jOMiSCID, un intergiciel sous OSGi pour l'informatique ubiquitaire

Patrick Reignier, Sofia Zaidenberg, Rémi Emonet, Dominique Vaufreydaz, Julien Letesssier

jOMiSCID, un intergiciel sous OSGi pour l'informatique ubiquitaire

P. Reignier, S. Zaidenberg, R. Emonet, D. Vaufreydaz, J. Letesssier
Laboratoire Gravir
655 Avenue de l'Europe
38334 Saint Ismier Cedex, France
(reignier, zaidenberg, emonet, vaufreydaz, letessier)@inrialpes.fr

RESUME

Cet article présente un intergiciel multi-plateforme (Windows, Linux, MacOSX) et multi-langage (C++, Java) pour l'informatique ubiquitaire. Cet intergiciel permet de faire abstraction de la communication réseau. Il permet l'inspection de services distants et s'appuie sur DNS-SD[1] pour permettre leur découverte.. La version Java est disponible sous deux formes : un fichier `jar` "classique" et un bundle pour la mise en oeuvre sous OSGi. La version OSGi permet de tirer parti des avantages de cet environnement en répondant aux contraintes imposées par l'informatique ubiquitaire : facilité de déploiement à partir d'un serveur de composants, possibilité d'ajout, de suppression et de mise à jour "à chaud" des composants.

MOTS CLES

OSGi, intergiciel, découverte de services, informatique ubiquitaire

1. INTRODUCTION

L'espace de travail d'un utilisateur est actuellement constitué d'une station de travail proposant une vaste liste de services (logiciels) indépendants. Cela peut être par exemple l'agenda électronique, le lecteur mail, le navigateur web ou le traitement de texte. L'utilisateur, en fonction de sa tâche, choisit le logiciel qu'il souhaite utiliser et dialogue avec lui en utilisant le clavier et la souris. Ce schéma classique possède un certain nombre de limitations : dans l'évolution technologique actuelle, l'utilisateur ne dispose plus d'une seule station de travail mais peut interagir à l'aide de plusieurs dispositifs hétérogènes, éventuellement partageables et utilisables en coopération (PDA, téléphones portables, etc). La communication homme-machine est essentiellement explicite et basée sur le clavier et la souris. Dans le cadre de la communication homme-homme au contraire, une grande partie de l'information est transmise sans communication explicite [7]. Il peut s'agir d'expressions faciales, de relations avec les autres personnes ou des objets présents à proximité, etc. L'ensemble des éléments permettant de définir cette communication implicite est appelé le contexte. L'étude des deux points précédents fait partie du domaine de l'informatique ambiante. L'objectif de l'informatique ambiante est de faire disparaître l'informatique traditionnelle au profit d'un espace informatisé au service des activités humaines [16]. Cet espace informatisé doit prendre en compte la multiplicité des plateformes et percevoir le contexte afin de mieux comprendre et anticiper les besoins de l'utilisateur et être en mesure de proposer automatiquement les services appropriés.

Cet espace informatique globalisé nécessite la mise en oeuvre d'applications non plus monolithiques mais sous forme de composants répartis sur les différentes plateformes. Après un rapide tour d'horizon des intergiciels pour l'informatique ambiante, nous présenterons les objectifs de notre approche, ses caractéristiques principales et son intégration à OSGi. Nous terminerons avant de conclure par quelques exemples de services mis en oeuvre.

2. INTERGICIELS POUR L'INFORMATIQUE AMBIANTE

L'émergence rapide de l'informatique ubiquitaire a transformé la manière de concevoir les applications. Celles-ci sont désormais fréquemment distribuées sur un grand nombre d'ordinateurs possédant des caractéristiques techniques diverses (de serveurs haut de gamme aux smartphones et aux PDA). Construire des applications distribuées robustes représente un challenge pour le développeur. Les intergiciels ont pour objectif de faciliter cette tâche en masquant les détails de bas niveau.

Le concept d'intergiciels "zéro configuration" propose l'approche suivante : l'ensemble des composants de l'application sont des services déclarés sur le réseau. Lorsqu'un service nécessite un autre service pour fonctionner, il n'a pas besoin de connaître sa localisation (hôte et numéro de port). L'ensemble de ces informations lui sont fournies par un mécanisme de découverte de services en réponse à une requête basée sur le type ou les propriétés du service recherché.

