
Une approche pour un chargement contextuel de services dans les environnements pervasifs

Amira Ben Hamida * ** – Frédéric Le Mouél * – Stéphane Frénot * – Mohamed Ben Ahmed **

* Amazones INRIA / CITI, INSA-Lyon, F-69621, France

** RIADI, ENSI-Tunis, T-2010, Tunisie

{amira.ben-hamida, frederic.le-mouel, stephane.frenot}@insa-lyon.fr,
{mohamed.benahmed}@riadi.rnu.tn

RÉSUMÉ. L'installation des applications sur des terminaux mobiles pose le problème de la non adéquation des ressources proposées avec celles requises par les applications. Dans cet article, nous présentons AxSeL (A conteXtual Service Loader) une architecture de chargement contextuel de services. AxSeL considère des applications composées de services chargés à partir de dépôts distants. Les services et leurs dépendances sont représentés sous la forme d'un graphe bidimensionnel, que nous colorions selon les contraintes des dispositifs cibles. La coloration a pour but de choisir les services à charger selon le contexte matériel du dispositif.

ABSTRACT. While installing applications on mobile devices, issues like the lack of memory are encountered. In this paper, we introduce AxSeL (A conteXtual Service Loader) a contextual service loading architecture. Considered applications are service-oriented, and services are distributed in remote repositories. In AxSeL, services and their dependencies are represented as a bidimensional graph which is coloured considering devices and services constraints. Colouring goal is to choose to load (or not) a service according to the execution context (device constraints, available services, etc.) .

MOTS-CLÉS : Intergiciel, dispositifs contraints, services, composants, graphes, coloration de graphes

KEYWORDS: Middleware, constraint devices, services, components, graphs, graph colouring

1. Introduction

Les environnements pervasifs proposent de fournir aux utilisateurs la possibilité d'accéder, à travers leurs dispositifs mobiles, aux applications de proximité. Installer une application sur un dispositif mobile revient à la découvrir, la rapatrier localement, la déployer et enfin à l'exécuter. Or, les capacités d'accueil des dispositifs mobiles sont très hétérogènes. Une application pouvant être chargée sur un noeud mobile donné ne l'est pas forcément sur un autre. Il est donc nécessaire d'adapter ce déploiement, si possible, en tenant compte des contraintes du dispositif. L'adaptation permet de fournir à un plus large éventail de dispositifs la possibilité de charger les applications proposées.

La programmation orientée composants permet de concevoir une application comme étant un ensemble de composants dépendants. Le paradigme service (cf 3.1.1) fournit les possibilités de description, de publication, de composition et de réutilisation des services. Nous exploitons ces propriétés et proposons une solution pour adapter les applications orientées services aux contraintes contextuelles. Ces contraintes peuvent être d'ordre matériel (limite des ressources des terminaux), ou en relation avec la qualité du service ou les préférences d'un utilisateur.

Nous considérons des applications orientées services, et attribuons aux services des niveaux de priorité accordés selon l'importance de ceux-ci pour l'application. Nous assignons également aux services considérés et aux terminaux mobiles cibles des contextes de description : capacité d'accueil, taille mémoire, priorité. L'adaptation est réalisée en confrontant un contexte requis par les services et les utilisateurs avec un contexte fourni par les périphériques. Nous ne choisissons ensuite que les services obéissant aux contraintes rencontrées. Nous offrons ainsi des configurations contextuelles des applications à charger en fonction des contraintes.

Dans cet article nous présentons AxSeL, une architecture de chargement contextuel de services. Elle est composée de plusieurs services et peut être distribuée sur un ensemble de machines. AxSeL propose :

- Une vue unifiée des services et composants fournis : les applications considérées par AxSeL sont localisées sur des dépôts distants. Des descripteurs de dépôts de services permettent de donner une vue locale du serveur ainsi que des services qu'il fournit. Afin de charger les applications, les utilisateurs accèdent à ces descripteurs de dépôts, rapatrient et déploient les services composants une application donnée. AxSeL propose une prise en compte du contexte au niveau de ces descripteurs de dépôts. En effet, nous agrégeons plusieurs descripteurs de dépôts dans un seul fournissant ainsi une vue unifiée des différents dépôts de services.

- Une décision contextuelle du chargement : en se basant sur la vue unifiée des dépôts distants, les services, composants et leurs dépendances sont représentés sous la forme d'un graphe bidimensionnel (cf 3.2.1). Nous exploitons

l'apport de la théorie des graphes et réalisons la coloration de ce graphe à travers un algorithme tenant compte des caractéristiques du service ainsi que celles de la plate-forme hôte. Deux couleurs sont utilisées pour exprimer la décision à prendre pour chaque service.

– La possibilité de chargement-déchargement de services. Les services jugés non nécessaires ou encombrants peuvent être déchargés du noeud, afin de libérer les ressources de celui-ci. Le déchargement confère à AxSeL un caractère minimal et évolutif.

Dans la suite de cet article, nous présentons un état de l'art sur les mécanismes de chargement et les plates-formes de déploiement contextuel, en section 2. L'architecture AxSeL est présentée dans la section 3, avec nos définitions du modèle de service, la mise en place d'un graphe bidimensionnel et les algorithmes de coloration contextuelle de ce graphe. Le prototype d'AxSeL est dans la section 4 avec les détails de conception et les résultats des tests de performance mémoire et temps d'exécution.

2. Etat de l'art

Nous divisons notre état de l'art en deux parties : la première traite des mécanismes de chargement de services, composants et autres modules logiciels existants, et la seconde aborde les architectures de déploiement prenant en compte le contexte.

2.1. Mécanismes de chargement de services, composants et autres modules logiciels

Nous définissons la notion de chargement comme étant le mécanisme à travers lequel des objets logiciels sont inclus en mémoire lors de leur instantiation par les plates-formes permettant leur lancement. Pour effectuer cela, certaines architectures comme java utilisent des chargeurs de classes, une classe étant un fichier contenant du bytecode provenant soit d'un emplacement local soit d'un emplacement distant (Parker et al., 2004). D'autres approches, telles qu'OSGi (Hall et al., 2004) ou la plate-forme .NET (Chappell, 2006) considèrent les composants et non les fichiers et reposent sur des chargeurs de modules pour déployer leurs composants. Le module d'OSGi (bundle) et celui de .NET (assembly) contiennent des classes rassemblées dans une unique unité de déploiement. Celle-ci est ensuite déployée explicitement par la plate-forme en question à partir d'un emplacement local ou distant, et ne démarrera que si ses dépendances de classes et de modules sont satisfaites. Des approches telles que OBR (OBR, 2008) fournissent un descripteur pour un dépôt de bundles OSGi et tracent les dépendances de ces derniers afin d'en garantir le rapatriement lors de l'installation d'un bundle donné. Dans les approches orientées

services nécessitant le chargement de ceux-ci, les plates-formes reposent également sur des chargeurs de modules, cependant, les dépendances d'un service en termes de services et composants peuvent être satisfaites par plusieurs autres services ou composants. Le chargement est effectué dans l'ensemble des architectures décrites en ne prenant en compte que le critère des dépendances entre les composants ou les services comme dans (Microsystems, n.d.). Cependant, nous évoquons également le besoin de prise en compte d'autres critères tels qu'une meilleure adaptation aux contraintes des plates-formes matérielles et une adéquation aux préférences utilisateurs lors du chargement. Le contexte (Coutaz et al., 2005) permet d'atteindre ces objectifs. Certaines approches de déploiement incluent dans leur modèle l'élément contextuel, nous en présentons quelques unes dans la section suivante.

