Équipement plug and play et plateforme de service : L'informatique pervasive au sein du réseau domestique

SIMON Éric

FRANCE TELECOM R&D
28, chemin du vieux chêne,
38240 Meylan. FRANCE
simon.eric3@gmail.com

Abstract

The Home Network is designated to be an pervasive environment. by nature. Indeed, such network is composed of many dynamic and heterogeneous devices; these last have all aspect of embedded systems.

In our case, to use a service oriented approach allows offering the loose coupling required to the home environment.

We will address the integration of these devices in an $OSGi^{TM}$ framework running over $Java^{TM}$, and its advantages for the applications. Finally, we will present a possible overture for the continuation project.

Par nature, le réseau domestique est voué à être un environnement pervasif. En effet, un tel réseau est composé de nombreux équipements hétérogènes et dynamiques ; ces derniers ont tous les aspects des systèmes embarqués.

Dans notre cas, utiliser une approche orientée service nous permet d'offrir le couplage lâche requis par l'environnement domestique.

Nous aborderons l'intégration de ces services au sein du framework $OSGi^{TM}$ s'exécutant sur $Java^{TM}$, et les avantages que celui-ci comporte pour ces applications. Finalement, nous présenterons des ouvertures possibles pour la suite du sujet.

Table des matières

0.1 Terminologie 4

1 Introduction **6**
1.1 Contexte 6
1.2 Problématique 7
1.3 Organisation du document 7

2 État de l'art **9**
2.1 (Web) Service Oriented Architecture 9
2.1.1 Avec annuaire centralisé (fournisseur, consommateur, répertoire) . . 9
2.1.2 Décentralisé (Fournisseur, Consommateur) 10
2.1.3 Fonctionnalités requises des (W)SOA 10
2.1.4 Notion de service (au sens WSOA) 11
2.1.5 Mode de découverte (SOA non centralisé) 11
2.2 Protocoles plug and play 11
2.2.1 $UPnP^{TM}$ 11
2.2.2 DPWS 16
2.2.3 Comparaison 21
2.3 Plateforme domestique de services 22
2.3.1 $OSGi^{TM}$ 22
2.4 Device Access 27
2.4.1 Fondamentaux 27
2.4.2 $UPnP^{TM}$ Base Driver 28

3 SmartPnP project : *DPWS* base driver **31**
3.1 Cas d'utilisation 31
3.2 Problèmatique 32
3.3 Solutions 33
3.3.1 Lazy/Immediate Discovery 33
3.3.2 Lazy/Immediate Loading 34
3.3.3 Lazy/Immediate Networking 34
3.3.4 Import et export de *device* 35
3.3.5 *White Board* pattern 35
3.3.6 Architecture 36

3.4 Tests d'efficacité 38
3.4.1 Description des conditions de test 38
3.4.2 Résultats 38
3.4.3 Analyse des résultats 39

4 Conclusion et Perspective **40**
4.1 Contribution 40
4.2 Perspective 40

5 Référence / Bibliographie **42**
5.1 Bibliographie 42
5.2 Spécification 43
5.3 Référence technique 44

Table des figures

2.1 Exemple : SOA centralisée 10
2.2 Exemple : SOA décentralisée 10
2.3 Représentation d'un *device* $UPnP^{TM}$ 12
2.4 Représentation d'un service $UPnP^{TM}$ 13
2.5 Pile $UPnP^{TM}$ 14
2.6 Message de disponibilité d'un *device* : Cycle 15
2.7 Exemple d'architecture basée sur DPWS 16
2.8 Représentation d'un *device* DPWS 17
2.9 Représentation d'un service DPWS 18
2.10 Pile DPWS 18
2.11 Message de disponibilité d'un *device* : Cycle 20
2.12 Exemple réseau domestique 22
2.13 Vue en "couches" de la structure d'$OSGi^{TM}$ 23
2.14 Cycle de vie d'un bundle 24
2.15 Exemple de bundle, requérant et fournissant services et packages, avec ses composants 26
2.16 Utilisation idéale des services dans $OSGi^{TM}$ 26
2.17 $UPnP^{TM}$ base driver 29

3.1 Interactions *device* / *control point* 31
3.2 Catégorisation (cf. [2.]) de *DPWS* 33
3.3 Interface DPWS 35
3.4 Architecture globale du système 36
3.5 Architecture du base driver et de l'API 37
3.6 Test d'efficacité 38

0.1 Terminologie

1. **API :** Application Programming Interface ;
2. **AV :** Audio-visual ;
3. **DAAP :** Digital Audio Access Protocol ;
4. **DHCP :** Dynamic Host Configuration Protocol ;
5. **DNS-SD :** Domain Name System - Service Discovery ;
6. **DLNA :** Digital Living Network Alliance ;
7. **DP :** Discovery proxy ;
8. **DPWS :** Device Profile for Web Services ;
9. **GENA :** General Event Notification Architecture ;
10. **HTTP :** Hypertext Transfer Protocol ;
11. **HTTPMU :** Hypertext Transfer Protocol Multicast ;
12. **HTTPU :** Hypertext Transfer Protocol Unicast ;
13. **IETF :** Internet Engineering Task Force ;
14. **IGD :** Internet Gateway Device ;
15. **IP :** Internet Protocol ;
16. **ITEA :** Information Technology for European Advancement ;
17. **mDNS :** multicast Domain Name System ;
18. $OSGi^{TM}$ **:** Open Services Gateway initiative (obsolète, s'utilise maintenant en adjectif) ;
19. **PDA :** Personal Digital Assistant ;
20. **POJO :** Plain Old Java Object ;
21. **RFC :** Request For Comment ;
22. **RFP :** Request For Proposal ;
23. **RPC :** Remote Procedure Call ;
24. **SCPD :** Service Control Protocol Definition ;
25. **SOA :** Service Oriented Architecture ;
26. **SOAP :** Simple Object Access Protocol ;
27. **SOHO :** Small Office Home Office ;
28. **SSDP :** Simple Service Discovery Protocol ;
29. **SSL :** Secure Sockets Layer ;
30. **TCP :** Transmission Control Protocol ;
31. **UDDI :** Universal Description, Discovery and Integration ;
32. **UDP :** User Datagram Protocol ;
33. $UPnP^{TM}$ **:** Universal Plug and Play ;

34. **UTL :** $UPnP^{TM}$ Template Language ;
35. **WS :** Web Service ;
36. **WSDL :** Web Service Description Language ;
37. **WSOA :** Web- Service Oriented Architecture.

Chapitre 1

Introduction

Il y a dix ans de cela, Mark Weiser avait déjà mis en évidence le paradoxe suivant :

« The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. » (cf. [1.])

Spécialement dans le cas de l'informatique pervasive. Notre but est de faciliter la vie de l'utilisateur, et accessoirement celui du développeur. Or l'un des domaines de prédilection est le réseau domestique, où il existe un ensemble hétérogène d'équipements complexes. Un des enjeux pour le futur est de les interconnecter afin qu'ils interagissent dans le but de s'autogérer collectivement ou indépendamment, proposer des services ciblant l'utilisateur, etc Afin de les interconnecter, il est nécessaire de cibler un protocole permettant le déploiement d'équipement complexe.

1.1 Contexte

Par le passé, beaucoup d'industriels ont produit des équipements électroniques communicants pour l'environnement domestique (systèmes vidéo/audio, etc). Dans un premier temps nombreux ont imaginé des solutions propres à leurs besoins. Chacun a spécifié son propre format de données, ses propres moyens de communications, etc. Parallèlement la complexité des équipements a crû rapidement ce qui a entrainé une multiplication des supports de communications, des capteurs et des terminaux au sein de la maison. Cette multiplication d'équipement a considérablement complexifié la gestion et la configuration de ces derniers par l'utilisateur.

Pour les équipements domestiques IP, une convergence vers un protocole est apparu vers l'année 2001. En effet, un groupe d'équipementier : l'$UPnP^{TM}$ *Forum*, s'est fondé en octobre 1999 autour de la spécification du protocole $UPnP^{TM}$ proposée par Microsoft©. Il compte actuellement 836 membres (industriels, etc), et propose de nombreuses spécifications pour des équipements IP. Ces spécifications, certaines proposées par *Digital Living Network Alliance* (DLNA), ont pour but de rendre les différents équipements interopérables. Par conséquent le protocole $UPnP^{TM}$ est actuellement le plus utilisé par les industriels car celui-ci spécifie l'ensemble des couches logiques nécessaires à la production d'un

équipement IP allant de la couche réseau jusqu'à la spécification de type d'équipement (AV, IGD, etc).

En mai 2004, Microsoft© publie la spécification du *DPWS* : *Device Profiles for Web-Services*. En effet, le protocole $UPnP^{TM}$ montre des limitations dans son utilisation, tel que :

- le passage à l'échelle au sein d'un réseau ;
- une description des équipements simpliste ;
- , etc

Le but de Microsoft© était que *DPWS* soit la version 2 de $UPnP^{TM}$; cependant l'$UPnP^{TM}$ *Forum* n'a pas été de cet avis. Malgré cela, Microsoft© a intégré la technologie *DPWS* dans la suite de service *Windows Rally* de son nouveau système d'exploitation : *Windows Vista*.

1.2 Problématique

L'intégration d'équipements IP au sein d'un réseau domestique soulève un ensemble de problèmes :

- **hétérogénéité** : les équipements domestiques ont des types et des domaines différents (systèmes AV, surveillance, etc) ;
- **disponibilité** : leur disponibilité varie dans le temps ; en effet ils peuvent être mis en veille par l'utilisateur ou disparaitre du réseau : l'utilisateur sort de chez lui avec son téléphone portable (équipement possible), etc ;
- **dynamicité** : outre le fait de leur disponibilité sur le réseau, ceux-ci interagissent avec leur environnement : l'utilisateur et les autres équipements domestiques :
- **interopérabilité** : afin d'interagir avec d'autres, un équipement doit être conforme à une spécification.

Les équipements forment un ensemble hétérogène. Leur disponibilité sur le réseau varie dans le temps ; durant leur disponibilité les équipements interagissent avec d'autres de nombreuses fois.

Les technologies *plug and play*, tel qu'$UPnP^{TM}$ ou *DPWS*, répondent au besoin d'un environnement dynamique et autonome permettant de faire communiquer des *devices* cohabitant sur un même réseau. En couplant une technologie plug and play avec une passerelle domestique, il peut permettre aux équipementiers de faire évoluer (*update*, *patch*, etc) les équipements ou offrir de nouveaux services. De plus il permet à l'utilisateur de gérer/configurer l'ensemble de ses équipements via la passerelle.

1.3 Organisation du document

Les sections suivantes détaillent un ensemble de protocoles, concepts, technologies et frameworks nécessaire pour le déploiement d'une passerelle domestique intégrant une ou plusieurs technologies *plug and play*.

Par la suite, nous aborderons l'intégration, déjà existante, de la technologie $UPnP^{TM}$ dans le framework $OSGi^{TM}$; puis nous détaillerons l'intégration de la technologie *DPWS* pour ce même framework.

Chapitre 2

État de l'art

2.1 (Web) Service Oriented Architecture

La notion d'architecture orientée service (SOA) est un paradigme abstrait dans le cas des SOA, ce qui n'est pas nécessairement le cas des WSOA, sans aucune référence à une implémentation technique.

