit pas l'intégralité des dépendances des services d'une application donnée. Il n'inclut pas non plus de données non fonctionnelles sur les composants ou services. Ces limites nous empêchent de réaliser notre objectif de contextualisation du chargement d'une application donnée. Pour ce faire, nous nous basons sur ce même dépôt de composants et services que nous enrichissons avec des critères contextuels pertinents, comme la taille ici.

Le descripteur suivant décrit l'application du cas d'utilisation du visionneur de documents.

```
<repository>
  <resource id='pdfviewer/0.0.1' presentationname='PDF viewer'
            symbolicname='pdfviewer' uri=''>
    <description>PDF document viewer</description>
    <size>10250</size>
    <capability name='bundle'>
      <p n='manifestversion' v='2'/>
      <p n='presentationname' v='PDF viewer'/>
      <p n='symbolicname' v='pdfviewer'/>
      <p n='version' t='version' v='0.0.1'/>
    </capability>
    <require filter='(service=pdfnavigatorservice.DocumentNavigatorService)'..>
      Import Service pdfnavigatorservice.DocumentNavigatorService
    </require>
    <require filter='(service=fr.inria.amazones.pdfservice.PDFServiceIfc)'..>
      Import Service fr.inria.amazones.pdfservice.PDFServiceIfc
    </require>
    <require filter='(&amp;(package=fr.inria.amazones.pdfservice)(version&gt;=0.0.0))'..>
      Import package fr.inria.amazones.pdfservice
    </require>
    <require filter='(&amp;(package=pdfnavigatorservice)(version&gt;=0.0.0))'..>
      Import package pdfnavigatorservice
    </require>
  </resource>
</repository>
```

5.2.2. *l'API Bundle Repository en bref*

Nous exploitons l'API Bundle Repository qui facilite l'extraction des données du descripteur du dépôt à travers un ensemble de classes et d'interfaces :

– *Repository*, décrit le dépôt de services et de bundles, il s'agit de la première entrée du descripteur de dépôt. Le *Repository* inclut plusieurs ressources (*Resource*),

– *Resource*, correspond à un bundle OSGi, une ressource possède un nom, une description, une taille mémoire, des informations sur la licence, le copyright, mais décrit essentiellement ce que le bundle fournit *Capability* et ce qu'il requiert en *Requirement*,

– *Capability*, définit ce qui est fourni par une ressource en service, package et bundle. Une *Capability* est décrite par un type (service, bundle, package), nom, une version, etc.,

– *Requirement* correspond à l'import fait par une ressource d'un package, service ou bundle.

En partant du descripteur de services et de composants OSGi décrit précédemment et à l'aide de l'API Bundle Repository, nous construisons le graphe de dépendances de services et de composants relatifs à l'application à déployer. Les nœuds du graphe sont ensuite enrichis par les critères contextuels ajoutés du descripteur et sur lesquels ils peuvent enregistrer un écouteur.

5.3. *Du graphe d'application au déploiement contextuel*

Sur la base du graphe de dépendances préalablement extrait et des contraintes contextuelles de déploiement, AxSeL procède à la coloration des nœuds du graphe. L'interface graphique d'AxSeL fournit une vue des graphes traités. La figure 9 illustre les éléments de l'application pdfviewer tels qu'extraits à partir du descripteur du dépôt. Le nœud d'entrée du graphe est distingué par un cercle double. Les niveaux de composants et de services sont distingués en niveau 0 et 1. Enfin, l'opérateur logique OU apparaît par une branche se divisant au niveau du choix entre les deux éléments de navigation par clavier (A) ou par interface graphique (F).