2.2. Approches de déploiement contextuel

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

(Hoareau et al., 2008) considèrent le problème de fragmentation du réseau causée par la volatilité des hôtes dans les environnements pervasifs. 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 consistance du déploiement.

Les contraintes abordées peuvent aussi être d'ordre matériel tel que l'énergie d'un terminal ou son espace mémoire. Dans (Kichkaylo et al., 2004b), les auteurs 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. Smart Deployment Infrastructure (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 ubiquitaire. 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.

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.

Les architectures décrites réalisent du déploiement de composants logiciels sur des plates-formes 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 distribué en considérant aussi bien les contraintes des composants que celles des machines. Les questions auxquelles elles répondent sont : quels composants déployer et sur quelles machines ?

Système	Déploiement	Unité	Décision
(Hoareau <i>et al.</i> , 2008)	installation et activation	composant	ADL
(Kichkaylo <i>et al.</i> , 2004b)	installation	composant	intelligence artificielle
(Dearle <i>et al.</i> , 2004)	configuration et mise à jour	composant	descripteur de déploiement
(Poladian <i>et al.</i> , 2004)	configuration et adaptation	service	modèle mathématique
(Taconet <i>et al.</i> , 2003)	installation	composant	descripteur de déploiement
(Park <i>et al.</i> , 2005)	adaptation	service	règles prédéfinies
(Gu <i>et al.</i> , 2004)	adaptation	service	raisonnement

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

AxSeL opère au niveau local (terminal) et propose selon les contraintes rencontrées (matérielles et autres) une configuration 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 question que nous tentons de résoudre est : quels services charger sur le terminal en respectant les contraintes des terminaux, celles des services et celles des utilisateurs ?

3. AxSeL : une architecture de chargement contextuel de services

Dans cette section nous présentons d'abord, les modèles et définitions sur lesquelles repose notre système, soient le modèle de service et le modèle de contexte. Ensuite, nous détaillons l'architecture AxSeL.

3.1. Modèles et définitions

3.1.1. Modèle de composant/service

(K.Kui *et al.*, 2007) présente un état de l'art des différentes définitions de composant fournies par la littérature. Nous en retenons trois afin de positionner notre définition du composant, du service et des propriétés qui lui sont intrinsèques.

1) déf. 1 : Szyperski : "A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

2) déf. 2 : Meyer : "A component is a software element (modular element), satisfying the following conditions : (i) It can be used by other software elements, its "clients". (ii) It possesses an official usage description, which is sufficient for a client author to use it. (iii) It is not tied to any fixed set of clients."

3) déf. 3 : Heineman and Councill : "A component is a software element that conforms to a component model and be independently deployed and composed without modification according to a composition standard".

Nous reposons sur les définitions fournies par la littérature pour proposer un modèle de service basé sur une séparation entre les niveaux services et composants. Ainsi, un composant est une unité logicielle renfermant du code et éventuellement des fonctionnalités. Un composant peut être déployé et exécuté (déf. 2 et déf. 1). Il peut posséder des interfaces. Une interface correspond à un service. Un composant peut publier et importer zéro ou plusieurs services. Les services servent à exporter ou importer des fonctionnalités d'un composant à un autre. Un composant est sujet à la composition avec d'autres composants à travers les services publiés et importés qu'il peut avoir (déf. 1). Un composant et ses services ne sont pas dédiés à un utilisateur particulier (déf. 3). Pour des besoins de conception nous introduisons la notion de service nul que nous définissons comme étant un service publié par un composant et affichant des méthodes répondant nul à l'appel. Ce dernier permet dans le cas de l'absence du service adéquat de le remplacer par un service vide. La figure 1 illustre le modèle de service/composant considéré :

Un service est caractérisé par des :

- Interfaces : des interfaces fonctionnelles (méthodes fournies par le service), des interfaces requises (services, composants ou paquets requis par le service),
- Implémentations : des implémentations des méthodes définies dans les interfaces fonctionnelles (composant publiant le service),
- Propriétés : des propriétés non fonctionnelles du service (taille mémoire requise, priorité, etc.)

Les composants et leurs dépendances appartiennent à l'étape du déploiement, alors que les services et leurs dépendances apparaissent à l'exécution. Les dépendances entre les interfaces proposées et celles requises peuvent être satisfaites, s'il existe des services ou composants disponibles y répondant.

La découverte de services peut être faite statiquement à travers des dépôts de services, composants, ou dynamiquement à travers des protocoles de découverte tels qu'UPnP (Corporation, 2000) ou Jini (Kumaran, 2002).

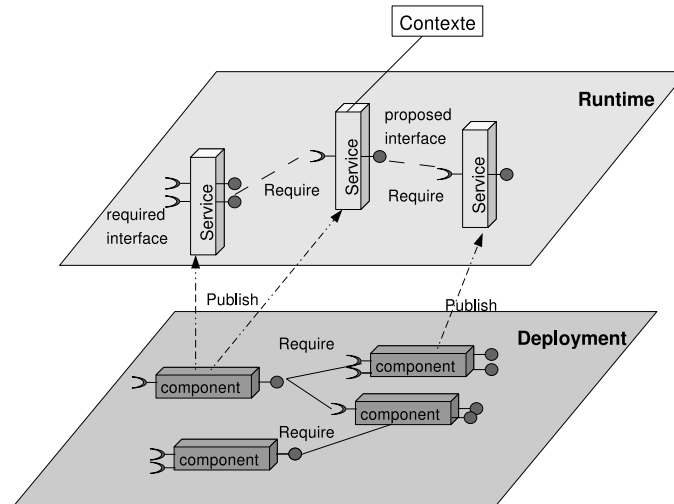


Figure 1. Modèle de service d'AxSeL

D'autres plates-formes fournissent des modèles orientés-services tels que EJB (Hamilton, 1997), CORBA (Zahavi, 1999), OSGi (Alliance, 2008) ou les services web (Iverson, 2004).

La séparation entre services et composants proposée permet d'associer à chacun d'eux des propriétés qui leurs sont relatives. Le choix réalisé au chargement peut être opéré aussi bien au niveau des services, qu'au niveau des composants en considérant un ensemble de propriétés. Afin de prendre en compte les propriétés des services et des composants lors du chargement un modèle de contexte (cf 3.1.2) leur est associé. Dans la section suivante nous définissons notre propre modèle du contexte.