L'objectif d'une architecture orientée services est de décomposer une fonctionnalité complexe d'un système en un ensemble de sous-fonctions qui sont nommées services. L'idée sous-jacente est de construire une architecture décomposée en services correspondant à différents processus métiers d'un système.

Une architecture orientée service possède par nature un *couplage lâche* ; c'est à dire que les services d'un système sont reliés par un réseau de communication mais qu'ils poursuivent leur propre logique de fonctionnement. En d'autres termes : ses services sont indépendant. Par conséquent une application ayant une architecture orientée service est plus souple (remplacement dynamique de service...) et plus ouverte (extensibilité, maintenabilité...).

Deux possibilités, l'architecture orienté service :

- est centralisée : elle s'appuie sur un service de registre de service ;
- est décentralisée : elle s'appuie sur un mécanisme de découverte de service en multi-cast sur le réseau.

2.1.1 Avec annuaire centralisé (fournisseur, consommateur, répertoire)

Prenons l'exemple du fournisseur/consommateur : dans un premier temps, le fournisseur publie sa description à un service de registre (par exemple UDDI), nommé répertoire dans le schéma ci-dessus. Lorsqu'un consommateur a besoin d'un service, il recherche dans ce même répertoire les descriptions des services disponibles, il récupère la référence qui l'intéresse (le premier ou en fonction d'un filtre de recherche). Une fois la référence récupéré, il établit une communication avec le service désiré.

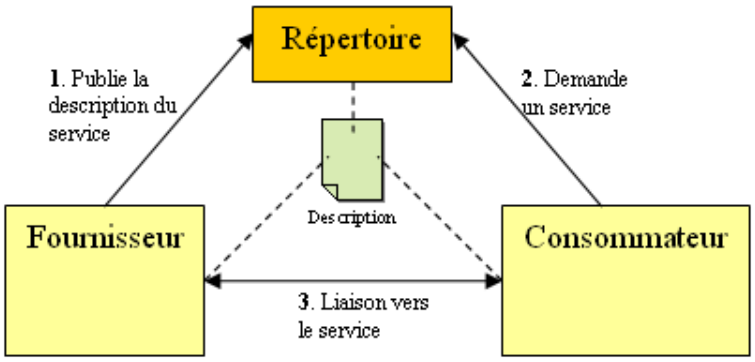

FIG. 2.1 – Exemple : SOA centralisée

2.1.2 Décentralisé (Fournisseur, Consommateur)

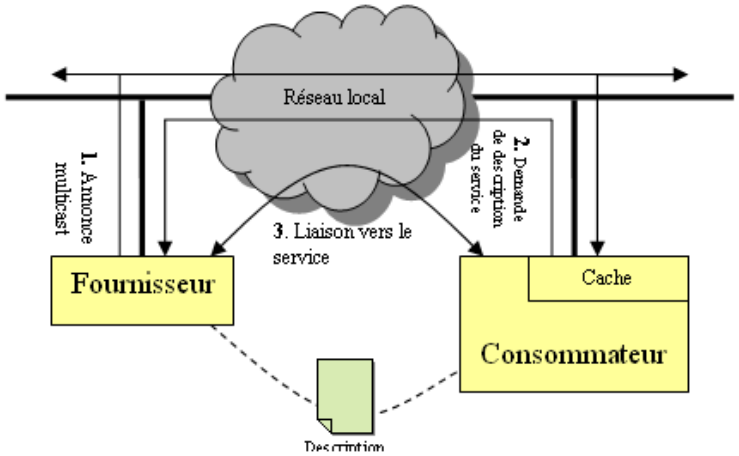

FIG. 2.2 – Exemple : SOA décentralisée

Dans le cas d'une SOA décentralisée, la découverte de service ne se fait plus en se connectant à un service de registre. L'enregistrement dans le registre est remplacé par une annonce (message Hello) en multicast. Si un consommateur est intéressé il demande la description du service, et établit une connexion vers le service désiré.

2.1.3 Fonctionnalités requises des (W)SOA

Par conséquent, les architectures orientées service doivent fournir les fonctionnalités de :

- **publication (advertising) :** publie dans un registre/ou émet les services disponibles aux utilisateurs ;
- **découverte (discovery) :** permet la découverte sur le réseau de services qui ont été publiés ;
- **invocation :** qui effectue la connexion et les communications nécessaire à une action entre un service et un client.

2.1.4 Notion de service (au sens WSOA)

Un service est une interface bien définie implémentée par un composant ; il est substituable, il a une disponibilité dynamique et s'auto-contient. L'implémentation d'un service est encapsulée dans un composant. Un service peut être invoqué à l'aide d'une requête contenant un ou plusieurs paramètres, il renvoie un ou plusieurs messages de réponse.

Un service est la "terminaison" (endpoint) d'une connexion. Ce qui implique qu'un service doit implémenter certains types de protocole qui dépendent de la connexion.

2.1.5 Mode de découverte (SOA non centralisé)

Il existe deux types de découverte et deux manières de procéder pour chaque type : multicast ou service de registre ; par ailleurs, il existe des protocoles qui prévoient de passer d'un mode à l'autre en fonction du contexte (par exemple la découverte de la présence d'un service de registre sur le réseau) :

- **La découverte passive :**
  Le client reçoit des notifications de la part des *devices* (ou du registre) lorsqu'un service apparaît sur le réseau ou devient indisponible.
- **La découverte active :**
  Le client émet une requête de recherche (pour des *devices* ciblés ou non) vers le canal multicast ou le registre, et reçoit en retour (de façon synchrone ou asynchrone selon les cas) les descriptions des services correspondants.

Notons que ces deux modes sont souvent utilisés de façon couplée : un client commence par faire une découverte active pour s'informer de tous les *devices* présents et remplir un cache local, puis utilise la découverte passive pour mettre à jour ce cache.

2.2 Protocoles plug and play

2.2.1 $UPnP^{TM}$

En 1999, Microsoft© avait lancé $UPnP^{TM}$. Le $UPnP^{TM}$ Forum est formé en octobre 99, il est composé de 836 membres dans le domaine de l'informatique, de l'impression, de l'électronique, ... Le but du Forum est de permettre la connexion d'équipement de manière transparente et de faciliter leur implantation dans l'environnement domestique.

On distingue dans $UPnP^{TM}$ 3 entités :

- les équipements ;
- les services ;

– les points de contrôle.

Les équipements(*device*)

Un équipement, que l'on appellera désormais *device*, fournit des services. Il peut être composite, et donc être composé par d'autres *devices* (atomiques ou composites).

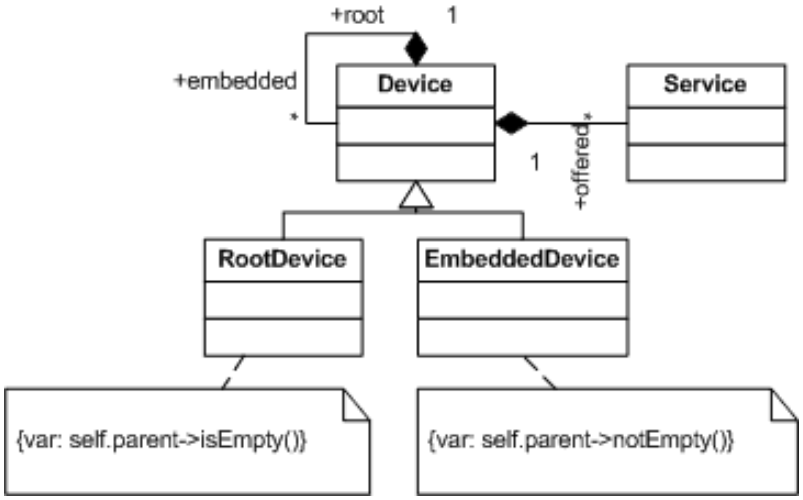

FIG. 2.3 – Représentation d'un *device* $UPnP^{TM}$

Prenons l'exemple d'un téléphone portable qui possède les fonctionnalités d'un appareil photo et un lecteur mp3. Cet équipement peut être considéré comme composite. Dans la terminologie $UPnP^{TM}$, cet équipement est un *device* racine (root device) qui contient des *devices* embarqués (embedded devices).

Les services (*service*)

Un service regroupe un ensemble d'actions et de variables d'état d'un *device*. Les actions sont utilisées par des composants clients (control points), elles sont appelées à l'aide de requêtes SOAP. Elles peuvent accéder (lecture/écriture) à une ou plusieurs variables d'états. Par conséquent, un service se compose d'une table d'état, d'un serveur d'action et d'un serveur d'évènement qui envoie des notifications aux *devices* qui s'abonnent.

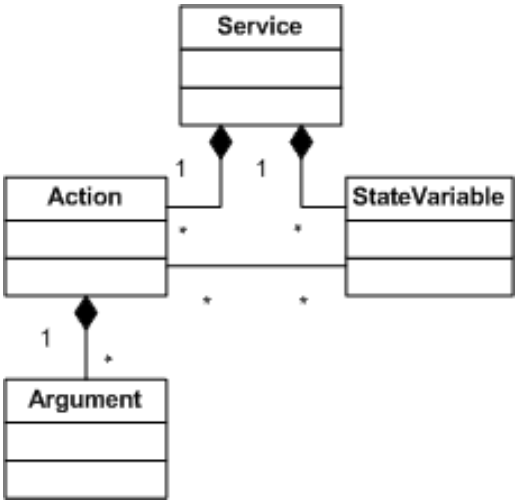

FIG. 2.4 – Représentation d'un service $UPnP^{TM}$

Les points de contrôle (*control point*)

Les points de contrôle permettent la découverte des services des *devices* afin d'utiliser les actions correspondantes.

La pile $UPnP^{TM}$

Les **producteurs de logiciel** utilisant $UPnP^{TM}$, les comités de travail de **l'**$UPnP^{TM}$ **Forum** et le document de **l'architecture de Device** $UPnP^{TM}$ définissent les couches les plus hautes des protocoles utilisés pour l'implémentation. Basé sur l'architecture des *devices*, les comités de travail définissent les informations générales pour spécifier les types des *devices* tels que :

- *Internet Gateway Device (IGD)* ;
- *MediaServer et MediaRenderer* ;
- ...

IP : La pile de protocoles $UPnP^{TM}$ était basé originellement sur le protocole IPv4, par la suite elle a été étendue afin de supporter le protocole IPv6.

HTTP,HTTPMU,HTTPU : TCP/IP et UDP/IP fournissent la base de la pile de protocole pour fournir la connectivité entre le réseau et les équipements $UPnP^{TM}$. HTTP , qui est largement responsable dans le succès d'internet, est aussi une partie primordiale de $UPnP^{TM}$. Tous les aspects de $UPnP^{TM}$ sont construits sur le protocole HTTP ou sur une de ses variantes.

HTTPU et HTTPMU sont des variantes de HTTP pour délivrer des messages en UDP/IP à l'instar de TCP/IP. Leur spécification n'est pas standardisé. Ces protocoles

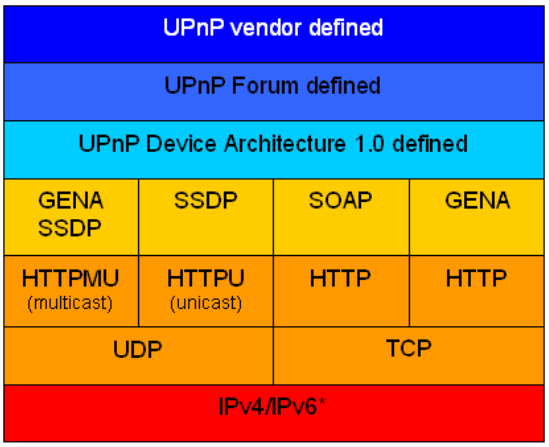

FIG. 2.5 – Pile $UPnP^{TM}$

sont utilisés par SSDP (décrit plus loin). Leur format de message colle avec HTTP et permet la communication multicast.