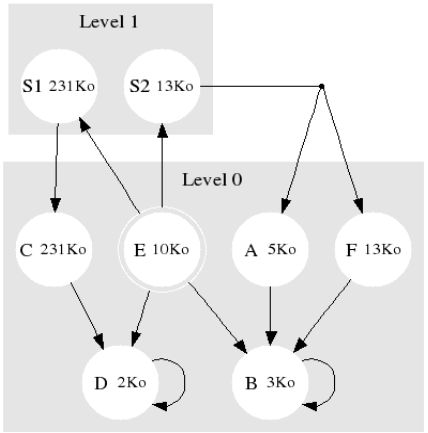

Figure 9. *Graphe de dépendances de l'application avant décision*

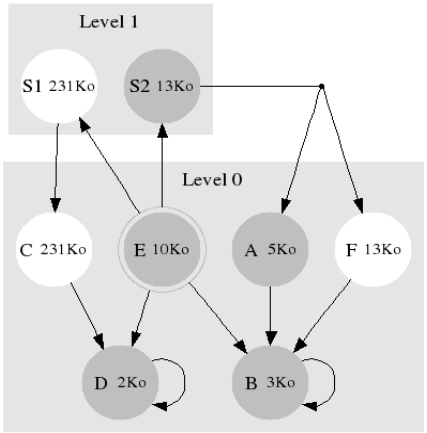

Figure 10. *Graphe de dépendances de l'application après décision*

Lorsqu'un composant importe des paquetages ou des services qui lui sont propres ces dépendances apparaissent dans le descripteur du dépôt, elles sont renseignées dans

notre graphe par des arcs d'autodépendances tels qu'au niveau des composants (D) et (B).

La figure 10 montre le graphe extrait de l'application de visualisation de document PDF après coloration. Pour des contraintes de ressources mémoire, la coloration a abouti à une navigation par clavier plutôt que par interface graphique, car le composant A est moins gourmand en mémoire que F, de même que le service S1 qui n'est pas chargé.

Dans ce cas l'application déployée permet de visualiser le document PDF et de naviguer dedans à travers le clavier (A). Nous évaluons les performances d'AxSeL en temps d'exécution (ms) et mémoire (Ko) durant les phases d'extraction du graphe et de décision (coloration) et installation (tableau 4). L'extraction se fait à partir du descripteur de services et de composants. La décision et l'installation sont respectivement relatives au parcours du graphe et à l'installation des bundles sur la plate-forme OSGi. Il est à noter que le coût d'extraction du graphe n'est pas souvent présent vu qu'il s'agit d'une opération réalisée au départ ou lors de certains événements.

Nous avons considéré une méthodologie pour l'évaluation des performances présentées. Nous réalisons les tests cinq fois de suite et gardons la valeur stabilisée de la consommation en mémoire et en temps d'exécution. Nous avons opté pour cette méthodologie car nous ne voulions pas que les mesures soient faussées par le coût dû au chargement initial des classes à partir de la mémoire de stockage, coût intrinsèque imputable à la machine virtuelle (13 ms et 216 Ko). Cependant, les valeurs se stabilisent autour d'une constante après plusieurs exécutions et ensuite leur écart type ne varie pas de plus de 10 %.

Nombre de nœuds	Extraction	Décision et installation
8	18,75 ms et 194 Ko	13 ms et 381 Ko

Tableau 4. *Performances en temps d'exécution et consommation mémoire pour l'extraction, décision et installation*

5.4. *Altérations du graphe de dépendances*

Le graphe de dépendances évolue selon l'ajout de nouveaux nœuds ou la désinstallation de ceux déjà installés. Lorsqu'un nouveau nœud apparaît il est d'abord ajouté au graphe de dépendances et est ensuite pris en compte dans le processus de coloration et d'installation (figure 11). Le cas illustré montre l'ajout d'un nœud X qui amène une fonctionnalité supplémentaire de visualisation de documents PostScript. Cependant, pour cause de limites de ressources mémoire il n'a pas été coloré et installé. Dans ce cas l'application déployée permet quand même de visualiser le document PDF et de naviguer dedans à l'aide des touches du clavier.