3.1.2. Modèle de contexte

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." (Dey et al., 2001)

Nous définissons le contexte comme étant l'ensemble des données pouvant être recueillies à partir des services et composants (nom, version, besoin en ressources), des dispositifs (capacité d'accueil), et des utilisateurs (préférences). La figure 2 illustre l'association d'un contexte (requis et fourni) à un service et un composant donnés.

AxSeL utilise ces données du contexte pour effectuer le chargement des services, en confrontant un contexte requis (service, utilisateur) à un contexte

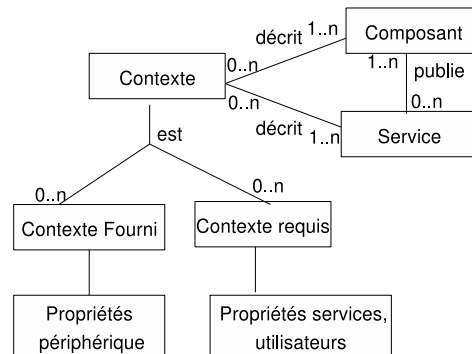


Figure 2. Liaison entre le modèle de service et le contexte

fourni (noeud mobile). La comparaison permet de repérer les services à charger. Dans cet article, afin de simplifier la prise de décision, nous ne retenons que deux données du contexte :

- La priorité : à travers laquelle nous exprimons le degré d'importance d'un service par rapport à d'autres dans le cadre d'une application. Elle est affectée par le fournisseur du service et renseigne sur la nécessité de charger ou non un service. Le service est chargé si la priorité est haute, sinon il est moins utile de l'avoir sur le dispositif et nous gagnons ainsi l'espace mémoire qu'il aurait pu occuper.

- La taille mémoire : pour optimiser l'occupation mémoire des terminaux mobiles, nous introduisons une seconde propriété qui est la taille mémoire. Celle-ci informe sur l'occupation d'espace mémoire d'un service en exécution. Nous supposons que cette valeur est donnée par le fournisseur du service. Si le service requiert plus que l'espace mémoire disponible sur le dispositif, nous ne pouvons pas le charger et optons, si possible, dans ce cas pour son remplacement par un autre service consommant moins de mémoire.

AxSeL peut considérer d'autres contraintes afin de mieux adhérer aux besoins des utilisateurs, dispositifs mobiles ou services. Ces contraintes peuvent relever des services (confiance, durée de vie, etc), du matériel (autres besoins en ressources matérielles), ou des préférences utilisateurs.

3.2. Architecture générale

Dans un environnement pervasif, les services sont hébergés sur des terminaux mobiles, ou des dépôts distants, et sont décrits par des descripteurs. Les descripteurs de services incluent des données sur les services (dépendances, emplacements, taille mémoire, etc.). Les terminaux mobiles accèdent à ces dépôts à travers leur descripteur de services, et y puisent les services requis. 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 3 présente une vue générale de l'architecture.

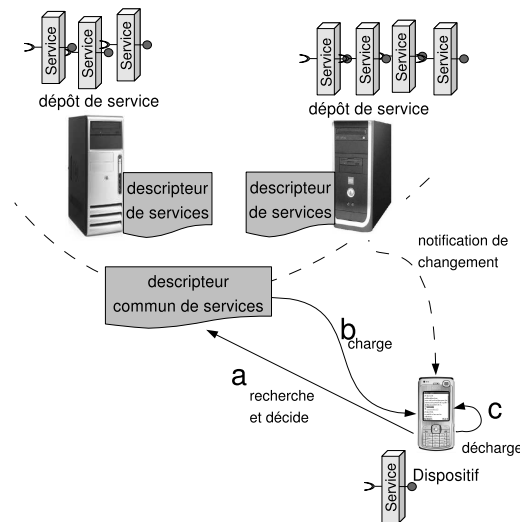


Figure 3. Comportement de la plate-forme AxSeL : (a) le dispositif recherche et décide du service à charger, (b) il le charge ainsi que ses dépendances localement, (c) en cas de changement le service peut être déchargé.

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 (descripteur commun de services, figure 3).

2) La décision du chargement : cette étape consiste en un 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 dépendance, une décision de chargement ou non, selon les contraintes, est prise (étape a, figure 3).

3) Le chargement : est l'étape qui applique la décision précédente en effectuant techniquement le télé-chargement des composants publiant le service, son installation et son démarrage (étape b, figure 3).

4) L'adaptation contextuelle : est déclenchée soit par le changement d'état d'un ou plusieurs services, soit par la suppression des services jugés non né-

cessaires. Cette étape déclenche une nouvelle prise de décision avec comme conséquences d'éventuels chargements/déchargements (étape c, figure 3).

AxSeL est une architecture orientée services où chacune des étapes 1, 2 et 4 correspond à un service. Ainsi, les services d'extraction et d'adaptation peuvent s'exécuter sur le hôte A alors que le service de décision est sur le hôte B (figure 4).

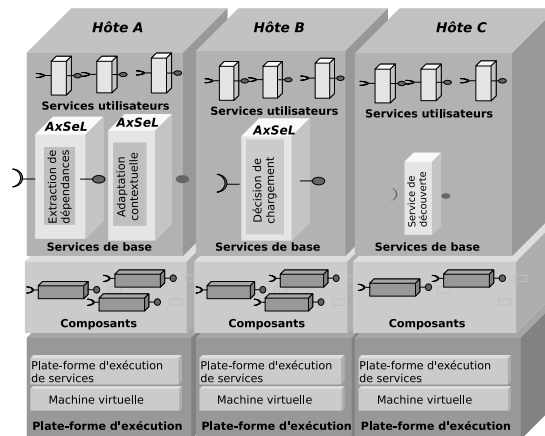


Figure 4. Vue par couche de la plate-forme d'exécution

Dans les sections suivantes, nous présentons le graphe des dépendances de services, la coloration considérée pour la prise de décision du chargement, et enfin l'adaptation contextuelle. Le mécanisme de chargement ne sera pas présenté car nous délégons cette tâche à la plate-forme d'exécution sous-jacente (voir section 4).

3.2.1. Graphe bidimensionnel de services/composants

Structure du graphe : à partir de la description de services, nous extrayons dans un graphe l'ensemble des dépendances possibles d'un service, ainsi que les données relatives aux services et aux composants l'implantant (noms et emplacement des services/composants, taille mémoire requise). Nous incluons ces données dans les graphes extraits afin de les prendre en compte lors de la décision de chargement. Le choix du chargement est opéré aussi bien sur les services que sur les composants, d'où le besoin de considérer un graphe bidimensionnel (figure 5) incluant un niveau de services et un niveau de composants. Il s'agit d'un graphe orienté où les services et composants sont représentés par des noeuds et les dépendances par des arcs les reliant.