SSDP : *Simple Service Discovery Protocol* définit comment des services peuvent être découverts sur le réseau. SSDP est construit sur HTTPU (HTTP Unicast) et HTTPMU (HTTP MUlticast). Il définit des méthodes pour qu'un point de contrôle localise des ressources l'intéressant ; mais aussi pour que des équipements annoncent leur disponibilité sur le réseau.

En définissant l'utilisation des requêtes de recherche et des annonces de présence, SSDP élimine les coûts qui auraient été nécessaire si un seul de ces mécanismes avait été utilisé. En conséquence, chaque point de contrôle présent sur le réseau possède l'information complète sur l'état du réseau, tout en maintenant le trafic de bas-niveau.

Les points de contrôle peuvent envoyer une requête de recherche SSDP (via HTTPMU), pour découvrir les équipements et les services qui sont disponible sur le réseau. De plus ils peuvent affiner la recherche afin de trouver des équipements ou des services de type particulier ou même un équipement particulier.

Les équipements $UPnP^{TM}$ écoutent le port multicast 1900 à l'adresse 239.255.255.250. Lors de la réception d'une requête de recherche, l'équipement examine s'il est concerné. Si c'est le cas, il envoie une réponse SSDP, en unicast, au point de contrôle (via HTTPU).

De manière similaire, un équipement, connecté au réseau, enverra des annonces de présence de SSDP en multicast, pour annoncer les services qu'il fournit.

L'annonce de présence et les messages de réponse des équipements contiennent un pointeur sur le document de description de l'équipement. Ce document contient les informations sur l'ensemble des propriétés et des services supportés de l'équipement.

En plus des possibilités de découverte fournies, SSDP fournit également une manière pour les équipements et les services associés de se déconnecter du réseau "proprement"

(BYE notification). Ponctuellement, un message alive est envoyé pour annoncer que l'équipement est toujours actif. À chaque réception de message alive, le timeout, correspondant au *device*, est réinitialisé. À l'expiration du timeout, les informations correspondant au *device* sont purgées, car celui-ci est considéré comme inactif.

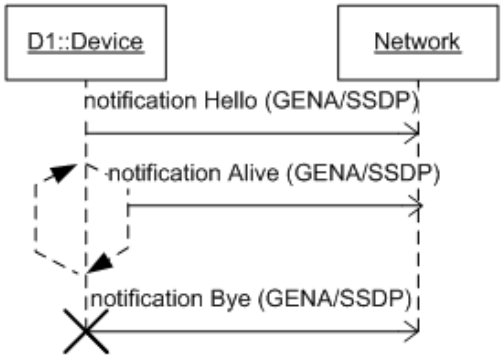

FIG. 2.6 – Message de disponibilité d'un *device* : Cycle

GENA : *Generic Event Notification Architecture* définit un protocole permettant d'envoyer et de recevoir des notifications utilisant HTTP sur TCP/IP et HTTPMU sur UDP multicast. GENA définit aussi le concept de subscriber/publisher des notifications pour permettre des évènements.

Des formats de GENA sont utilisés dans $UPnP^{TM}$ pour créer les annonces de présence à envoyer en utilisant SSDP, ainsi que pour fournir l'aptitude à notifier les changements d'état des services pour la gestion d'évènement de $UPnP^{TM}$. Un point de contrôle désirant recevoir des notifications d'évènement provenant d'une source (équipement), enverra une requête de souscription contenant le service concerné, une interface recevant les évènements et une durée de souscription.

La souscription doit être renouvelée périodiquement pour continuer à recevoir les notifications, et peut être également annulée en utilisant GENA.

SOAP : *Simple Object Access Protocol* définit l'utilisation du XML et du HTTP pour exécuter les appels des procédures distantes. Il est devenu la norme pour la communication basé sur RPC au dessus de l'Internet. En plus d'utiliser l'infrastructure du réseau Internet, il peut aussi fonctionner efficacement avec des pare-feu et des proxies. Il permet aussi d'utiliser Secure Sockets Layer (SSL) pour la sécurité et utilise les outils de gestion de connexion de http rendant de ce fait la communication distribuée aussi facile que l'accès à une page web.

$UPnP^{TM}$ utilise SOAP pour délivrer des messages de contrôles aux équipements, et renvoyer les résultats ou les erreurs aux points de contrôles.

Chaque requête de contrôle $UPnP^{TM}$ est un message SOAP qui contient l'action à invoquer avec un ensemble de paramètres. La réponse est aussi un message SOAP, qui contient le statut, la valeur de retour et renvoie certains paramètres.

Les messages de souscription et de notification d'évènement sont exprimés par le protocole GENA.

UTL, SCPD : Le XML Schema pour la description des *devices* $UPnP^{TM}$ est appelé $UPnP^{TM}$ *Template Language* (UTL).
Service Control Protocol Definition (SCPD) définit un format de description de services.

2.2.2 DPWS

Initialement soumis en mai 2004 par Microsoft, DPWS était destiné à constituer la nouvelle version majeure de l'architecture $UPnP^{TM}$ Device : $UPnP^{TM}$ v2. Néanmoins DPWS s'est vu refuser ce statut ; malgré cela Microsoft vient de l'intégrer dans sa nouvelle version de son système d'exploitation : Windows Vista. Comme dans $UPnP^{TM}$, on distingue 3 entités :

- les équipements ;
- les services ;
- les points de contrôle.

Les équipements(*device/"hosting service"*)

La spécification de DPWS définit une architecture dans lequel des *devices* exécutent deux types de services : des services "d'hébergement" (*hosting services*) et des services hôtes (*hosted services*). Dont le service d'hébergement racine est le *device*.

Les services d'hébergement sont directement associés à un *device* et jouent une part importante dans le protocole de découverte de *device*.

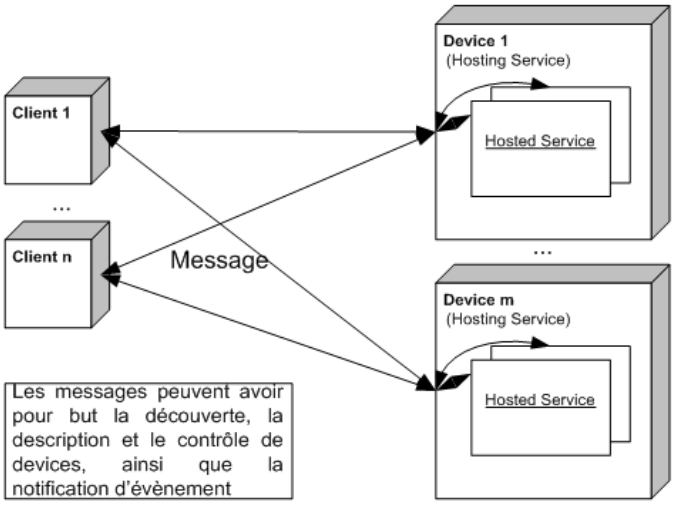

FIG. 2.7 – Exemple d'architecture basée sur DPWS

Dans la figure précédente, la profondeur de la hiérarchie est de 2 ; cependant, il est théoriquement possible qu'un service d'hébergement soit lui même hébergé par un autre service.

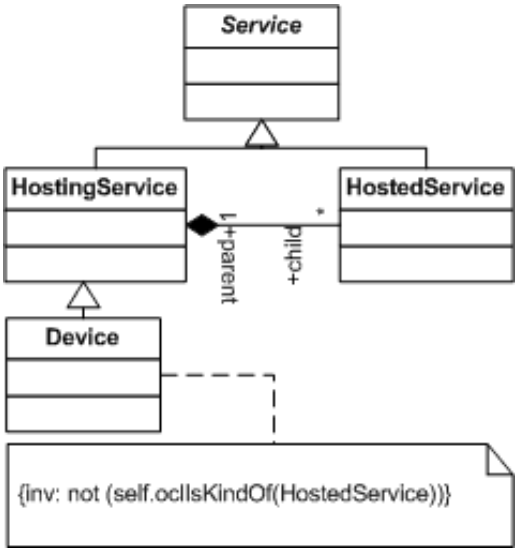

FIG. 2.8 – Représentation d'un *device* DPWS

Les services hôtes sont pour la plupart fonctionnels et dépendent de leur service d'hébergement pour la "découverte".

Les services(*"hosting/hosted" Service*)

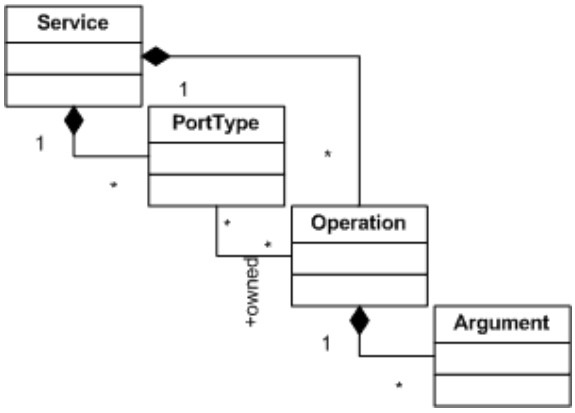

FIG. 2.9 – Représentation d'un service DPWS

Les points de contrôle (*control point*)

Les points de contrôle permettent la découverte des services des *devices* afin d'utiliser les opérations correspondantes.

La pile DPWS

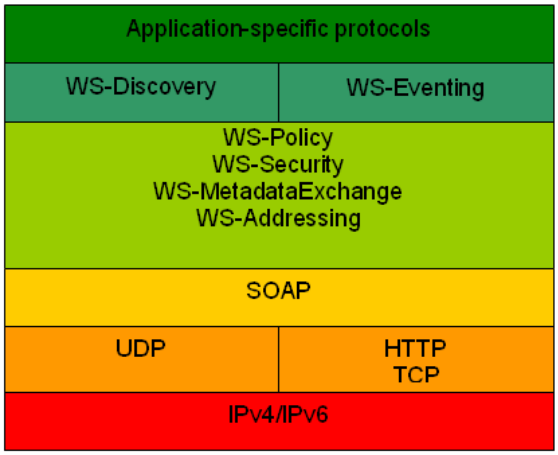

FIG. 2.10 – Pile DPWS

IP : La pile de protocoles DPWS est basée sur le protocole IP aussi bien v4 que v6.

UDP, HTTP, TCP : Tout comme $UPnP^{TM}$, TCP/IP et UDP/IP fournissent la base de la pile de protocole pour fournir la connectivité entre le réseau et les équipements DPWS. HTTP est, là aussi, une partie primordiale de DPWS.

SOAP 1.2 : Si $UPnP^{TM}$ v1 utilisait déjà SOAP pour son protocole de contrôle, DPWS utilise SOAP pour tous les protocoles de haut niveau. L'usage de SOAP est un aspect clé de cette pile de protocole. En effet, en raison des dispositifs d'extensibilité établis dans SOAP, l'architecture des Web Services est hautement composable, ce qui permet d'être intégré individuellement et incrémentalement, sans perturber le reste de la pile de protocole.