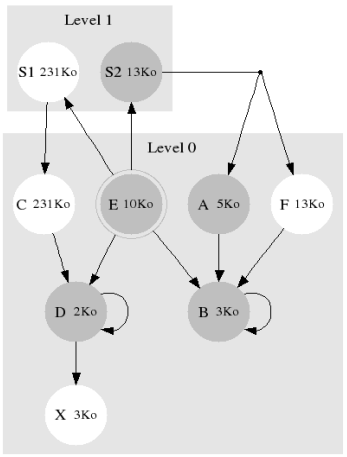

Figure 11. *Graphe de dépendances de l'application : ajout d'un nœud*

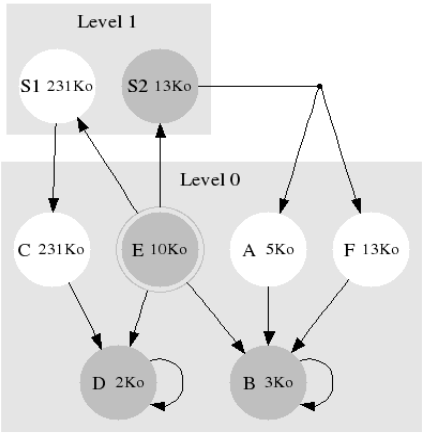

Figure 12. *Graphe de dépendances de l'application : désinstallation d'un nœud*

Lors de la désinstallation d'un nœud donné, dans notre cas le nœud A (figure 12), AxSeL le décolore et le décharge de la plate-forme ainsi que ses dépendances tant qu'elles ne sont pas requises par des nœuds rouges, d'où la décoloration de A et non de B. Dans ce cas l'application fournit un document mais ne permet pas de naviguer dedans, il est toutefois possible à travers la ligne de commande par exemple de l'imprimer ou de l'envoyer par mail.

Nous évaluons les performances en temps et mémoire observées lors de l'adaptation à l'ajout et à la désinstallation d'un nœud (cf. tableau 5). L'adaptation correspond au lancement du processus de coloration partielle sur le graphe de services composants. La désinstallation est relative à celle d'un bundle OSGi de la plate-forme Felix. Notons que le processus d'adaptation est très rapide et ne consomme presque pas de mémoire.

Action	Adaptation
Ajout d'un nœud	1 ms et 2 Ko
Désinstallation d'un nœud	1 ms et 2 Ko

Tableau 5. *Temps d'exécution et consommation mémoire*

Les opérations d'ajout et de désinstallation de nœuds du graphe sont rapides et peu gourmandes en mémoire. Cependant, les performances varient également en fonction du nombre de nœuds à traiter dans le graphe (voir section 5.6).

5.5. *Changement de politiques de déploiement*

Les contextes pouvant évoluer, nous préconisons également le changement en conséquence des politiques de coloration de graphes. Par exemple dans un contexte contraint, les stratégies orientées ressources matérielles sont à prendre en compte en priorité par rapport à d'autre. Selon la politique adoptée le graphe est coloré différemment. Le changement de stratégies de coloration nous permet d'explorer et d'exploiter différemment les graphes de dépendances.

Dans la figure 13, AxSeL adopte une politique de coloration basée sur la taille mémoire et la priorité. La mémoire maximale est à 90 Ko et la priorité minimale est à 3. Les nœuds ayant une priorité supérieure à 3 sont chargés uniquement, ce qui explique que le nœud B avec une priorité 3 n'ait pas été chargé. L'application déployée permet d'avoir le document mais pas de naviguer dedans, il reste toutefois exploitable par lignes de commande par exemple.

Dans la figure 14, nous illustrons le graphe de dépendances sur lequel AxSeL a appliqué une deuxième politique de déploiement qui ne considère que la taille mémoire des nœuds. La mémoire maximale allouée est à 40 Ko. Les nœuds sont d'abord triés dans un ordre croissant selon la taille mémoire. Ensuite, seuls les nœuds peu gourmands en mémoire sont localement chargés. La coloration aboutit au chargement du service S2 et du composant A permettant la navigation dans le document à l'aide du clavier.

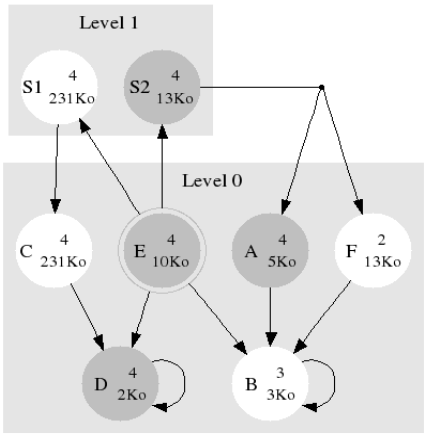

Figure 13. *Graphe de dépendances de l'application : politique orientée taille mémoire et priorité*

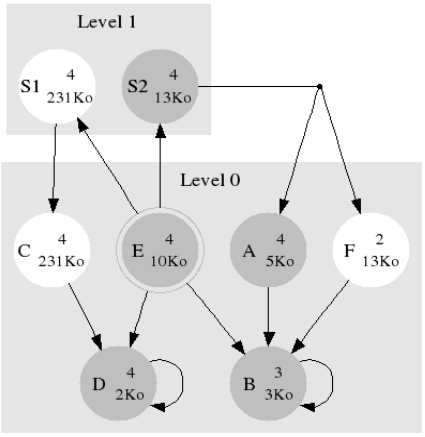

Figure 14. *Graphe de dépendances de l'application : politique orientée mémoire*

Nous évaluons les performances temps et mémoire pour les deux politiques et le changement d'une politique à l'autre.