– Noeuds : les noeuds du graphe représentent les services et les composants. Les services sont représentés par des noeuds ronds alors que les composants

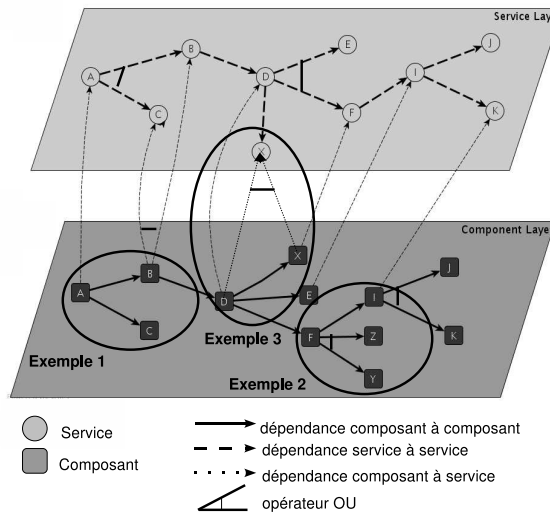


Figure 5. Graphe de dépendance d'un service

par des noeuds carrés. Chaque noeud possède ses propres caractéristiques et son propre poids. Les noeuds portent des propriétés.

– Arcs : représentent les dépendances. Les noeuds appartenant au même niveau sont liés par des arcs horizontaux, alors que ceux appartenant à des niveaux différents sont reliés par des liens verticaux. Nous faisons correspondre le modèle de services proposé à un graphe bidimensionnel où nous traçons les dépendances que nous définissons comme étant la relation d'import/export entre les éléments du modèle (noeuds). Les services sont implantés par des composants, les composants peuvent importer d'autres composants. Enfin, les services importent d'autres services. Un arc va d'un service à un autre, ou d'un composant à un autre ou d'un composant à un service. Les arcs sont orientés et peuvent avoir des annotations spécifiques.

Nous incluons dans notre modèle deux opérateurs logiques : ET et OU. Une dépendance portant l'opérateur ET renseigne sur la nécessité de charger les noeuds de cette dépendance. L'exemple 1 de la figure 5 illustre ce cas. Le composant A doit charger les composants B ET C. Une dépendance portant l'opérateur OU représente la possibilité de choisir un noeud ou un autre. Cet opérateur apparaît dans le cas de l'existence de plusieurs versions d'un service par exemple. Son inclusion dans le modèle est relative à notre volonté de puiser les services de différents dépôts. Dans l'exemple 2 de la figure 5, le composant F dépend des composants I ET (Z OU Y). Enfin, l'exemple 3 de la même figure

montre le cas où un service est publié par un ou plusieurs composants. Le service X peut être fourni soit par le composant X, soit par le composant D.

Algorithme d'extraction du graphe : le processus d'extraction se fait sur deux opérations :

1) Retrouver les dépendances : cette opération se base sur l'analyse du descripteur des services. Nous désignons un service ou un composant par le terme générique (Resource). Nous en dégageons l'ensemble des dépendances (Requirement) grâce à la fonction (getRequirements()). Nous parcourons ensuite ceux-ci et lançons récursivement l'extraction de leurs dépendances. L'algorithme 1 se déroule jusqu'à ce que nous ayons retrouvé toutes les dépendances d'un service.

2) Construire le graphe : cette opération est réalisée au fur et à mesure que l'opération précédente se déroule. Les noeuds sont créés et inclus dans le graphe selon leur type. Un service correspond à un noeud appartenant au niveau des services (ServiceLayer) et un composant correspondra à un noeud du niveau des composants (ComponentLayer). Les liens sont inclus entre les noeuds d'un même niveau et ceux de différents niveaux. A chaque étape de l'algorithme, l'inclusion entre le graphe courant et le graphe de dépendances de ses fils est enrichi d'une dépendance entre le point d'entrée du graphe courant et celui de ses fils. L'algorithme d'extraction a une complexité de $O(n \log(n))$.

Algorithm 1 Graph extractDependency(Resource resource)

```

1: G ← new Graph(resource)
2: Requirement[] children ← resource.getRequirements()
3: for i, Gi ← extractDependency(children[i])
4: if Gi.firstNode provides Service then
5:   include Gi.firstNode in ServiceLayer
6: else if Gi.firstNode provides Component then
7:   include Gi.firstNode in ComponentLayer
8: end if
9: G.include(Gi)
10: G.addDependencyEdge(G.firstNode, Gi.firstNode)
11: return (G)

```

3.2.2. Coloration du graphe de services

Une fois le graphe de dépendances d'un service extrait, nous passons à l'étape suivante qui est la prise de décision des services à charger. Elle se base sur la confrontation de données contextuelles des services et des contraintes de la plate-forme. Ces dernières ne sont pas forcément compatibles, nous ne chargeons donc que les services obéissant à nos contraintes. La décision du chargement se base sur un algorithme de marquage par coloration des noeuds du graphe.

Principe de coloration : nous parcourons le graphe et affectons une de ces deux couleurs aux noeuds du graphe bidimensionnel extrait, selon la décision à prendre (chargement ou non) :

- Le rouge est affecté aux noeuds obéissant aux critères de chargement et qui seront chargés localement sur la plate-forme.

- Le blanc est affecté aux noeuds n'obéissant pas aux contraintes prédéfinies. Dans ce cas la décision est la création et la re-direction vers un service nul. Notons que la couleur blanche ne peut pas briser une chaîne de noeuds rouges (figure 6).

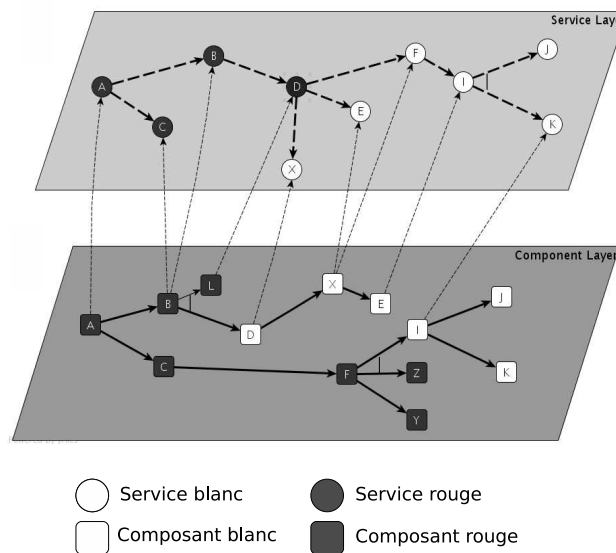


Figure 6. Coloration du graphe de dépendances d'un service

Lors du parcours du graphe, lorsqu'un noeud rouge est rencontré il est chargé. Quand il s'agit d'un noeud blanc, AxSeL crée et redirige vers un service nul. Nous sommes conscients que rediriger vers un service nul ne fait pas pleinement fonctionner l'application chargée. Mais, cependant, de cette manière nous garantissons la bonne exécution d'une application (pas de crash dû à un service manquant) et de ne charger que le nécessaire, en attendant la présence d'un autre service équivalent et plus adéquat à nos contraintes.