Un autre avantage de l'utilisation de SOAP est de pouvoir factoriser des fonctionnalités communes telles que la sécurité pour les mécanismes de contrôle, de découverte et d'évènement.

DPWS utilise la dernière version de SOAP (1.2). Or SOAP 1.2 inclut de nombreuses améliorations par rapport à la version 1.1 : l'élimination des ambiguïtés, un modèle de traitement plus propre, une meilleure interopérabilité, l'indépendance du protocole de transport et un format plus extensible.

WS-Discovery : *Web Services Dynamic Discovery* définit un protocole de découverte en multicast pour localiser des services. Par défaut, des sondes sont envoyées à un groupe en multicast, les services ciblés retourne une réponse directement à l'émetteur (donc en unicast).

WS-Discovery est basé sur SOAP et supporte aussi bien la version 1.1 que la 1.2. Il est construit sur WS-Addressing, WS-Policy et les variantes des standards WS-Security. La spécification de WS-Discovery définit un protocole de découverte multicast pour rechercher et localiser des services disponibles des *devices* connectés au réseau.

Le mode par défaut de la découverte est un client (*control point*) recherchant un ou plusieurs services ciblés. Dans le contexte de DPWS, un service ciblé est un *device*. La recherche peut être plus spécifique, en indiquant le type et la portée du *device* recherché. Le message de sonde est envoyé à un groupe en mode multicast, les *devices* concernés renvoient un message de réponse en mode unicast. Un *device* peut être aussi ciblé par son nom, de manière similaire, à travers un protocole d'échange impliquant un message de résolution (sonde) en mode multicast et le *device* ciblé renvoie une réponse en mode unicast.

Pour réduire au minimum l'envoi de sonde, les *devices* rejoignant le réseau envoient en multicast un message "Hello" pour annoncer leur disponibilité ; de même lorsqu'ils quittent le réseau ils envoient un message "Bye" en multicast pour annoncer leur indisponibilité. En écoutant le groupe multicast, un *control point* peut donc détecter les nouveaux services disponibles sans devoir sonder à nouveau le réseau.

Les messages du protocole WS-Discovery sont envoyés en UDP, afin de réduire au minimum le coût sur le réseau. La découverte en multicast est limitée aux sous-réseaux locaux.

Pour que la découverte puisse passer à l'échelle des grandes entreprises, WS-Discovery introduit la notion de proxy de découverte (Discovery proxy : DP). Il a deux buts : supprimer les messages en multicast afin de réduire le trafic sur le réseau, et étendre le protocole

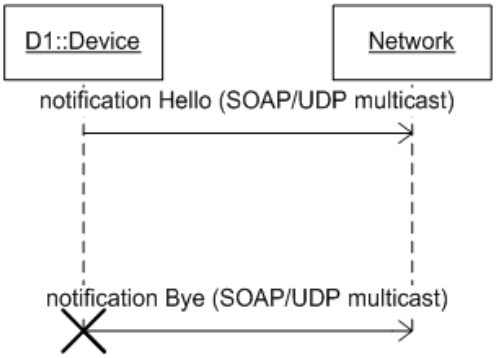

FIG. 2.11 – Message de disponibilité d'un *device* : Cycle

de découverte au-delà du sous-réseau. Quand un proxy de découverte détecte une sonde ou une requête de recherche envoyé en multicast, il renvoie un message "Hello" à l'envoyeur. En écoutant ces annonces le client détecte le proxy de découverte et commute sur le protocole spécifique au DP. Cependant lorsqu'un DP devient "insensible" aux requêtes, les clients repassent au protocole ordinaire de découverte.

WS-Eventing : *Web Services Eventing* a été soumis au W3C en mars 2006.

Il décrit comment construire un modèle d'échange de messages orienté évènement en utilisant des concepts de WS-Addressing, permettant à des Web-Services d'agir comme source d'évènement pour des souscripteurs.

Ce protocole définit les opérations requises pour gérer les souscriptions à des sources d'évènement, aussi bien que la manière dont les messages d'évènement sont construits.

Il décrit un protocole permettant à un client de souscrire auprès d'un Web-Service (appelé source d'évènement) et de recevoir les évènements par un mécanisme de notification provenant de ce dernier. Pour améliorer la robustesse, un ou plusieurs Web-Services peuvent souscrire à une source d'évènement, cette souscription expire avec le temps. Une source d'évènement peut, ou non, autoriser un Web-Service à renouveler sa souscription. Comme WS-Discovery, WS-Eventing est basé sur SOAP et est construit sur WS-Addressing au dessus de SOAP. WS-Eventing n'a besoin de définir aucun nouveau protocole. Il spécifie simplement les définitions WSDL associées aux opérations de gestion des souscriptions : Subscribe, Unsubscribe, Renew et SubscriptionEnd (utilisé par une source d'évènement lorsque celui-ci supprime son service d'évènement). Les messages de notification d'évènement ne sont pas, habituellement, acquittés par les destinataires. Les données contenues dans le message de notification d'évènement peuvent être de n'importe quel type. Une source d'évènement peut filtrer afin de limiter le nombre de notifications envoyée à un Web-Service ayant souscrit à cette source.

WS-MetadataExchange : *Web-Services MetadataExchange* définit des messages pour récupérer des métadonnées d'un type particulier associées à un *device* donnée. Pour cela,

WS- MetadataExchange utilise les informations contenues dans sa politique, son schéma et son fichier WSDL. Elle utilise donc le WS-PolicyFramework ainsi que la spécification WS-Addressing.

WS-Addressing : Le W3C a ratifié en mai 2005 la spécification Core WS-Addressing 1.0

Web-Services Addressing fournit un mécanisme d'adressage des services Web et d'identification des messages indépendamment du transport. Il étend les capacités des services Web en permettant les échanges de messages asynchrones et en permettant l'interaction de plus de deux services à la fois.

WSDL : *Web Service Description Language* est un format XML pour décrire des Web-services. Les opérations et les messages sont décrits de manière abstraite et passent en un protocole de réseau concret et un format de message pour définir un endpoint. WSDL est extensible pour permettre la description de endpoint et leurs messages indépendamment du format de message ou du protocole réseau employés pour communiquer.

2.2.3 Comparaison

	$UPnP^{TM}$	***DPWS***
Addressing	DHCP/AutoIp	
Naming	DNS(Optional)	
Discovery	SSDP	SOAP/HTTP
Description	DCP/SCPD	WSDL
Invocation	SOAP/HTTP	
Eventing	GENA	SOAP/HTTP

2.3 Plateforme domestique de services

FIG. 2.12 – Exemple réseau domestique

2.3.1 $OSGi^{TM}$

Rappelons tout d'abord la notion de middleware : il s'agit de logiciels servant d'intermédiaires entre d'autres logiciels. Il pourra s'agir de logiciels de niveau d'abstraction différente. La couche middleware d'un système peut permettre d'interfacer une application de haut niveau avec un pilote de bas niveau, ou encore assurer la communication entre des éléments de système complexe, éventuellement à travers un réseau. Voyons un exemple dans le domaine des applications distribuées : il est possible d'offrir à une application des fonctionnalités de communication à travers un middleware, en faisant abstraction du protocole utilisé.

La spécification $OSGi^{TM}$ décrit une plateforme de middleware se voulant, d'après l'Alliance $OSGi^{TM}$, "universelle". Cette technologie permet la construction d'applications architecturées selon les principes des SOA et des architectures à composants. Cela nous offre une gestion avancée du cycle de développement de logiciels pour $Java^{TM}$, une application pouvant être construite comme un assemblage de plugins.

Cette plateforme est le fruit des travaux de l'Alliance $OSGi^{TM}$, qui fut fondée en 1999 et rassemble un nombre important de grandes entreprises mondiales.

Par exemple on peut démarrer et exécuter certains bundles dont on a l'utilité, puis les désactiver une fois utilisés afin d'alléger la charge de la plateforme d'exécution. Les implémentations d'$OSGi^{TM}$ offrent donc une souplesse et un fonctionnement dynamique particulièrement adapté pour les systèmes embarqués. Le framework peut évidemment être utilisé pour des systèmes de taille plus importante, ainsi Eclipse s'exécute sur une implémentation d'$OSGi^{TM}$, on peut aussi créer des serveurs web, etc.

On ajoutera qu'une passerelle peut fournir grâce à des bundles l'accès à des services extérieurs au réseau local, par exemple sur internet.

Une dernière fonctionnalité est également la possibilité pour $OSGi^{TM}$ de télécharger de nouvelles versions de bundles et de gérer les dépendances entre ceux-ci, permettant un

entretien régulier du système.

Architecture $OSGi^{TM}$

Au-delà de cette définition du rôle initial de la spécification $OSGi^{TM}$, ce framework permet dans ses diverses implémentations la réalisation de n'importe quel logiciel sous forme de composants.

Comme nous l'avons vu, l'un des points forts de la spécification est la possibilité d'installer, mettre à jour, arrêter, désinstaller des composants logiciels sans interrompre le fonctionnement d'autres composants sur la plateforme : ils peuvent en effet être notifiés du changement d'état des services ou packages dont ils dépendent afin de réagir en correspondance sans être interrompus.

De nombreux fonctionnalités ont déjà été définies et implémentées, par exemple : un bundle HTTP, des bundles d'administration, un bundle d'analyse de documents XML, ou encore un bundle de communication $UPnP^{TM}$ qui retiendra notre attention en particulier.

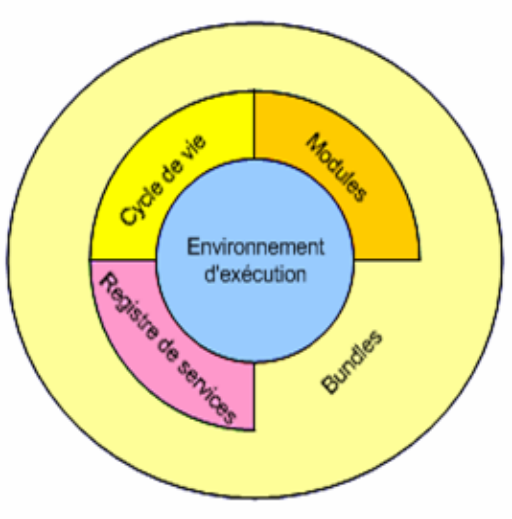

FIG. 2.13 – Vue en "couches" de la structure d'$OSGi^{TM}$

L'environnement d'exécution consiste en une machine virtuelle classique JSE/EE, ou $Java^{TM}$ ME/CDC (contenant le sous-ensemble de Foundation Profile OMEE, $OSGi^{TM}$ Minimum Execution Environment) pour pouvoir exécuter un framework $OSGi^{TM}$ sur un système embarqué, une passerelle ou un téléphone mobile par exemple.

La couche bundles correspond à la partie applicative du framework, s'exécutant "par-dessus" les autres. Un bundle contient nécessairement un fichier manifest décrivant ses imports et exports de packages, ainsi que la version du bundle.

L'accès à un package permet de créer des objets d'un type particulier avec les possibilités que cela comporte (l'application peut utiliser les classes du package). Entre les 2 couches que nous avons vues (bundles, environnement d'exécution) nous allons voir qu'il existe

d'autres couches offrant des fonctionnalités avancées : les couches Modules, et Cycle de vie.