Tous les intergiciels orientés services proposent un mécanisme de découverte. Certains d'entres eux imposent néanmoins de connaître où se trouve ce dépot centralisé nécessitant une configuration préalable lors du déploiement. Il existe essentiellement deux infrastructures zéro configuration : UPnP et Zeroconf.

UPnP (Universal Plug and Play [5]) propose un ensemble de fonctionnalités incluant SSDP (Simple Service Discovery Protocol) prenant en charge la découverte dynamique de services. UPnP est répendu, très attractif et répond à la plupart de nos contraintes. Il possède néanmoins une limitation forte par rapport aux applications que nous envisageons : il

[1]Environnement pour la découverte de services basé sur DNS

n'existe pas de mécanisme de notification "instantanée" de disparition d'un service. Il n'y a donc pas de moyen d'avoir une vue à jour de l'ensemble des services disponibles.

La seconde grande infrastructure de type zéro-configuration est Zeroconf [9]. Zeroconf repose sur DNS-SD [2] combiné à Multicast DNS [4] pour offrir une découverte de service distribuée ainsi qu'un système de notification rapide de connexion et déconnexion

Il existe également un certain nombre d'intergiciels dédiés (par exemple [14, 15, 3]) mais ceux ci sont soit trop spécialisés soit centrés uniquement Java. Nous avons donc décidé de contruire notre propre intergiciel au dessus de DNS-SD : OMiSCID.

3. OBJECTIFS

Nous allons détailler dans cette section les objectifs que nous nous sommes fixés pour notre intergiciel. Bien que celui-ci doive être le plus générique possible, nous avons néanmoins pris en compte les contraintes propres aux applications ubiquitaires.

3.1 Multi-langage, multi-plateforme

Les composants d'une application ubiquitaire doivent pouvoir se répartir sur les différentes plateformes présentes dans l'environnement : ordinateur de bureau, PDA, smartphone, etc. Ces composants peuvent être proches du matériel (driver X10, acquisition vidéo ou sonore, utilisation du processeur de la carte graphique pour le traitement d'images). Ils peuvent également être de haut niveau (reconnaissance de contexte, moteur de raisonnement onthologique, IHM, etc). Ces différents niveaux, ainsi que la culture et les habitudes du développeur vont avoir un rôle dans le choix du langage de programmation retenu pour un composant. Il est donc important que l'intergiciel soit nativement conçu pour être à la fois multi-langage et multi-plateforme.

3.2 Coût réseau

Les applications considérées sont des applications interactives (l'utilisateur est présent dans le système) impliquant entre autre des traitements d'images et de son. Pour garantir le caractère interactif, il est nécessaire de disposer d'un système de communication proposant une latence la plus faible possible. L'intergiciel doit donc être léger en termes de surcharge des communications entre les composants.

3.3 Robustesse

Nous nous plaçons dans le cadre d'applications devant fonctionner en permanence. Une approche du type intergiciel centralisé sur un serveur offrant ses services à un réseau d'ordinateurs clients ne peut donc pas convenir : la chute du serveur entraîne le blocage de l'ensemble des composants. Il est donc nécessaire d'envisager des solutions décentralisées.

3.4 Couplage faible

Le type d'applications que nous considérons met en jeu des composants répartis sur un ensemble dynamique d'ordinateurs : les systèmes (en particulier les PDAs et les smartphones) peuvent apparaître et disparaître en fonction des déplacements des utilisateurs. Il est donc nécessaire que le couplage entre les composants soit le plus faible possible afin d'être robuste à la disparition brutale de l'un d'entre eux. De même, l'ensemble des composants présents dans le système ne peut être connu statiquement. Lorsqu'un utilisateur entre par exemple dans une salle de réunion, son PDA doit pouvoir déterminer l'ensemble des composants disponibles (par exemple : contrôle du projecteur, visionneuse powerpoint etc) au niveau de la salle afin de pouvoir les exploiter. Les composants doivent donc pouvoir se déclarer auprès d'un annuaire. Pour des raisons de robustesse (cf section 3.3), cet annuaire ne doit pas être centralisé mais distribué.

4. OMISCID ET JOMISCID

Nous allons maintenant présenter brièvement l'architecture de notre intergicie (OMiSCID et sa version Java jOMiSCID)l. Plus de détails sont disponibles dans [8].

L'intergiciel est constitué actuellement de 2 couches : une couche communication réseau, et une couche service. Une troisième couche (onthologie) est en cours de conception (cf perspectives).