Algorithme de coloration : nous nous inspirons de la théorie des graphes pour déduire à partir des algorithmes de parcours de graphes existants celui répondant à nos besoins. Des approches telles que le chemin le plus court fournies par exemple par Kruskal, PRIM, Dijkstra, Bellman-Ford-Moore, Floyd Warshall (Lacomme et al., 2003), nous renseignent sur la manière avec laquelle

la décision du chargement peut être prise en prenant compte des tailles des composants à charger et de la capacité mémoire du terminal à ne pas dépasser. Nous nous sommes également intéressés aux méthodes heuristiques telles que les algorithmes génétiques et A*, afin d'optimiser le parcours des graphes.

L'algorithme de coloration d'AxSeL utilise deux métriques représentant la priorité d'un service prio et sa taille sizeNode :

- prio() est une fonction qui retourne la priorité d'un service. prioMin est le niveau en dessous duquel les services ne pourront pas être choisis. L'utilisateur fixe prioMin de telle sorte que seuls les services qu'il souhaite soient chargés.
- sizeNode représente la somme des tailles mémoires des services (noeuds) de la solution en cours de coloration. sizeMax est une valeur constante représentant la taille mémoire maximale allouée sur un dispositif. Au delà de cette valeur nous ne pouvons plus charger de services.

La coloration de graphe s'effectue en deux étapes :

1) Coloration initiale en blanc : au départ, l'ensemble du graphe est coloré en blanc. Par défaut, aucun service n'est donc chargé mais redirigé vers un service nul. Cette étape est absolument non coûteuse car nous ne parcourons pas réellement le graphe pour le colorier, mais nous avons prémarqué les noeuds en blanc lors de leur création.

2) Coloration en rouge : nous parcourons réellement le graphe pour effectuer la coloration en rouge en décidant à chaque noeud s'il respecte les contraintes du contexte requis par rapport à celles du contexte fourni par le périphérique. Cette étape s'initie en appelant l'algorithme 2 avec le point d'entrée du graphe de dépendances et une valeur de solution courante nulle : colour(new List[G.firstNode], 0)

Algorithm 2 List colour(List nodeList, Integer sizeNode)

```

1: nodeList.sort(prio)
2: firstNode = nodeList.firstElement()
3: if firstNode.getPrio() < prioMin and sizeNode + firstNode.getSize() < sizeMax then
4:   firstNode.changeColour(red)
5:   return colour(nodeList-firstNode+children(firstNode), sizeNode+firstNode.getSize())
6: else
7:   return colour(nodeList-firstNode,sizeNode)
8: end if

```

L'algorithme trie les noeuds selon leur priorité, ensuite prend le premier noeud de la liste triée. Si ce noeud obéit aux contraintes de priorité et de taille mémoire il est coloré en rouge, et nous procédons au traitement de ses fils. Le

noeud coloré est ôté de la liste de départ. L'algorithme retourne une liste de noeuds coloriés. La complexité de cet algorithme est évaluée à $O(n \log(n))$.

3.2.3. Adaptation dynamique d'AxSeL

Les environnements pervasifs sont sujets à des variations continues dues à la mobilité des utilisateurs, des services, etc. De telles variations sont prises en considération dans AxSeL et ce au travers d'une écoute des éléments contextuels (dispositif, utilisateur, service, etc). Cette section décrit d'abord, le principe d'adaptation d'AxSeL : les événements auxquels est sensible AxSeL et leurs répercussions. Un algorithme implantant ce principe est ensuite présenté.

Principe d'adaptation : AxSeL prend en compte la dynamique du contexte. Cette dynamique est mise en place à l'aide d'une approche événementielle. AxSeL répond à deux types d'événements :

- Les événements liés au changement du dépôt de services : ces événements sont déclenchés lorsque des nouveaux services apparaissent, ou d'autres déjà chargés sont mis à jour (nouvelle version, nouvelle implantation, indisponibilité).
- Les événements liés au périphérique : ceux-ci sont déclenchés dans le cas où il existe un changement au niveau de la ressource mémoire du périphérique. Si celle-ci est insuffisante ou qu'elle vient d'être libérée et peut donc probablement héberger d'autres services.

En réponse à ces événements AxSeL réalise deux types d'actions sur le graphe :

- Action de modification de la structure du graphe : l'apparition d'un service ou sa disparition entraîne respectivement l'ajout ou le retrait du noeud le représentant dans le graphe. Il en est de même pour les dépendances entre les noeuds.
- Action de changement de la coloration d'un noeud : l'état de la mémoire influence celle des noeuds du graphe. En effet, lorsque celle-ci est libérée il devient possible de charger de nouveaux services (recoloriage en rouge) tant que l'espace mémoire le permet. Et lorsque la mémoire est encombrée certains noeuds rouges sont décoloriés (blanc). Les préférences utilisateur influencent également l'état des noeuds, si ce dernier désire charger, décharger ou modifier les priorités de services particuliers. Enfin, le changement de l'état d'un service (disparition/apparition/modification) engendre le changement de sa couleur.

Les actions listées entraînent une nouvelle prise de décision. L'algorithme de coloration considéré est incrémental et prend en compte les services préalablement chargés, pour ne pas re-parcourir le graphe et le re-colorier totalement.

Algorithme d'adaptation : l'algorithme 3 d'adaptation du chargement est appelé sur notification d'un événement de l'environnement auquel AxSeL est abonné. Les événements listés peuvent venir de deux sources :

- Ajout d'un noeud : nous vérifions s'il existe un parent au noeud ajouté qui soit déjà rouge. Dans ce cas, nous lançons la re-coloration des noeuds avec le nouveau noeud, et la valeur courante `sizeNode`.

- Changement de l'état d'un noeud : si le noeud est déjà rouge alors nous le décolorons (blanc), déduisons sa taille de `sizeNode` et enfin, lançons la fonction `uncolor` (algorithme 4) sur ses fils. La re-coloration des noeuds ainsi que le noeud modifié est relancée avec la variable `sizeNode`.

Algorithm 3 adaptation(Event e)

```

1: if e.getType() == NewNode then
2:   if ∃ nodei / father(node) and nodei.isColored(red) then
3:     colour(l+node, sizeNode)
4:   end if
5: else if e.getType() == NodeChange then
6:   if node.isColored(red) then
7:     node.changeColour(white)
8:     sizeNode = sizeNode - node.getSize()
9:     uncolor(new List (children(node)))
10:    color(l + node, sizeNode)
11:  else if node.isColored(white) and ∃ nodei / father(node) and no-
12:    dei.isColored(red) then
13:    colour(l+node, sizeNode)
14:  end if
15: end if

```

La fonction `uncolor(List)` permet de vérifier qu'un noeud rouge n'a pas de prédécesseurs rouges. Dans ce cas, la coloration du noeud en question en blanc est possible. La taille du noeud est déduite de `sizeNode`, et la fonction propage ce traitement pour les fils du noeud traité. Un noeud traité n'est pas repris en compte. La complexité de cet algorithme est de $O(n^2)$, car le décoloriage implique également un parcours pour retrouver les noeuds parents.