La couche Modules définit les politiques de chargement des classes, permettant le cloisonnement des bundles et limiter la vue extérieure de chaque application. A chaque bundle du framework correspondra un classloader différent : Contrairement à l'exécution d'applications $Java^{TM}$ standard utilisant un unique classpath leur laissant l'accès à toutes les autres classes publiques chargées, un bundle peut choisir lesquelles de ses classes resteront privées, et lesquelles il va partager. Il ne sera donc possible d'accéder aux classes d'un autre bundle que par le biais des packages exportés par celui-ci.

Voyons enfin la couche Cycle de vie, pour affiner notre compréhension des mécanismes internes au framework. Un bundle peut prendre divers états au cours de son existence :

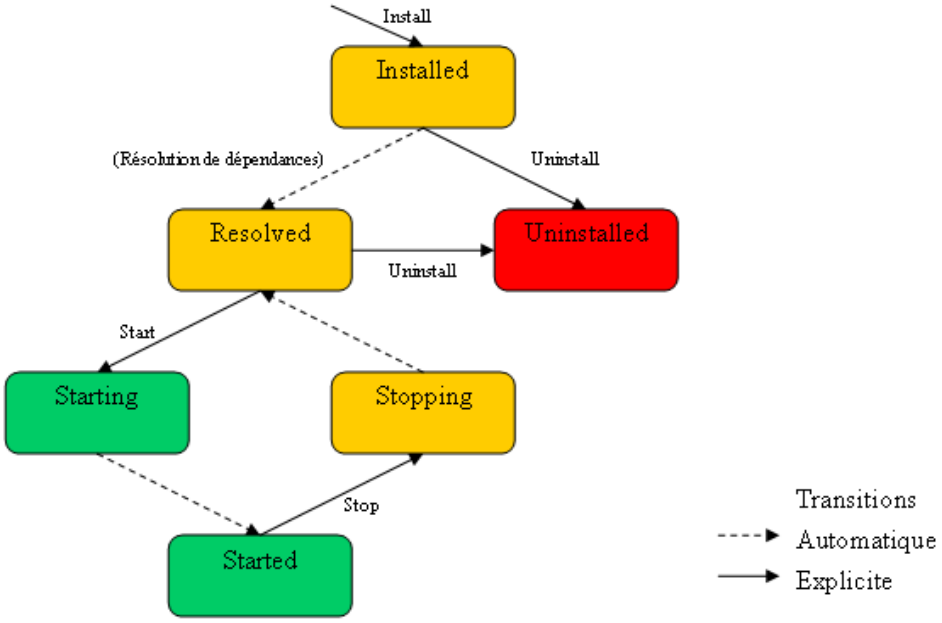

FIG. 2.14 – Cycle de vie d'un bundle

Voyons brièvement à quoi correspondent les états d'un bundle :

Installed : Le bundle est installé dans le framework mais ses classes ne peuvent être utilisées, et ses dépendances avec d'autres bundles dont il utilise des packages ou services ne sont pas résolues.

Resolved : Les dépendances de packages et services ont été résolues, on pourra démarrer l'exécution du bundle sans contre-indication. Si ce bundle exporte des packages, un autre bundle pourra instancier des objets de type contenus dans le bundle et exécuter les méthodes y étant associées. Cependant cela est une propriété d'un bundle "inerte". Pour utiliser un service, il faudra que le bundle ait été démarré (grâce à une classe spéciale

Activator).

Starting : Une fois le package résolu, le bundle peut être lancé (nécessite un appel explicite de lancement). Au démarrage, le bundle instancie les composants des bundles pouvant requérir ou fournir des services.

Started : Le composant a bien été créé et peut exécuter les demandes qui lui sont transmises ou en émettre vers d'autres services, utiliser d'autres packages ou d'autres services.

Stopping : Le composant commence à s'arrêter ; dans le cas d'un service, il demande au framework que celui-ci le retire du registre de services. Cette étape effectuée, le bundle retourne à l'état *Resolved*.

Uninstalled : Le bundle est complètement supprimé du framework.

Il faut noter que tout bundle peut utiliser des listeners pour être notifié de changements d'états d'autres bundles ou services (éventuellement en spécifiant un filtre, ou par type). Ainsi les composants d'un bundle pourront être démarrés ou arrêtés suivants l'état de leurs dépendances, ce qui permet robustesse et dynamicité de l'application).

On notera de plus que les bundles peuvent communiquer directement avec l'environnement d'exécution $Java^{TM}$, sans appel au framework. On pourra donc accéder directement à l'API $Java^{TM}$ sans Overhead lié au framework. Celui-ci n'est donc pas pénalisant au niveau performances. Il nous paraît important de spécifier cela, ceci n'étant pas intuitif dans la vue que nous verrons ultérieurement, très couramment utilisée.

Orientation Services au sein d'$OSGi^{TM}$

On peut voir les bundles ainsi :

La différence entre package et service peut s'expliquer assez simplement. Un package peut fournir des classes que l'on pourra instancier sous forme d'objets, et peut donc être utilisé alors que le bundle qui le fournit est simplement à l'état Resolved. Un bundle exportant un package peut être "inerte", sans Activator. A l'opposé, un service est dynamique et doit nécessairement être en exécution.

Dans son implémentation, un service d'un bundle est réalisé sous forme d'un objet $Java^{TM}$ classique (POJO), et accédé directement par les composants l'utilisant : la plateforme $OSGi^{TM}$ n'intercepte pas les appels.

Le framework $OSGi^{TM}$ offre un Registre de services, permettant l'enregistrement et la découverte de ceux-ci. Cette architecture est fidèle aux concepts de SOA. Un service est spécifié par son interface, et cette interface se doit d'être partagée tel un contrat entre le fournisseur et les utilisateurs du service. Un bundle délègue une tache à un service grâce aux méthodes définies dans son interface.

Plus concrètement :

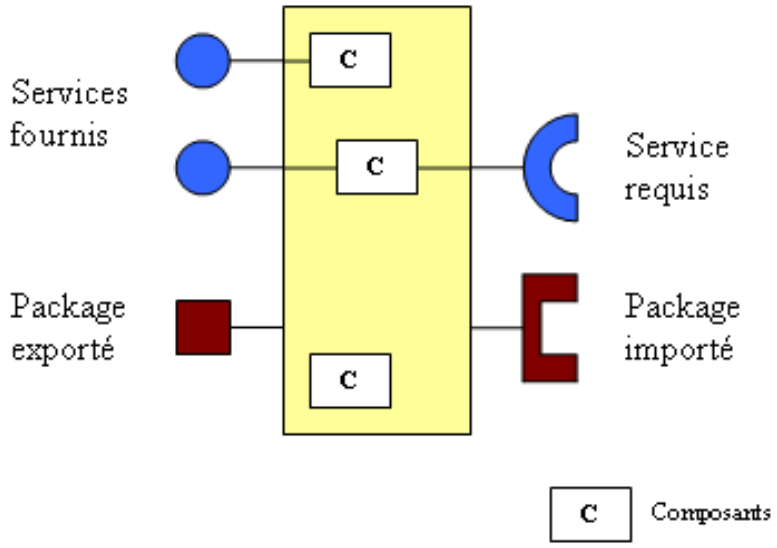

FIG. 2.15 – Exemple de bundle, requérant et fournissant services et packages, avec ses composants

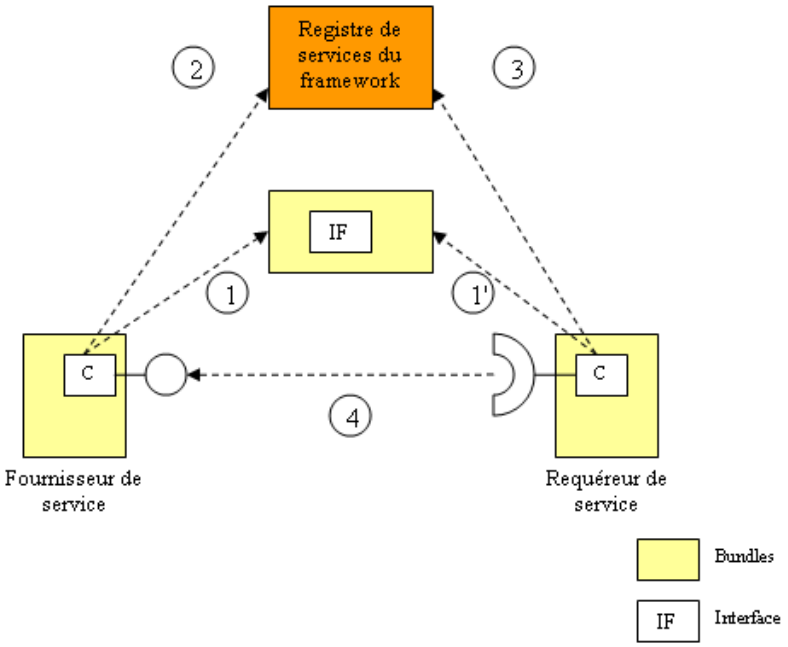

FIG. 2.16 – Utilisation idéale des services dans $OSGi^{TM}$

- En 1 et 1', le fournisseur et le demandeur de services accèdent à une interface $Java^{TM}$ contenue par un bundle tiers. Cette interface va être utilisée tel un contrat pour spécifier les méthodes et objets que doit fournir le service, et auxquels a accès le requéreur.
- En 2, le fournisseur de service créé une référence vers l'objet de son composant implémentant le service. Cette référence est évidemment créée à partir de l'interface obtenue lors de l'étape 1. La référence est alors publiée dans le registre de services, et donc accessible de l'extérieur.
- En 3, le requéreur de services fait appel au registre de services pour connaître les services disponibles. Il est possible de spécifier un filtre lors de la demande, en spécifiant l'interface obtenue en 1', pour un résultat plus ciblé. Le registre de services retourne la liste des services correspondant à la demande.
- En 4, le requéreur de service a trouvé le service désiré correspondant à l'interface obtenue en 1' dans la liste, et peut utiliser le service, appeler ses méthodes, etc.

2.4 Device Access

L'Alliance $OSGi^{TM}$ définit, dans la partie *Device Access specification* (cf.[14.]), un ensemble de règles pour les "*devices*"

2.4.1 Fondamentaux

Si la spécification $OSGi^{TM}$ précise les moyens de communication entre bundles, elle couvre également la façon de gérer des équipements (*devices*). Cela se fait par le biais de bundles pilotes (drivers). Il est imposé que cette gestion soit le plus générique possible pour une classe de *device* donné, et non spécifique à certains matériels. Les spécifications externes des pilotes ne dépendent pas des protocoles exploité à bas niveau afin qu'il soit le plus générique possible .

Une architecture constituée de *devices* et Drivers est proposée. Celle-ci contient principalement des interfaces *Driver Locator*, *Driver Selector* ainsi que *Match*, ainsi qu'une classe *Driver Manager* pour les cas où pour un *device* donné, l'on ait plusieurs drivers disponibles : ils sont alors en compétition. Le Driver Manager se chargera alors de trouver (et éventuellement télécharger et installer) le driver le plus approprié au *device* (Match), selon les propriétés du *device* et des drivers disponibles.