4.1 Couche 1 : Communication réseau

La couche la plus basse est une couche fine de gestion des communications réseau. Nous utilisons BIP (Basic Interconnection Protocol [10]) qui découpe un flot en un ensemble de messages. Le contenu d'un message peut être un ensemble de données binaires, un texte ASCII ou encore de l'XML. De manière à garantir un coût réseau faible (cf section 3.2), le surcoût imposé par l'intergiciel n'est que de 38 octets par message.

4.2 Couche 2 : Services

La couche service est construite au-dessus de la couche communication. Un service possède un nom unique sur le réseau. Il expose un ensemble de connecteurs. Ces connecteurs peuvent être de type entrée, sortie, ou entrée / sortie. Un service expose également un ensemble de variables applicatives représentant son état. Une variable peut être en lecture seule ou en lecture / écriture. Un service distant peut s'abonner de manière à être informé de la modification de la valeur d'une variable. L'ensemble des connecteurs et des variables d'un service peut être décrit dans un fichier `service.xml`. Ce fichier est chargé au démarrage du service et décharge le programmeur de la création de ces structures.

Chaque service possède au moins un connecteur appelé connecteur de contrôle. Ce canal permet l'inspection d'un service par un service distant.

Pour pouvoir être découvert, un service doit se publier dans un annuaire. De manière à satisfaire notre critère de robustesse, nous souhaitons avoir recours à une solution de type annuaire distribué (cf section 3.4). Nous avons retenu DNS-SD [2], appelé aussi Bonjour ou Rendez-Vous (cf Apple).

DNS-SD propose un mécanisme très réactif de notification d'apparition et de disparition de services. La notification de disparition permet entre autre de satisfaire la contrainte de couplage faible.

Remarque : ces deux couches sont disponibles en C++, en Java et en Tcl. L'implémentation Java (jOMiSCID) n'est

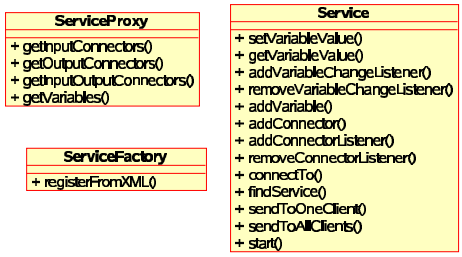

Figure 1: Schéma UML simplifié des interfaces exportées par le bundle jOMiSCID

pas une couche JNI de la version C++ mais une réécriture complète en Java pur.

5. IMPLÉMENTATION SOUS OSGI

La version java de l'intergiciel (jOMiSCID) permet de développer des composants s'exécutant dans leur propre machine virtuelle. Elle permet d'accéder aux différentes couches du système. Nous avons également réalisé à partir de jOMiSCID une version disponible sous forme de bundle OSGi afin de pouvoir tirer parti des avantages de cette plateforme : facilité de déploiement des composants, gestion propre du cycle de vie, facilité de mise à jour ...

5.1 Principe

Le bundle OSGi incorpore en interne la version java de l'intergiciel : `jOMiSCID.jar`. Afin de proposer une séparation claire entre les concepts et l'implémentation, les packages ne sont pas exportés. Une fine couche d'abstraction est mise en place sous forme d'une interface exportée et de son implémentation. Cette couche a pour objectif de faire le lien entre les bundles souhaitant utiliser jOMiSCID et jOMiSCID lui même. Cette abstraction cache les différentes couches et ne reprend que les concepts de plus haut niveau en en offrant une vue simplifiée (voir figure 1).

Un service OMiSCID est construit par un `ServiceFactory` en lui fournissant son fichier de description `service.xml`. Le service est publié sous DNS-SD dès l'invocation de sa méthode `start()`. Les variables et les connecteurs décrits dans le fichier sont automatiquement créés. Le `ServiceFactory` retourne un objet de type `Service`. Cette instance permet de réaliser les opérations suivantes : ajout d'une variable, lecture ou modification de sa valeur, enregistrement d'un listener sur modification de la variable, création d'un nouveau connecteur, enregistrement d'un listener sur réception d'un message dans un connecteur, recherche d'autres services publiés dans DNS-SD, envoi d'un message dans un connecteur.