4. Réalisation d'AxSeL

4.1. Conception d'AxSeL

AxSeL est développée avec le langage Java. Nous avons utilisé la machine virtuelle Java comme plate-forme d'exécution et la plate-forme Felix (Felix, 2008), implantation de la spécification OSGi (Alliance, 2008), pour

Algorithm 4 uncolor(List l)

```

1:  $\forall$  node  $\in$  l
2: if node.isColored(red) and @ nodei / father(node) and nodei.isColored(red)
   then
3:   node.changeColor(white)
4:   sizeNode = sizeNode - node.getSize()
5:   uncolor(l - node + children(node))
6: end if

```

le déploiement et l'exécution des services (cf. figure 4). Nous faisons correspondre les éléments de notre modèle avec ces plates-formes : un composant est implanté par un bundle OSGi (unité de déploiement composée de classes et d'un manifest), un service est implanté par un service OSGi représenté par une interface Java. Par soucis de clarté et de concision, nous présentons les interfaces d'AxSeL et détaillons deux packages uniquement. L'ensemble des packages d'extraction de graphes (repositoryextractor), de coloration de graphes (colouringdecision), de structure du graphe (resourcegraph), de chargement de ressources (loading) et enfin d'adaptation contextuelle (contextmanagement) constituent AxSeL (figure 7). Le package (org.apache.felix) représente la plateforme OSGi sur laquelle nous reposons.

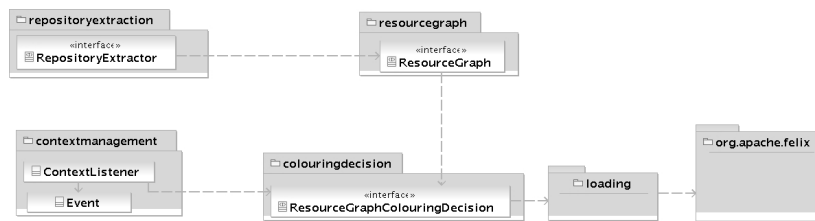


Figure 7. Architecture générale d'AxSeL, vue par packages

Package resourcegraph : la structure de ResourceGraph est implantée par la classe BundleRepositoryResourceGraph, et correspond au modèle présenté dans (section 3.2.1) (figure 8). Cette dernière s'appuie sur deux structures élémentaires : le noeud (ResourceGraphNode) et l'arc (ResourceGraphEdge). Aux noeuds Resource (composant ou service), nous assignons la structure de données ResourceGraphNode qui possède comme attribut descriptif : un lien sur la ressource représentée, son nom, la liste des arcs vers les fils, la couleur du noeud, sa taille mémoire, et enfin sa priorité. D'autre part, les liens entre les différents noeuds sont également extraits et implantés par la structure de données ResourceGraphEdge qui est définie par un ResourceGraphNode origine (from) et un vecteur (to) de ResourceGraphNode destinataires. Le choix des destinataires multiples nous permet de représenter l'opérateur logique OU.

ResourceGraphNode et ResourceGraphEdge sont respectivement implémentées par BundleRepositoryResourceGraphNode et BundleRepositoryResourceGraphEdge. Les classes ResourceGraphNodePriority et ResourceGraphNodeSize

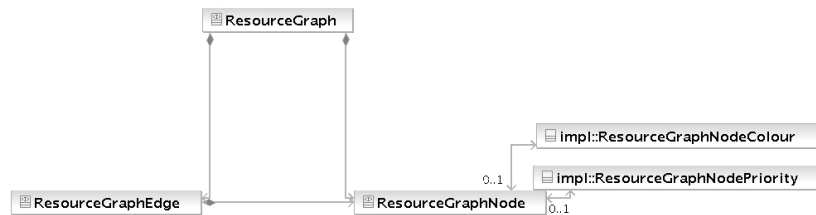


Figure 8. Diagramme de classes du ResourceGraph

désignent respectivement la priorité et la taille assignée à chaque noeud.

Package colouringdecision : la décision du chargement est réalisée grâce à un algorithme de coloration de graphe (cf. 3.2.2). Nous avons encapsulé les classes propres à la décision à l'intérieur des packages colouringdecision et colouringdecisionimpl. Ceux-ci incluent trois interfaces ResourceGraphColouringPath, ResourceGraphColouringDecision et ResourceGraphConstraint et leurs implantations respectives ResourceGraphPriorityColouringPath, CommonPathColouringDecision et ResourceGraphNodePrioritySizeConstraint (figure 9). L'algorithme de coloration CommonPathColouringDecision prend en para-

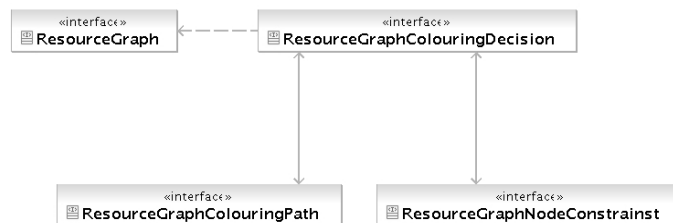


Figure 9. Diagramme de classes du package colouringdecision

mètres un chemin de parcours de coloriage (ResourceGraphColouringPath) et des contraintes locales applicables à chaque noeud (ResourceGraphConstraint). Nous appliquons ensuite le design Pattern Strategy qui permet de définir une famille d'algorithmes interchangeables pouvant varier selon le besoin. Nous instancions donc dynamiquement grâce à ce design pattern un parcours du graphe en terme de priorité (ResourceGraphPriorityColouringPath) et un test des contraintes de priorité et de taille pour chaque noeud (ResourceGraphNode). Nous proposerons plusieurs autres types de parcours et contraintes interchangeables dynamiquement dans de futurs travaux.

4.2. Évaluation des performances d'AxSeL

Nous avons évalué les performances d'AxSeL et l'avons comparé à la plate-forme OBR (OBR, 2008) d'OSGi. La plate-forme OBR puise les bundles à installer localement sur une machine dans un dépôt distant. Cependant, elle opère uniquement au niveau composant et non service, n'applique pas de décision respectant une contrainte particulière mais essaie de charger tous les bundles, et ne permet pas d'adaptation. Après examen d'un dépôt comme celui de Felix, nous avons réalisé les tests avec des graphes représentant une application moyenne de vingt noeuds services/composants, chaque noeud ayant entre un et cinq dépendances au niveau composant et trois dépendances au niveau service. Bien qu'AxSeL supporte les graphes cycliques (la détection n'entrave pas son bon fonctionnement et ses performances), nous avons testé avec des graphes sans boucles comme OBR ne les supporte pas. Nous comparons ces architectures à travers les phases d'extraction de graphe, de prise de décision et enfin d'adaptation dynamique, et évaluons leurs complexités temps (ms) et mémoire (Ko).