Plusieurs types de drivers existent, on citera notamment :

- **Les Base Drivers** : Ce sont les implémentations minimales de drivers, représentant le *device* au "plus bas niveau". Ces drivers n'entrent pas en compétition avec d'autres drivers car ils se doivent d'être génériques ; ce sont des briques de base que l'on ne peut réduire. Ils peuvent être réutilisés par des drivers de plus haut niveau.
  Il existe 3 distinctions parmi les base drivers, à savoir : les Base Drivers simples pour les *devices* directement connectés à la machine physique exécutant le framework, et ne supportant pas la recherche dynamique (port parallèle, série, ...) ; les Discovery Base Drivers pour les matériels directement connectés à la machine physique supportant

la découverte dynamique (USB, IEEE 1394, ...) ; les *Pure Discovery Base Drivers* pour la détection de *devices* distants utilisant des technologies de type Plug and Play ($UPnP^{TM}$, Bonjour, DPWS, ...).

- **Les drivers "d'affinage"** : Ils permettent différents niveaux d'abstraction sur les *devices* ; par exemple : un Base Driver USB peut être "affiné" avec un driver "souris USB", plus spécifique.
- **Les drivers "de pontage"** : Prenons par exemple le cas d'un *device* adaptateur USB pour réseau Ethernet. Il est dans ce cas intéressant de fournir une interface exclusivement Ethernet pour les bundles du framework, permettant de masquer le passage en USB. Ainsi un bundle voulant communiquer sur le réseau Ethernet communiquera les données qu'il veut émettre au bundle driver, qui convertira le tout implicitement pour USB. Le périphérique USB matériel assure ensuite la continuité du port USB physique de la machine vers le réseau physique Ethernet. La communication est donc transparente.

Il existe encore d'autres types de drivers mais nous n'avons sélectionné que les plus intéressants dans notre cas. Chaque driver n'étant pas un Base Driver doit s'inscrire dans le Registre de services pour pouvoir "entrer en compétition" avec les autres et être utilisé.

La spécification indique également que les *devices* gérés par des bundles du framework doivent être enregistrés dans le Registre de services avec au minimum la catégorie de *devices* à laquelle ils appartiennent. Cela permet aux pilotes ayant enregistré un listener d'être notifiés des arrivées ou départs des *devices* : c'est une approche de souscription de service de type publish/subscribe auprès du Registre de services. Il est également possible aux pilotes ayant détecté des *devices* de les enregistrer dans le registre de services afin que d'autres bundles puissent les utiliser.

2.4.2 $UPnP^{TM}$ Base Driver

Nous avons vu ce qu'est un Base Driver, ainsi que la technologie $UPnP^{TM}$. Il est donc assez aisé à ce stade du document de comprendre l'utilité du Base Driver $UPnP^{TM}$: celui-ci offre des fonctionnalités génériques pour l'utilisation du protocole $UPnP^{TM}$ au sein des plateformes OSGi. Tout comme sur le schéma précédent où le driver permettait la communication avec un périphérique USB, le driver $UPnP^{TM}$ permet la communication sur le réseau.

Il offre aussi bien des fonctionnalités pour écrire des points de contrôle $UPnP^{TM}$ que pour exposer des objets comme des *devices* $UPnP^{TM}$ sur le réseau. Les objets $Java^{TM}$ souhaitant se comporter ainsi se doivent d'implémenter les interfaces UPnPDevice ou UPnPService.

Voici les principales fonctionnalités du pilote pour les bundles points de contrôle développés sous le framework $OSGi^{TM}$:

- il permet la détection transparente des *devices*/services $UPnP^{TM}$ distants ;
- réifie les *devices* $UPnP^{TM}$ présents sur le réseau sous forme d'objets $Java^{TM}$ enregistrés auprès du registre $OSGi^{TM}$. Les points de contrôle peuvent découvrir ces objets via le registre $OSGi^{TM}$ et les utiliser pour interagir avec les *devices* $UPnP^{TM}$;

- il permet l'accès à la description des *devices* détectés : variables d'états, actions, URL de présentation ;
- permet l'émission d'actions vers les *devices*/services ;
- la souscription/désinscription aux évènements des *devices*/services locaux ou distants ;
- la réception des évènements auquel le point de contrôle a souscrit, à travers le réseau ;
- la signalisation de ces évènements aux bundles points de contrôle ayant contracté une souscription à un service, par le biais interne d'un listener.

Pour toute souscription à un évènement d'un service ou *device*, le service ou *device* est seul responsable de la périodicité de ses émissions d'évènements

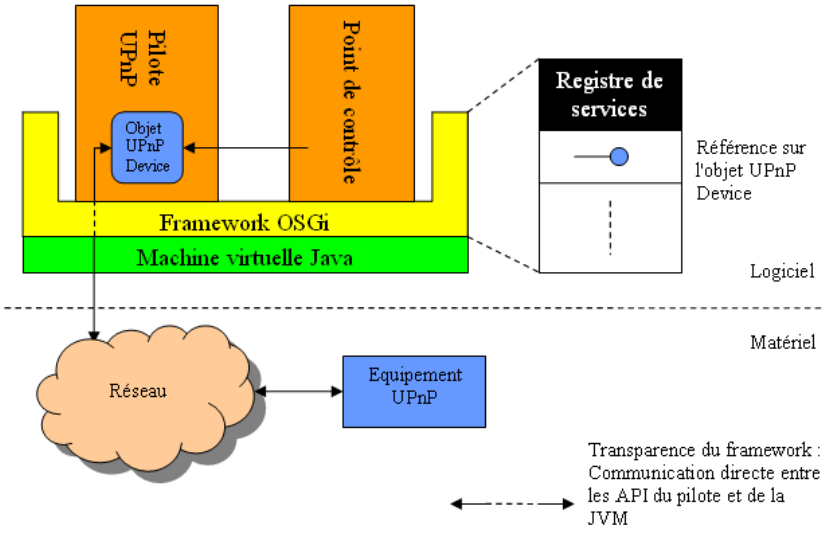

FIG. 2.17 – $UPnP^{TM}$ base driver

Comme nous l'avons vu, le pilote crée pour chaque *device* distant un objet local de type UPnPDevice le représentant. Le pilote enregistre une référence sur cet objet dans le registre de services, ce qui permet sa découverte par le point de contrôle. Cet objet est construit en fonction de la description du *device*. Il va donc permettre d'accéder à toutes les fonctionnalités du *device*, par simple appels de méthodes $Java^{TM}$, localement. L'objet transmet les demandes au pilote, qui transmet ensuite les demandes à la JVM de manière transparente, pour qu'elle les transmette enfin sur le réseau. La communication est donc masquée pour le point de contrôle qui ne voit qu'une référence $Java^{TM}$. Il est à noter que le *device* $UPnP^{TM}$ a évidemment sa propre couche logicielle interne.

Voyons maintenant les principales fonctionnalités du pilote pour les bundles *devices* (objets $Java^{TM}$) également développés sous $OSGi^{TM}$:

- lors de leur inscription auprès du framework, le pilote les détecte et leur associe un identifiant ;
- il gère l'envoi de messages "Hello" signalant les *devices* sur le réseau lors de cette

détection ;
- lors de la réception d'un message de recherche émis par un point de contrôle sur le réseau, il répond en émettant la description des *devices* qu'il a détecté dans le framework. Cette description contient notamment leur URL de présentation sous forme de page Web ;
- il gère automatiquement l'envoi périodique de messages "Alive" signalant que le *device*/service est toujours présent ;
- gère la communication d'informations aux points de contrôle ayant souscrit à un ou des évènements du *device*/service ;
- transmet les appels d'actions émis par les points de contrôle au *device*/service ;
- gère l'envoi de message de départ d'un *device* du réseau ("Bye").

La gestion de la page de présentation et son accès sont laissés complètement libres au développeur, et sont indépendants du Base Driver. Celui-ci ne publie à ce sujet que la propriété d'adresse de la page dans la description du *device* qu'il communique, à partir d'informations que lui a précisé le *device*.

Les *devices* et services distants sont enregistrés auprès du registre du framework avec la même interface $Java^{TM}$ que les locaux s'y enregistrant directement. Lorsqu'un point de contrôle cherche des *devices* et/ou services auprès du registre, il trouvera donc aussi bien les locaux que les distants, ce qui lui permet de piloter également les locaux. Cependant, les locaux précisent une propriété UPNP_EXPORT lors de leur inscription, offrant 2 possibilités : premièrement le filtrage de la recherche pour les points de contrôle ; deuxièmement que le Base Driver les détecte, sans redétecter les *devices* distants qu'il a enregistrés comme étant locaux et à publier sur le réseau (boucle).

Chapitre 3

SmartPnP project : *DPWS* base driver

3.1 Cas d'utilisation

DPWS définit deux types d'équipements :

- les *control points*
  Nous pouvons distinguer trois phases dans l'interaction entre un *control point* et un *device* : la découverte (cf. 1. de la fig. 3.1), la collecte d'information (cf. 2. et 3. de la fig. 3.1) et l'invocation d'opération (cf. 4. de la fig. 3.1).
  Outre l'invocation d'opération décrite dans le WSDL, le *control point* peut faire une demande de soucription auprès des services d'un *device*.
- les *devices*
  De la même manière, nous pouvons distinguer deux mécanismes actifs d'un *device* : l'annonce de sa disponibilité sur le réseau et l'émission d'évènement.

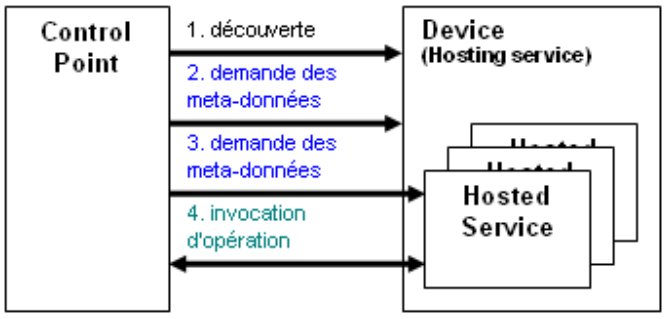

FIG. 3.1 – Interactions *device* / *control point*

Du point de vue d'un base driver nous obtenons 4 entités distinctes logiquement : *control points* et *devices* présents sur le réseau, ainsi que les *control points* (*local control points*) et *devices* (*Pseudo-devices*) logiques présents sur le framework. Nous obtenons donc 4 cas d'utilisations, soient les couples suivants :

- *control point*/ *device* ;
- *control point*/ *Pseudo-device* ;
- *local control point*/ *device* ;
- *local control point*/ *Pseudo-device*.

Parmi ces cas d'utilisation, nous pouvons différencier deux types de *control points* : spécifique ou générique. En effet, un *control point* est généralement implémenté pour un *device* spécifique et ne peut agir que sur ce type de *device* ; néanmoins, un *control point* générique offre l'avantage de pouvoir agir sur l'ensemble des *devices*, mais cela ce fait généralement au détriment de l'ergonomie et de l'intuitivité.

3.2 Problèmatique

Nous distinguons plusieurs mécanismes nécessaires dans l'implémentation d'un *DPWS* base driver :

i. mécanisme de découverte de *device* ;
ii. mécanisme de réification des devices sur le framework ;
iii. mécanisme d'exposition de *Pseudo-device* ;
iv. mécanisme d'invocation d'action sur un *device* ;
v. mécanisme de répartition des actions provenant du réseau vers les *Pseudo-devices* correspondants ;
vi. mécanisme de gestion de souscription à des sources d'évènement ;
vii. mécanisme d'émission d'évènement ;
viii. mécanisme de répartition des évènements provenant du réseau vers les *local control points* souscripteurs.