La recherche d'un service est réalisée grâce à la méthode `findService()`. Cette méthode interroge DNS-SD et retourne une instance de type `ServiceProxy`. Cette instance est la vision locale du service distant. Elle permet en particulier d'inspecter les caractéristiques de ce service en termes de variables et de connecteurs disponibles. La méthode `findService()` prend comme argument un filtre permettant de sélectionner le service souhaité. Des exemples de filtres sont le nom du service, le nom du propriétaire (la personne ayant lancé le service), le nom de connecteurs ou de variables, un *et* entre deux autres filtres (pour permettre la cascade des filtres), etc.

La liaison entre un service OMiSCID (un bundle OSGi voulant utiliser OMiSCID : on l'appellera dorénavant un service applicatif) et le bundle OMiSCID proprement dit est réalisée par le service binder [6]. La liaison est de type *static* (le bundle `OMiSCID` fait partie de l'infrastructure et n'est pas censé disparaître en cours d'exécution).

Lorsqu'un service applicatif ajoute un listener sur un connecteur ou une variable, il transmet un objet de son bundle vers le bundle `OMiSCID`. Il est donc nécessaire lors de l'arrêt ou de la mise à jour du service applicatif de désenregistrer au préalable tous les listeners afin de ne pas laisser "d'anciennes" versions d'objets au sein du bundle OMiSCID.

Remarque : cette gestion de cycle de vie (déclaration du service, désenregistrement des listeners) nécessite l'écriture de code "non applicatif" par le développeur du service applicatif. Cette écriture aurait pu être reportée du côté du bundle OMiSCID en ayant recours à un pattern de type inversion de contrôle [11]. Nous avons plutôt choisi une approche de type génération automatique de ce code par l'environnement de développement.

5.2 Environnement de développement

Afin de faciliter le développement de services applicatifs, nous avons mis en place des annotations des classes Java ainsi qu'un plugin Eclipse permettant de générer automatiquement une partie du code non applicatif lors de la création d'un projet.

5.3 Annotation des classes

La création d'un service applicatif nécessite d'une part la conception et le développement de classes métiers, et d'autre part la mise en place de la liaison entre ces classes métiers et OMiSCID. Cette liaison consiste en des classes implémentant les listeners de connecteurs et de variables. Dans le cas des connecteurs, il faut rediriger les appels vers les méthodes des classes métiers concernées. Dans le cas des variables, il faut convertir la variable OMiSCID (chaîne de caractères) vers le type correct avant de transmettre sa valeur vers les instances concernées. De manière à faciliter le développement, nous avons ajouté la possibilité d'annoter directement les classes métiers afin de générer la couche de liaison avec OMiSCID. Nous avons choisi le système d'annotations *Spoon* [12]. Spoon permet de modifier le code source de la classe annotée avant la compilation. Les annotations mises en place sont les suivantes :

- `@Service(descriptionFileName = "service.xml")`. Cette annotation est au niveau de la classe. Elle permet de spécifier la classe servant de "point d'entrée" pour le service. Le paramètre de l'annotation est le nom du fichier de description.
- `@Variable(name = "varName")`. Cette annotation est associée à un attribut. Le paramètre indique le nom de la variable OMiSCID liée à l'attribut Java. Toute modification de la variable OMiSCID est retranscrite sur l'attribut Java. Toute affectation de l'attribut est repropagée sur la variable OMiSCID.

- `@ConnectorInput(name = "connectorName")`. Cette annotation est au niveau méthode. Elle permet d'indiquer que la méthode doit recevoir les messages en provenance du connecteur spécifié en paramètre. Il existe de même une annotation pour les connecteurs en sortie. Les connecteurs en entrée-sortie nécessitent l'annotation de deux méthodes : une en entrée (`@ConnectorInput`) et une en sortie (`@ConnectorOutput`).

5.4 Plugin Eclipse

De manière à faciliter la mise en place d'un nouveau projet, nous avons réalisé un plugin Eclipse [13] proposant un wizard de création d'un projet de type OMiSCID. Ce wizard met en place les différents fichiers associés et les configure en fonction des informations saisies. Il génère aussi le code de gestion du cycle de vie (voir section 5.1).