4.2.1. Évaluation des performances temps/mémoire

Nous comparons le temps d'exécution d'AxSeL et d'OBR, lors du déroulement des différentes phases. Lors de l'extraction (figure 10), AxSeL construit le graphe en moyenne deux fois plus rapidement qu'OBR. La prise de décision, pour un même coloriage total du graphe, s'effectue en moyenne dix fois plus rapidement. Notons également qu'AxSeL réalise une phase d'adaptation absente dans l'architecture OBR, et qui reste très performante.

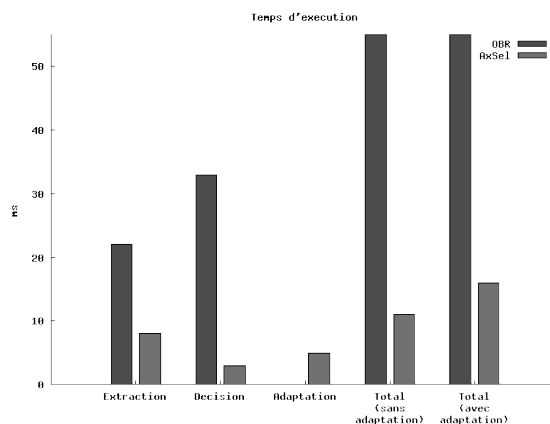


Figure 10. Évolution de la performance temps en fonction des phases

La figure 11 illustre l'allocation de la mémoire observée. AxSeL exploite plus de ressources mémoire à l'extraction du graphe (sur-coût du graphe bidi-

mensionnel), mais est cependant, deux fois moins gourmand en mémoire sur l'ensemble des phases restantes sans l'adaptation, et une fois et demi avec l'adaptation.

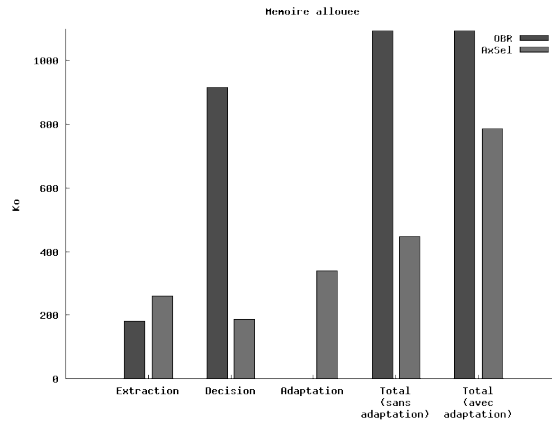


Figure 11. Évolution de la performance mémoire en fonction des phases

4.2.2. Variation des résultats en fonction du nombre d'exécution

Nous évaluons également les performances temps d'exécution et mémoire des deux plates-formes en fonction du nombre d'exécution des algorithmes. Les courbes de la figure 12 montrent une perte de performance au premier

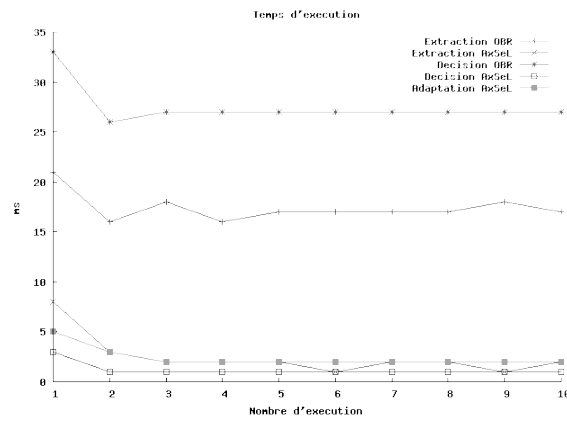


Figure 12. Évolution de la performance temps en fonction du nombre d'exécution

lancement de tous les algorithmes. Cette perte s'explique par le coût utilisé par

le chargeur de classes Java. Le sur-coût est le même pour les deux plates-formes qui ont ensuite une performance constante.

L'utilisation de la mémoire a été également évaluée en fonction du nombre d'exécution des algorithmes. Les courbes de la figure 13 montrent l'évolution de cette ressource en fonction de la phase observée. OBR utilise une mémoire

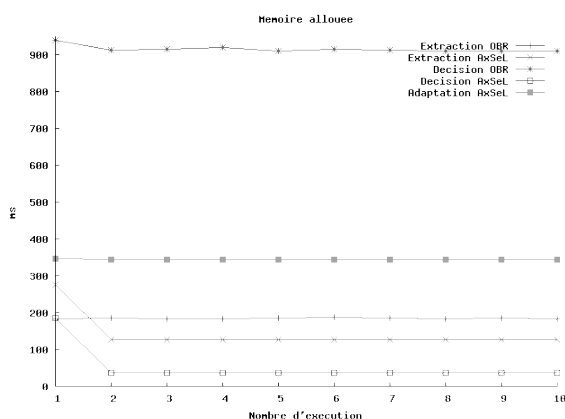


Figure 13. Évolution de la performance mémoire en fonction du nombre d'exécution

constante à la phase d'extraction et de décision. AxSeL réduit par contre considérablement l'utilisation de la mémoire dès la deuxième utilisation pour les phases d'extraction et de décision. Cette réduction s'explique par l'utilisation du design pattern Singleton pour ne pas réallouer des objets déjà créés. La phase d'adaptation est constante en allocation mémoire, et est cependant testée dans le pire des cas (recoloriage total), et utilise moins de mémoire pour un recoloriage partiel.

4.2.3. Variation des résultats en fonction du nombre de noeuds du graphe

Nous avons fait varier le nombre de noeuds des graphes testés et observé l'utilisation de la mémoire et le temps d'exécution des deux plates-formes. Les courbes de la figure 14 montrent que la phase d'extraction d'AxSeL est en moyenne deux fois plus rapide que celle d'OBR selon un comportement linéaire. La phase de décision d'OBR suit un comportement linéaire mais celle d'AxSeL est plus proche d'un temps constant considérablement plus performant. Enfin, seule AxSeL réalise l'adaptation qui observe un temps exponentiel correspondant bien à la complexité de l'algorithme.