La politique orientée mémoire uniquement paraît un peu plus performante que celle orientée mémoire et priorité (tableau 6). L'écart est cependant négligeable.

Politique	Décision et installation
Taille mémoire et priorité(A)	16,33 ms et 337 Ko
Taille mémoire(B)	11,33 ms et 312 Ko
A vers B	4 s et 163 Ko
B vers A	5 s et 215 Ko

Tableau 6. *Temps d'exécution et consommation mémoire pour le changement de politique de déploiement*

Le changement de politique semble dans les deux cas plus économique en termes de mémoire que les politiques indépendemment mais plus lent que le cumul de la durée des deux politiques. L'économie de la mémoire est observée en raison du chargement en mémoire des structures du graphe et du fait que les algorithmes de coloration aient un parcours optimisé.

5.6. *Surcoût de l'adaptation lors du passage à l'échelle*

Intégrer l'adaptation contextuelle au sein d'AxSeL a eu un coût supplémentaire en termes de consommation mémoire (Ko) et de temps d'exécution (ms). Nous observons ces coûts pour évaluer le passage à l'échelle d'AxSeL. En effet, nous réalisons des simulations sur des graphes que nous générons automatiquement. Les graphes générés ont pour chaque nœud entre une et cinq dépendances au niveau composant et trois dépendances au niveau service. Les tests ont été réalisés en variant le nombre de nœuds des graphes. Nous observons ensuite les performances relatives à l'exécution d'AxSeL avec et sans adaptation.

La courbe 15 illustre le coût en temps observé avec et sans adaptation. Sans surprise, AxSeL est plus lent avec adaptation que sans. Le temps total d'exécution ne dépasse jamais les 500 ms dans le pire des cas (graphe de 500 nœuds). Ce qui ne pénalise pas l'utilisateur avec un temps d'attente très long. Cela dit, le processus d'adaptation est un processus ponctuel qui n'est pas réalisé constamment. Ce qui signifie que la plate-forme n'aura à supporter ces coûts que de manière ponctuelle.

La courbe 16 représente le coût en termes de mémoire en fonction du nombre de nœuds. La courbe d'usage mémoire sans adaptation a un comportement linéaire allant de 400 Ko de RAM et n'excédant pas 1500 Ko dans le pire des cas (graphe à 500 nœuds). Quant à la courbe d'adaptation nous notons des oscillations et un écart type important de l'ordre de 40 %. Ces variations importantes sont difficiles à anticiper car le processus de recherche des dépendances pour chaque nœud étant plus important, le ramasse-miettes intervient alors inopinément pour nettoyer et désallouer de la mémoire. Les simulations ont toutefois été réalisées en considérant le pire des cas qui

est la recoloration totale du graphe. Dans des cas plus communs nous ne procédons pas à la recoloration du graphe en entier mais à une partie uniquement.