5.5 Quelques exemples de services

Quelques exemples de services que nous avons développés : *Text2Speech* (synthèse vocale du texte envoyé sur le connecteur d'entrée), *VncService* (démarrage et d'arrêt d'un client VNC en lui fournissant l'adresse du serveur où se connecter), *RemoteShell* (contrôle à distance d'une plateforme OSGi) et *DcopService* (pilotage des applications Kde en interprétant des ordres Dcop [1])

Exemple de scénario mettant en oeuvre ces services : l'utilisateur entre dans un bureau. Le composant agenda de son PDA détecte un rendez-vous proche et souhaite prévenir la personne. Le composant détecte la présence d'un ordinateur de bureau équipé d'une carte son. Le PDA contacte le composant *RemoteShell* de ce poste fixe, lui demande d'installer et de démarrer *Text2Speech* à partir du serveur de composants, se connecte au composant nouvellement lancé et lui transmet le message à destination de l'utilisateur. De manière similaire, on peut imaginer que lorsque l'utilisateur entre dans la pièce et s'approche d'un poste de travail libre (données fournies par un tracker vidéo ou par une antenne Bluetooth par exemple), le PDA demande à ce poste de charger le composant *VncService* et lui transmet l'adresse du serveur associé à cet utilisateur. De même, lorsque l'utilisateur s'éloigne, la session est automatiquement refermée. Cela permet de mettre en place une migration automatique de session.

6. CONCLUSION ET PERSPECTIVES

Nous avons présenté dans cet article un intergiciel multi-langage et multi-plateforme : OMiSCID. Cet intergiciel s'appuie sur DNS-SD (cf Bonjour d'Apple, précédemment appelé Rendez-Vous) pour l'enregistrement et la découverte des services. Nous l'avons embarqué au sein d'un bundle OSGi afin de pouvoir utiliser cette plateforme comme support d'exécution de nos services Java et de tirer profit de ses avantages, tels que la mise à jour à chaud des composants et la facilité de déploiement.

Dans les perspectives à court terme, nous considérons les deux axes suivants :

- Amélioration de l'intergiciel : un service est actuellement décrit par un nom et une liste de noms de variables et de connecteurs. Ces propriétés sont utilisées par les services pour se rechercher et constituent donc la base de connaissance partagée par l'ensemble des composants du système. Cette représentation de connaissances est assez pauvre. Nous souhaitons avoir recours aux ontologies afin d'enrichir la description de nos services et permettre la mise en place de raisonnement pour leur sélection et leur inter-connexion..
- La communication entre deux services passe actuellement par les couches réseaux, aussi bien si les deux services sont sur deux JVMs distinctes ou au sein de la même. Afin de minimiser les coûts de communication, nous allons substituer automatiquement les appels réseaux par des appels de méthodes directs entre deux services lorsque ceux-ci seront présents sur la même machine virtuelle.

7. REFERENCES

[1] Dcop : Desktop communication protocol. http://developer.kde.org/documentation/other/dcop.html.

[2] Dns-sd : Dns service discovery. http://www.dns-sd.org.

[3] Jini network technology. "http://wwws.sun.com/software/jini/".

[4] Multicast dns. "http://www.multicastdns.org".

[5] Upnp device architecture. "http://www.upnp.org/download/UPnPDA10_20000613.htm", 2000.

[6] Humberto Cervantes. Service binder : Simplifying application development on osgi platform. http://www.humbertocervantes.net/servicebinder/index.html.

[7] A.K. Dey and G.D. Adowd. The context toolkit : Aiding the development of context-aware applications. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[8] Rémi Emonet, Dominique Vaufreydaz, Patrick Reignier, and Julien Letessier. O3miscid: an object oriented opensource middleware for service connection, introspection and discovery. In *1st IEEE International Workshop on Services Integration in Pervasive Environments*, June 2006.

[9] Erik Guttman. Autoconfiguration for ip networking: Enabling local communication. *Internet IEEE Computing*, pages 81–86, June 2001.

[10] J. Letessier and D. Vaufreydaz. Draft spec : Bip/1.0 - a basic interconnection protocol for event flow services. http://www-prima.imag.fr/prima/pub/Publications/2005/LV05/, 2005.

[11] R.C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Pearson Education, 2002. ISBN 0-13-597444-5.

[12] Renaud Pawlak, Nicolas Petitprez, and Carlos Noguera. Spoon. http://spoon.gforge.inria.fr/.

[13] Patrick Reignier. Plugin eclipse pour la création d'un projet omiscid. http://www-prima.inrialpes.fr/reignier/update-site/.

[14] Da Qing Zhang Tao Gu, Hung Keng Pung. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, 2005.

[15] M.-T. Tran, B. Hirsbrunner, and M. Courant. A context-aware middleware for multimodal dialogue applications with context tracing. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8, New York, USA, 2005. ACM-Press.

[16] M. Weiser. The computer of the 21st century. *Scientific American*, 3(265), 1991.