Les courbes de la figure 15 illustrent l'évaluation de l'utilisation mémoire selon la variation du nombre de noeuds. Pendant l'extraction OBR et AxSeL suivent un comportement asymptotique où AxSeL alloue en moyenne deux fois

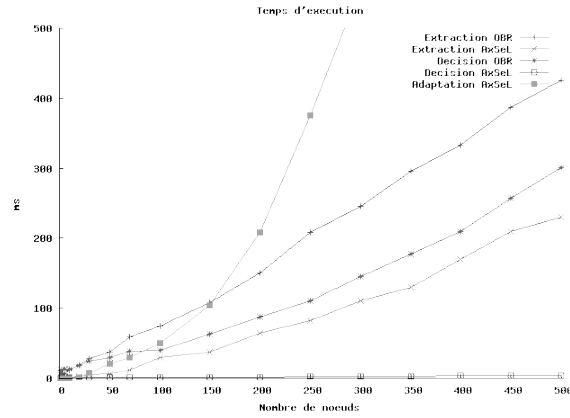


Figure 14. Évaluation de la performance temps en fonction du nombre de noeuds du graphe

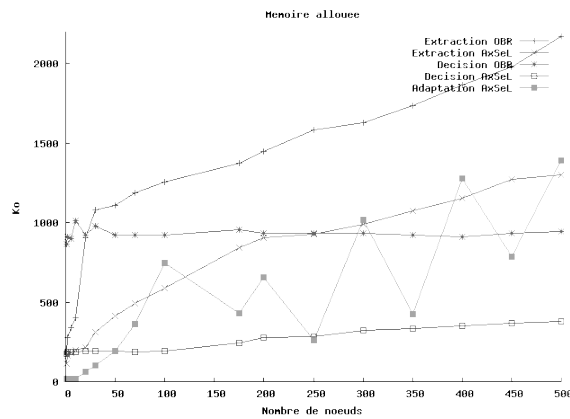


Figure 15. Évaluation de la performance mémoire en fonction du nombre de noeuds du graphe

moins de mémoire. L'allocation mémoire observée dans la figure 11 correspond donc à un comportement inversé à l'origine. A la phase de décision, OBR et AxSeL ont une allocation mémoire constante. Enfin, la courbe observée à la phase d'adaptation a un comportement oscillatoire, en raison d'un parcours total du graphe lors de l'adaptation. Ces valeurs dépendent fortement de la structure du graphe et peuvent être améliorées ultérieurement avec des changements dans la conception de notre système (recherche des noeuds parents plus performante).

5. Conclusion

Dans cet article, nous avons présenté une architecture de chargement contextualisé de services : AxSeL. Notre approche considère des applications orientées services. AxSeL repose sur un modèle de service représenté sous la forme d'un graphe bidimensionnel. Les noeuds du graphe sont enrichis par des propriétés contextuelles, comme la taille ou la priorité. Ce graphe est obtenu grâce à un algorithme réalisant l'extraction des dépendances d'un service en partant d'une description unifiée d'un ensemble de services disponibles depuis un dépôt distant. Le processus de prise de décision est réalisé grâce à un algorithme de coloration de graphes et prend en compte le contexte d'exécution. Deux couleurs exprimant la décision de chargement sont attribuées : le rouge (chargement) et le blanc (pas de chargement). Dans le cas du non chargement d'un noeud, AxSeL redirige cet appel vers un service nul. Enfin, AxSeL possède un mécanisme d'adaptation dynamique aux changements contextuels (changement de l'état d'un service, insuffisance des ressources matérielles, etc.).

AxSeL est implantée comme étant elle-même, une application orientée services, pouvant donc être déployée sur un environnement distribué. Nous évitons ainsi d'encombrer les terminaux mobiles par des processus coûteux de calcul et de parcours de graphes. Chaque algorithme (extraction, coloration, adaptation) est ainsi disponible au sein d'un service.

Nous avons réalisé le prototype d'AxSeL et l'avons soumis à un ensemble de tests, et comparé à d'autres plates-formes disponibles dans OSGi. AxSeL offrent de bonnes performances en terme de temps d'exécution et de mémoire allouée.

A court terme, nous planifions de tester de nouveaux parcours de graphes possibles selon le contexte. Nous planifions également de rendre les interfaces de services d'AxSeL conformes à la spécification OSGi, pour permettre sa diffusion et son utilisation dans la communauté. A long terme, nous souhaitons profiter de la dissémination des services dans les environnements pervasifs afin d'intégrer dans la décision du chargement, des accès distants à des services en exécution sur des plates-formes distantes. En effet, les services trop volumineux pour être chargés localement peuvent faire objet d'appels distants. Ainsi, les noeuds actuellement blancs de notre graphe pourront être redirigés vers des services distants et être colorés par exemple en bleu.

6. Bibliographie

- Alliance O., « OSGi-The Dynamic Module System for Java », <http://www.osgi.org/>, 2008.
- Carzaniga A., Fuggetta A., Hall R. S., Heimbigner D., Hoek A., Wolf A. L., A Characterization framework for Software Deployment Technologies, Technical report, University of Colorado Department of Computer Science, 1998.

- Chappell D., Introducing the .Net Framework 3.0 : A white paper, Technical report, Microsoft Corporation, 2006.
- Corporation M., Understanding UPnP : A white paper, Technical report, UPnP Forum, 2000.
- Coutaz J., Crowley J., Dobson S., Garlan D., « Context is Key », *Commun. ACM* Volume 48, 2005.
- Dearle A., Kirby G., McCarthy A., « A Framework for Constraint-based Deployment and Autonomic Management of Distributed Applications », ICAC conference, 2004.
- Dey A., Salber D., Abowd G., « A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications », HCI conference, 2001.
- Felix, « The Apache Felix Project », <http://cwiki.apache.org/FELIX/index.html>, 2008.
- Gu T., Pung H., Zhang D., « A Middleware for Building Context-Aware Mobile Services », IEEE VT conference, 2004.
- Hall R. S., Cervantes H., « An OSGi Implementation and Experience Report », IEEE CCNC conference, 2004.
- Hamilton G., « The JavaBeans Specification », Sun Microsystems, 1997.
- Hoareau D., Mahéo Y., « Middleware support for the deployment of ubiquitous software components », PUC conference, 2008.
- Iverson W., *Real Web services*, O'Reilly, 2004.
- Kichkaylo T., Ivan A., Karamcheti V., Sekitei : An AI planner for Constrained Component Deployment in Wide-Area Networks, Technical report, 2004a.
- Kichkaylo T., Karamcheti V., « Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications », 13th IEEE ISHPDC, 2004b.
- K.Kui, Z.Wang, « Software Component Models », IEEE TSE conference, 2007.
- Kumaran S., « JINI Technology An Overview », Prentice Hall PTR, 2002.
- Lacomme P., Prins C., Sevaux M., *Algorithmes de graphes*, Editions Eyrolles, 2003.
- Microsystems S., « Service Loader », <http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html>, n.d.
- OBR, « OBR Bundle Repository », <http://www.osgi.org/Repository/HomePage>, 2008.
- Park N., Lee K., Kim H., « A Middleware for Supporting Context-Aware Services in Mobile and Ubiquitous Environment », IEEE ICMB conference, 2005.
- Parker D., Cleary D., « A P2P approach to ClassLoading in Java », LNCS APPC workshop, 2004.
- Poladian V., Sousa J., Garlan D., Shaw M., « Dynamic configuration of resource-aware services », 26th ICSE conference, 2004.
- Taconet C., Putycz E., Bernard G., « Context-Aware Deployment for Mobile Users », 27th IEEE ICSAC conference, 2003.
- Zahavi R., « *Entreprise Application Integration with Corba Component and Web-Based solutions* », John Wiley & sons, 1999.