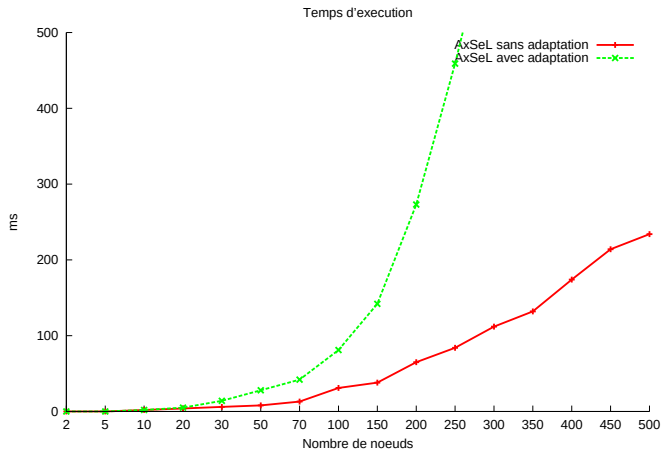

Figure 15. *Évaluation du coût de l'adaptation en temps d'exécution*

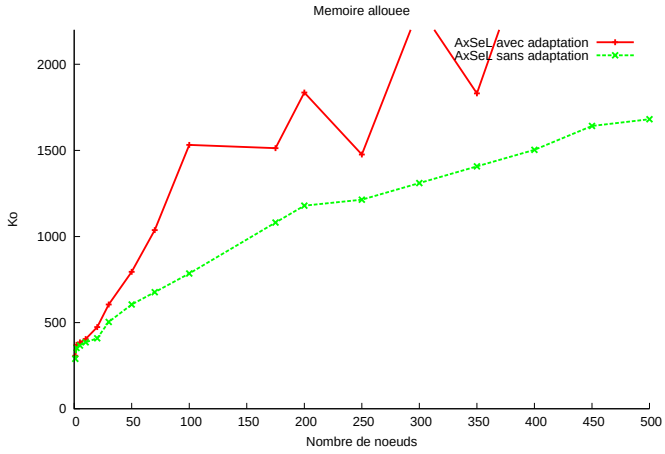

Figure 16. *Évaluation du coût de l'adaptation en utilisation mémoire*

Si nous nous plaçons dans un cas contraint, par exemple un téléphone mobile Nokia 6630 avec 10 M de RAM interne et une carte de 64 M, notre plate-forme AxSeL, avec des applications usuelles avec une centaine de nœuds en moyenne, ne dépassera pas la valeur de 2000 Ko avec ou sans adaptation, soit 2,7 % de la mémoire du Nokia. Ce coût est raisonnable. Après un usage intensif par chargement des applications de calendrier, de la liste des contacts, de l'appareil photo et caméra et d'un navigateur

web, nous avons bien observé un taux important d'occupation de la mémoire de 97 %. Dans le cas de non-utilisation d'AxSeL, l'utilisateur ne peux plus exécuter d'applications et doit en arrêter manuellement une pour exécuter la nouvelle. Dans le cas d'utilisation d'AxSeL, la désinstallation de l'application la plus gourmande s'effectue automatiquement, sans intervention de l'utilisateur, en exécutant la nouvelle application.

6. Conclusions et perspectives

La connaissance du contexte environnant est un requis primordial dans les environnements omniprésents. Outre les contraintes matérielles à prendre en compte, ces environnements mettent l'utilisateur au centre de l'équation en lui fournissant des services personnalisés destinés à l'assister dans son quotidien. Ensuite, l'expansion des périphériques électroniques offrant des services permet à plusieurs fournisseurs d'offrir des services similaires. Intégrer ces aspects multidépôts et multifournisseurs permet d'enrichir le processus de décision du déploiement.

En partant de ce constat, les applications usuelles généralement statiques ne permettent pas de répondre aux exigences de ces environnements, elles n'appliquent pas de décision lors du déploiement et n'intègrent pas les aspects multidépôts et multifournisseurs. Nous proposons une vision différente des applications sous la forme d'un graphe de dépendances global, dynamique et évolutif. Notre hypothèse de départ réside dans le fait que tous les composants d'une application n'ont pas à figurer sur la même machine intégralement. Nous assignons donc des poids sur les arcs et les nœuds pour orienter la décision des nœuds à charger vers une adéquation avec le contexte fourni. Les aspects multifournisseurs de services et multidépôts inhérents aux environnements omniprésents sont représentés à l'aide d'opérateurs logiques ajoutés au graphe.

Dans cet article, nous avons présenté AxSeL une plate-forme pour un déploiement contextuel d'applications orientées services dans des environnements contraints. La contribution d'AxSeL se compose de trois éléments essentiels : un modèle d'application global et flexible basé sur les graphes de dépendances de services et de composants, un modèle du contexte dynamique, un déploiement multicritère basé sur la coloration du graphe de dépendances de services et une composante d'adaptation contextuelle intégrant les possibilités d'utiliser plusieurs stratégies de décision et de changer de stratégies en cas de besoin lors de notifications du contexte à l'exécution.

A travers le prototype réalisé nous avons validé et pu démontrer la faisabilité de notre modèle conceptuel et évaluer les performances de notre système sur un cas réel et sur des simulations pour évaluer le coût de l'adaptation lors d'un passage à l'échelle.