Ce projet s'inscrit dans un domaine domestique, ce qui implique implicitement que l'application puise être embarquée sur des supports aux ressources limitées (IGD, téléphone portable, pda...). La spécification *DPWS* souligne que :

> "In highly-constrained circumstances a [DPWS] client will know all it needs to know about a device and its hosted services to correctly send and receive application-specific messages"

C'est-à-dire : dans des "circonstances" hautement contraignantes, un client saura tous ce qu'il a besoin de savoir à propos d'un *device* et de ses *hosted services* pour envoyer et recevoir correctement des messages d'une application spécifique.

Pour récupérer la description d'un *device* et de ses *hosted services*, le mécanisme correspondant ne devrait pas utiliser la bande passante ainsi que la ressource mémoire inutilement.

Le problème lié à la contrainte "embarquée" se répercute sur la couche middleware. En effet celui-ci ajoute un coûtcout) de traitement supplémentaire qui se traduit par une augmentation des besoins aussi bien en ressource mémoire qu'en ressource de calcul.

3.3 Solutions

Selon la catégorisation de Feng Zhu, Matt W. Mutka et Lionel M. Ni (cf. [2.]), nous avons établit les points de la figure 3.2

	DPWS
Service and attribute naming	Template-based and predefined
Initial communication method	Unicast and multicast
Discovery and registration	Query and announcement-based
Service dicovery and infrastructure	Nondirectory-based
Service information state	Hard state
Discovery scope	Network topology (LAN)
Service selection	Manual
Service invocation	Service location, comunication mechanism (XML, SOAP, and HTTP), and application operations
Service usage	Explicitly released
Service status inquiry	Notification and polling
Security dependency	WS-Security

FIG. 3.2 – Catégorisation (cf. [2.]) de *DPWS*

Les sous-sections suivantes définissent les mécanismes et concepts à implémenter.

3.3.1 Lazy/Immediate Discovery

Problématique : i.

Une application réseau devrait en théorie s'adapter ou se configurer en fonction de l'environnement et le trafic d'un réseau.

En effet, sur un vaste réseau (e.g. un réseau d'entreprise), un ensemble conséquent de *device* peut être présent. L'envoie d'une sonde engendrera une réponse de tous les *devices* et par conséquent perturbera potentiellement le réseau. Deux modes sont donc disponibles et sélectionnables pour le base driver :

- **la découverte immédiate :**
  Lors du démarrage du base driver celui-ci émet une sonde sur le réseau afin de découvrir les *devices* disponibles.

– **la découverte dite paresseuse (*lazy*) :**
Contrairement à la découverte immédiate, la base driver n'envoie aucune sonde sur le réseau et découvre seulement les *devices* s'annonçant ultérieurement à son lancement.

3.3.2 Lazy/Immediate Loading

Problématiques : i, ii.

Comme dit précédemment, une application réseau devrait en théorie s'adapter ou se configurer en fonction de l'environnement et le trafic d'un réseau. De plus, une application pouvant être déployé sur différents supports devrait en théorie, s'adapter ou se configurer en fonction des ressources et des besoins.

Sur un vaste réseau (e.g. un réseau d'entreprise), un ensemble conséquent de *device* peut être présent. Un utilisateur ne voulant pas utiliser l'ensemble des *devices*, devrait pouvoir découvrir un ensemble minimal d'information sur les *devices* du réseau ; il est en effet inutile de recevoir la totalité des informations sur des services inutilisés.

Deux modes de chargement/réification sont donc disponibles sur le base driver :

– **le chargement immédiat :**
Il est destiné à un réseau supportant le nombre de *device* présent (haut débit, peu de *device* présent, mémoire suffisante pour le base driver, ...). Dans ce cas, le base driver réifie les *devices* avec l'ensemble de leurs informations. L'utilisateur a donc accès à toutes les informations dés la découverte du *device*.
– **le chargement dit paresseux (*lazy*) :**
Il permet de réduire le trafic sur le réseau. Dans ce but, le base driver ne doit réifier qu'un ensemble minimal de métadonnée sur le *device* et chargé le reste de ses méta-données si l'utilisateur accède à celui-ci. De cette manière nous optimisons le trafic du réseau ainsi que la gestion de mémoire du base driver. L'inconvénient, c'est que ce mode induit un temps de latence réseau entre la demande d'accès aux informations du *device* et son accès effectif.

3.3.3 Lazy/Immediate Networking

Problématiques : i, ii.

Derrière le *Lazy/Immediate Loading* se cache un autre concept : le *Lazy/Immediate Networking*. On a vu que la réification se faisait à partir de données obtenues auprès du *device*. Or dans le cas du *Lazy Loading*, il n'est pas nécessaire d'obtenir l'ensemble des méta-données tant que l'utilisateur ne désire pas l'utiliser. Deux cas de figure se présentent :

– **Immediate Networking** Ce cas permet à l'utilisateur de pouvoir réifier un *device* immédiatement sans les coups réseaux nécessaires à la demande et à la réception des méta-données du *device* et de ses services. En effet dès la découverte, il est possible d'effectuer toutes ces demandes afin que les informations nécessaires à la réification soient déjà dans un cache (données brutes, non traitées).
– **Lazy Networking** Contrairement au précédent cas, les demandes de méta-données du *device* et de ses services sont invoquées seulement au moment où l'utilisateur désir y accéder. Ce cas à un intérêt dans le cas où le framework dispose d'une ressource

mémoire limitée ; mais engendrant un temps de latence réseau, pouvant être non négligeable, au moment d'accéder aux informations du *device* ou de ses services.

3.3.4 Import et export de *device*

Problématiques : ii, iii.

La représentation d'un *device* réifié est la même que celle d'un device exporté. L'API définit le model conceptuel d'un *DPWS Device*. Un utilisateur peut simplifier la création de *device* local au framework soit en utilisant une fabrique dynamique de *DPWS Services* ou en utilisant un outil de génération de code en statique.

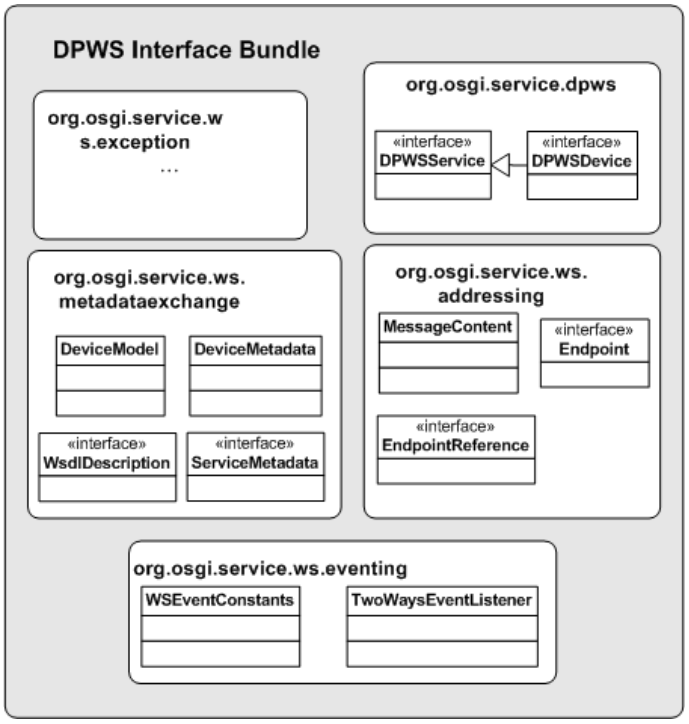

FIG. 3.3 – Interface DPWS

3.3.5 *White Board* pattern

Problématiques : vi, viii.

Le DPWS base driver applique le *White Board* pattern afin de souscrire à une source d'évènement. Le principe est qu'un point de contrôle local dépose dans le registre de service un *handler* avec des propriétés *matchant* un ou des services. Le base driver découvre le *handler* et récupère les propriétés ; il cible le ou les services distant concernés (via les réifications contenues dans le registre de service), et souscrit à ces derniers. Notons que le base driver se substitue aux points de contrôle distant et par conséquent se comporte

comme tel. Les souscriptions sur une source d'évènement locale sont pris directement en charge par le *device* concerné.

Les souscriptions peuvent être mutualisées pour un ensemble de contrôle point ciblant la même source d'évènement. Le base driver prend alors en charge les notifications et les redirige vers les *handlers* correspondants.

3.3.6 Architecture

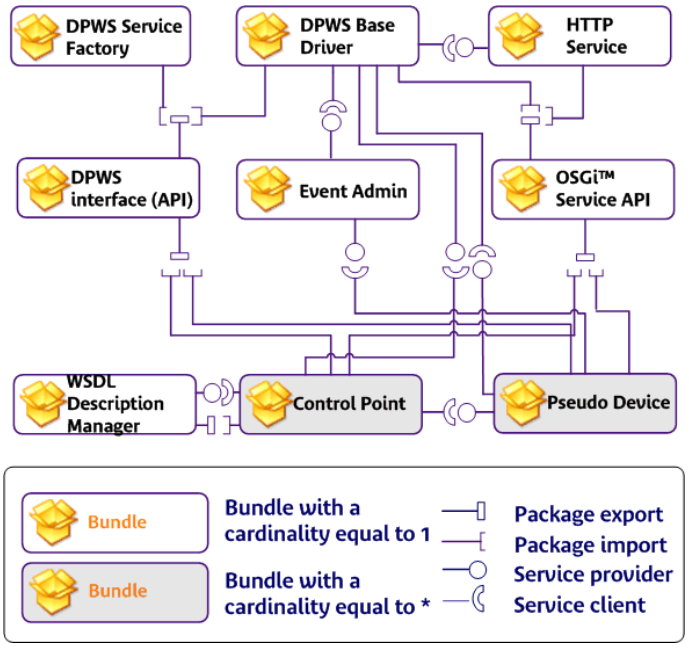

FIG. 3.4 – Architecture globale du système

Les critères ayant influé sur l'architecture sont la modularité, la réutilisabilité et la maintenabilité.

En effet, le base driver jouant le rôle uniquement de *control point* ne requière pas les même composants que s'il joue celui d'un serveur de *device*. Le but était donc de limiter les besoins au contexte prévu par l'utilisateur dans un but de pouvoir passer sur des framework embarqués (cf. [3.2]).

Nous avons conçut cette architecture sur les points suivants :

- séparation de l'API et de l'implémentation ; en effet le base driver se comporte comme un proxy entre le réseau et le framework en se substituant aux *control points*, aux *devices* et aux *pseudo-devices*. Par conséquent, il implémente l'API pour représenter ces derniers.
- réutilisation des standards $OSGi^{TM}$ (HTTP service, Event Admin, etc).
- séparation de la représentation objet du WSDL du base driver : *WSDL manager*. En

effet, elle peut être spécifique aux besoins de l'utilisateur et n'est pas nécessaire dans tous les contextes.

– séparation de la fabrique de service *DPWS* : *DPWS Service Factory*. De même que pour le *WSDL manager*, la fabrique peut être propre aux besoins de l'utilisateur et ne doit pas être par conséquent intégrée au sein du base driver. La fabrique n'est pas nécessaire dans tous les cas d'utilisation.