Dans des travaux futurs, plusieurs points d'AxSeL peuvent être améliorés. D'abord, inclure la notion de hiérarchie entre contraintes pourrait être intéressant pour résoudre les conflits qui peuvent subvenir lors d'un choix entre deux politiques d'adaptation ou entre critères de coloration ou lors de l'assignation des poids sur les arcs. Une

hiérarchie pourrait définir par exemple qu'une contrainte est forte, faible, ou obligatoire (Borning *et al.*, 1992).

Ensuite, la composante de dynamicité de remplacement entre les politiques d'adaptation dans AxSeL peut être mise en place pour permettre de varier le comportement du déploiement selon l'utilité perçue. Nous pensons l'implanter à l'aide du design pattern *Observer* (Gamma *et al.*, 1995), mais également en implantant des paradigmes empruntés au domaine économique tels que les fonctions d'utilité permettant le choix entre une politique et une autre selon les contextes d'exécution.

Enfin, nous avons modélisé le contexte d'une manière simple, mais le recours à des représentations plus évoluées telles celles basées sur les ontologies est possible. Le raisonnement sur les informations véhiculées par les ontologies demeure cependant supporté par des raisonneurs (Jena, Axis) non adaptés aux ressources contraintes, ce qui nous empêche pour le moment de les utiliser. Des travaux tels que ceux de (Preuveneers *et al.*, 2008) qui proposent un raisonnement sur un ADL sémantique dans des environnements contraints sont des pistes à explorer et intégrer.

7. Bibliographie

Apache Software Foundation, The Felix Project, 2010. Version 3.0.3, http ://felix.apache.org/.

Baldauf M., Dustdar S., Rosenberg F., « A survey on context-aware systems », *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, n° 4, p. 263-277, 2007.

Ben Hamida A., Le Mouël F., Frénot S., Ben Ahmed M., « A Graph-based Approach for Contextual Service Loading in Pervasive Environments », *10th International Symposium on Distributed Objects and Applications (DOA 2008)*, vol. 5331 of *Lecture Notes in Computer Science*, Monterrey, Mexico, p. 589-606, November, 2008.

Borning A., Freeman-Benson B., Wilson M., « Constraint Hierarchies », *LISP and Symbolic Computation*, vol. 5, n° 3, p. 223-270, 1992.

Carzaniga A., Fuggetta A., Hall R. S., Heimbigner D., Hoek A., Wolf A. L., A Characterization Framework for Software Deployment Technologies, Technical Report n° CU-CS-857-98, University of Colorado, Department of Computer Science, 1998.

Dearle A., Kirby G., McCarthy A., « A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications », *International Conference on Autonomic Computing (ICAC 2004)*, New York, USA, p. 300-301, 2004.

Frénot S., Ibrahim N., Le Mouël F., Ben Hamida A., Ponge J., Chantrel M., Beras D., « ROCS : a Remotely Provisioned OSGi Framework for Ambient Systems », *12th IEEE/IFIP Network Operations and Management Symposium (NOMS 2010)*, Osaka, Japan, April, 2010.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns*, Addison-Wesley, Boston, MA, January, 1995.

Gu T., Pung H., Zhang D., « A Middleware for Building Context-Aware Mobile Services », *Vehicular Technology Conference (IEEE VT 2004)*, Singapore, p. 2656-2660, May, 2004.

Hoareau D., Mahéo Y., « Middleware support for the deployment of ubiquitous software components », *Personal and Ubiquitous Computing*, vol. 12, p. 167-178, 2008.

Ibrahim N., Frénot S., Le Mouël F., « User-Excentric Service Composition in Pervasive Environment », *24th IEEE International Conference on Advanced Information Networking and Applications (AINA 2010)*, Perth, Australia, p. 682-689, April, 2010.

Kichkaylo T., Ivan A., Karamcheti V., Sekitei : An AI planner for Constrained Component Deployment in Wide-Area Networks, Technical Report n° TR2004-851, New York University, Department of Computer Science, 2004a.

Kichkaylo T., Karamcheti V., « Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications », *13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, Washington, DC, USA, p. 150-159, 2004b.

Li X., Fan Y., Wang J., Wang L., Jiang F., « A Pattern-Based Approach to Development of Service Mediators for Protocol Mediation », *7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, Washington, DC, USA, p. 137-146, 2008.

OSGi Alliance, OSGi-The Dynamic Module System for Java, 2010. Release 4, http ://www.osgi.org/.

Park N., Lee K., Kim H., « A Middleware for Supporting Context-Aware Services in Mobile and Ubiquitous Environment », *IEEE International Conference on Mobile Business (ICMB 2005)*, Washington, DC, USA, p. 694-697, 2005.

Poladian V., Sousa J., Garlan D., Shaw M., « Dynamic Configuration of Resource-Aware Services », *26th International Conference on Software Engineering (ICSE 2004)*, Washington, DC, USA, p. 604-613, 2004.

Preuveneers D., Berbers Y., « Towards Context-Aware and Resource-Driven Self-Adaption for Mobile Handheld Applications », *ACM Symposium on Applied Computing (SAC 2007)*, New York, NY, USA, p. 1165-1170, March, 2007.

Preuveneers D., Berbers Y., « Encoding Semantic Awareness in Resource-Constrained Devices », *IEEE Intelligent Systems*, vol. 23, n° 2, p. 26-33, 2008.

Rossi G., Gordillo S., Lyardet F., « Design Patterns for Context-Aware Adaptation », *Applications and the Internet Workshops, SAINT Workshops*, Los Alamitos, CA, USA, p. 170-173, 2005.

Taconet C., Putycz E., Bernard G., « Context-Aware Deployment for Mobile Users », *27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, Washington, DC, USA, p. 74-81, 2003.

Article reçu le 7 avril 2009
Accepté après révisions le 20 octobre 2010

Amira Ben Hamida *est docteur en informatique de l'INSA de Lyon et de l'ENSI de Manouba depuis 2010. Ses travaux de recherche se sont d'abord orientés vers les intergiciels orientés composants (J2EE, OSGi) et leur adaptation dynamique aux contraintes d'un environnement ambiant (ressources limitées, fournisseurs multiples de services, etc.). Actuellement, elle est ingénieur de recherche chez Petals Link où elle occupe la fonction de « responsable haute distribution ». Elle s'intéresse particulièrement aux aspects chorégraphie et gouvernance de services dans les environnements hautement distribués.*

Frédéric Le Mouël *est maître de conférences à l'INSA de Lyon, département Télécommunications, laboratoire CITI et chercheur à l'INRIA Rhône-Alpes, équipe AMAZONES depuis 2004. Il a obtenu une thèse en informatique et télécommunications de l'Université de Rennes 1 / IRISA en 2003 sur l'exécution distribuée d'applications en environnements mobiles. Ses thèmes de recherche sont les intergiciels et architectures orientés services, plus spécifiquement dans les domaines de l'adaptation dynamique, de la composition et de la coordination de services. Il étudie ces sujets dans le domaine de l'intelligence ambiante, des environnements mobiles et contraints.*

Stéphane Frénot *est maître de conférences au Centre d'innovations en Télécommunications et Intégration de Services, et enseigne au département Télécommunications Services et Usages de l'INSA de Lyon. Il est responsable de l'équipe INRIA AMAZONES, qui travaille sur les architectures logicielles ambiantes. Ses domaines de compétences sont les architectures orientées services, les machines virtuelles et les systèmes d'exploitations. Il travaille actuellement sur des environnements permettant le contrôle de ressources tout en conservant un modèle d'ingénierie best-effort.*

Mohamed Ben Ahmed *est titulaire d'une thèse de 3^{e} cycle de la faculté des Sciences de Paris en 1966 et d'un doctorat ès-sciences informatiques de Paris IX en 1978. Il est professeur d'informatique depuis 1980 et professeur émérite depuis mars 2005. Il a créé l'École Nationale des Sciences de l'Informatique (ENSI) au sein de l'université de Tunis en 1984 et a dirigé cette institution de 1984 à 1990. Il a créé le laboratoire de recherche RIADI en 1992 et l'a dirigé jusqu'en décembre 2004. Il a également créé la première école doctorale d'informatique en Tunisie en 1994. Ses recherches portent sur la linguistique computationnelle, le génie cognitif, l'ingénierie ontologique, l'analyse de documents, le X-Mining, le e-learning…*