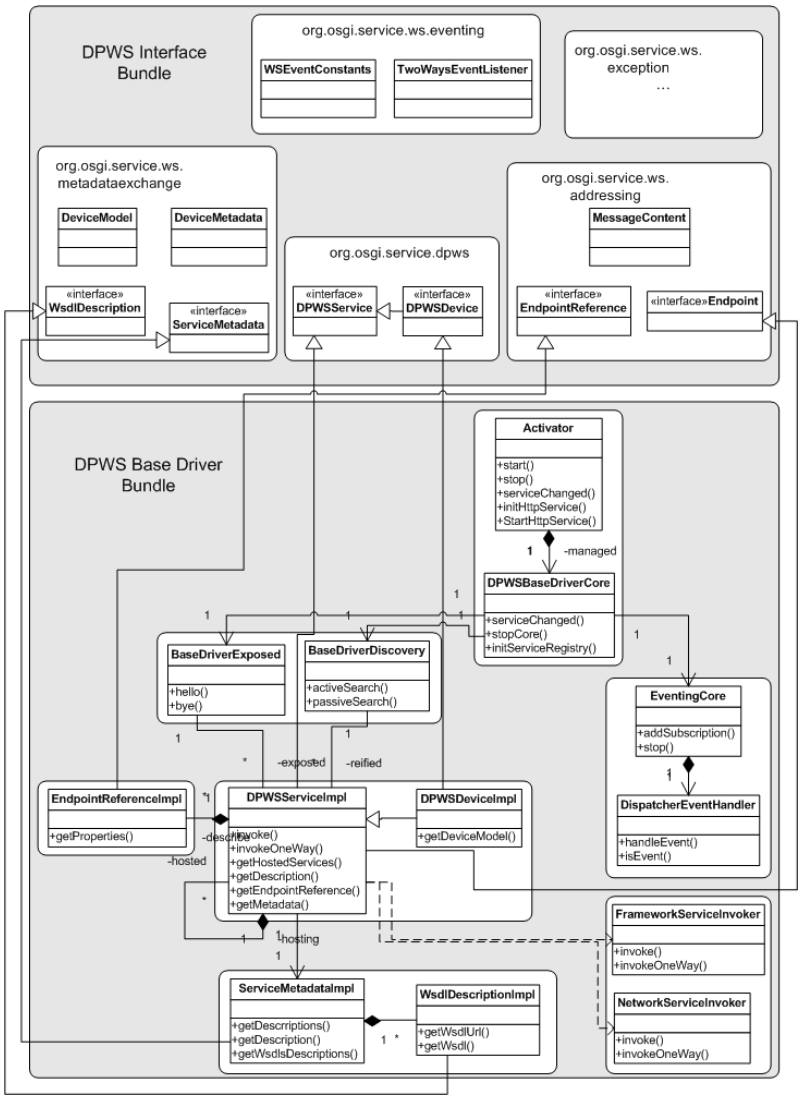

FIG. 3.5 – Architecture du base driver et de l'API

Le base driver implémente l'API que nous avions définit pour modéliser les *devices DPWS*. Le packaging de l'API repose sur l'appartenance des concepts à une spécification ; c'est à dire que les entités décrites par une spécification sont regroupé au sein d'un même package et dont le nom identifie la spécification correspondante. Le base driver reprendre donc logiquement ces "assemblages".

3.4 Tests d'efficacité

3.4.1 Description des conditions de test

Le scénario des tests consiste en une série de 1000 invocations d'action sur le service d'un *device*. Chaque scénario a été effectué par un unique client appelant un *device* : dans un premier temps un *device* embarquant la pile $Java^{TM}$ *DPWS*, puis sur un service *device* $OSGi^{TM}$.

Le client et les *device* sont installés sur deux machines distinctes : les *devices* s'exécutant sur la même machine et le *control point* sur la deuxième. Le réseau est composé de ces deux machines. Les deux *devices* offrent les mêmes fonctionnalités. La prise de temps commence à partir de la première invocation et se finit à la dernière.

- **test1 :** le client souscrit à une source d'évènement ; et invoque 1000 fois une action générant un évènement.
  Les invocations et la réception des évènements sont synchronisées.
- **test2 :** le client invoque 1000 fois une action *request-response*.
- **test3 :** le client invoque 1000 fois une action *one-way*.

Dans chaque scénario une moyenne est calculée : chaque test est effectué 10 fois dont les résultats des deux plus lents et des deux plus rapides ne sont pas utilisés.

3.4.2 Résultats

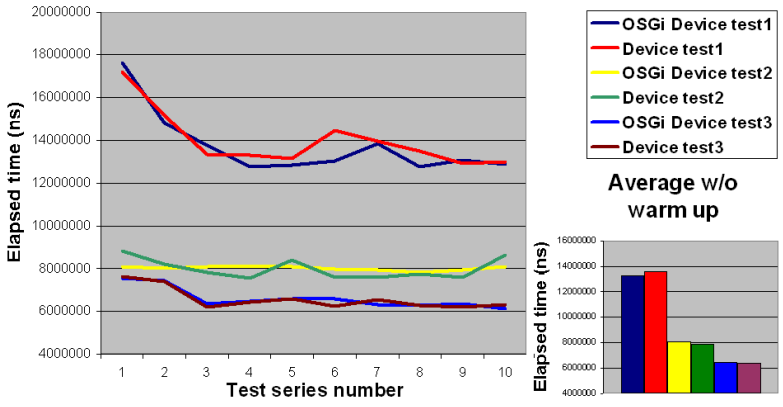

FIG. 3.6 – Test d'efficacité

3.4.3 Analyse des résultats

Dans le pire des cas, l'accès d'un *device* $OSGi^{TM}$ est plus lent de 1,88% que l'accès direct sur le *device* originel. Soient les temps 8 036 849ns contre 7 887 711ns.

Nous avons effectué les mêmes tests mais avec le client porté sur le framework $OSGi^{TM}$; les résultats montrent que dans le pire des cas, l'invocation du *control point* porté sur $OSGi^{TM}$ est 1,606% plus lent que le client originel.

Le coût de la couche middleware est raisonnable comparativement aux avantages obtenus par l'architecture du driver orienté service. Les bénéfices sont la simplicité du développement des applications *DPWS* et la flexibilité de l'architecture séparant la logique métier de l'utilisation d'un protocole donné. Les développeurs sont débarrassés des aspects réseaux et protocolaires.

Chapitre 4

Conclusion et Perspective

4.1 Contribution

Ce projet s'inscrit au sein d'un projet européen : ITEA ANSO. La technologie *DPWS* vient d'être intégrée au système d'exploitation *Windows Vista*, mais ne présente actuellement qu'un seul produit disponible sur le marché.

La spécification définit par Microsoft© pour *DPWS* rassemble un ensemble de spécifications de Web-Service. Une des premières étapes fut d'établir une API en partenariat avec Schneider Electric à partir des différentes spécifications dans le cadre d'une normalisation, afin d'obtenir une future interopérabilité entre différentes implémentations d'un base driver *DPWS*. Cette API a donnée lieu à la rédaction d'une RFC et d'une RFP. La RFP (cf. [7.]) a été accéptée , et la RFC est actuellement en cours de soumission auprès de l'Alliance $OSGi^{TM}$.

L'implémentation du base driver *DPWS* a mis en évidence la permissivité excessive de la spécification de *DPWS*.

Le projet a permis de mettre en place des mécanismes permettant une généricité à partir d'une pile (fournit par Schneider-Electric) basée sur des couples : *device*/ *control point*, appareillés.

Ces différents points ont été exposés dans [6.] soumis en septembre 2007.

4.2 Perspective

Le nombre d'équipements IP au sein du réseau domestique croît de manière exponentiel, l'un des enjeux du futur sera de les intégrer et de les interconnecter au sein du réseau. Dans les prochaines années, les industriels ne convergeront pas nécessairement vers le protocole *DPWS*, néanmoins celui-ci expose des concepts et des buts intéressants qui seront certainement repris par ses successeurs.

D'un point de vue développement, il serait intéressant de concevoir une application permettant de passer d'un protocole *plug and play* vers un autre. Une telle application utiliserait un modèle de *device* servant de pivot entre les protocoles. L'application permettant

ainsi d'interconnecter les composants sans à avoir à implanter l'ensemble des protocoles sur chacun d'entre eux. L'idéal serait néanmoins de converger vers un unique protocole.

Chapitre 5

Référence / Bibliographie

5.1 Bibliographie

[1.] « **The Computer for the Twenty-First Century** »
Auteurs : Mark Weiser
Date : Septembre 1991

Scientific American

[2.] « **Service Discovery in Pervasive Computing Environments** »
Auteurs : Feng Zhu[1], Matt W. Mutka[1] et Lionel M. Ni
Conférence : IEE CS et IEEE ComSoc
Date : 2005

[1]Michigan State University et [2]Hong Kong University of Science and Technology

[3.] « **Service-oriented architectures for devices using $UPnP^{TM}$ and *DPWS*** »
Auteurs : Harm Smit

Schneider Electric

[4.] « **Service-Oriented Architectures for Embedded Systems Using Devices Profile for Web Services** »
Auteurs : Elmar Zeeb, Andreas Bobek, Hendrik Bohn and Frank Golatowski
Conférence : 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)

Institute of Applied Microelectronics and Computer Engineering, University of Rostock

[5.] « **Pervasive Service Composition in the Home Network** »
Auteurs : André Bottaro[1,2], Anne Gérodolle[1], Philippe Lalanda[2]
Date : mai 2007

Conférence : 21st International Conference on Advanced Information Networking and Applications (AINA-07)

[1]France Telecom R&D, [2]LSR-IMAG

[6.] « **Dynamic Web Services on a Home Service Platform** »
Auteurs : André Bottaro[1,2], Eric Simon[1], Stephane Seyvoz[1] et Anne Gérodolle[1]
Date : Septembre 2007
Conférence : soumit à : 22st International Conference on Advanced Information Networking and Applications (AINA-08)

[1]France Telecom R&D, [2]LSR-IMAG

5.2 Spécification

[7.] « **RFP 86 - DPWS Discovery Base Driver** »
Auteurs : Sylvain Marié[1], André Bottaro[2], Anne Gérodolle[2], Stéphane Seyvoz[2], Eric Simon[2]
Date : 7 mai 2007
Organisme de standardisation : $OSGi^{TM}$ Alliance

[1]Schneider Electric, [2] France Telecom R&D

[8.] « **Devices Profile for Web Services** »
Auteurs : S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, J. Thelin, D. Walter, J. Weast, D. Whitehead, D. Wright, Y. Yarmosh
Date : février 2006
Organisme de standardisation : W3C

Microsoft© Developers
http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf

[9.] « **Web Services Addressing** »
Auteurs : Don Box[1], Erik Christensen[1], Francisco Curbera[2], Donald Ferguson[2], Jeffrey Frey[2], Marc Hadley[3], Chris Kaler[1], David Langworthy[1], Frank Leymann[2], Brad Lovering[1], Steve [1], Steve Millet[1], Nirmal Mukhi[2], Mark Nottingham[4], David Orchard[4], John Shewchuk[1], Eugène Sindambiwe[5], Tony Storey[2], Sanjiva Weerawarana[2], Steve Winkler[5]
Date : 10 août 2004
Organisme de standardisation : W3C

[1]Microsoft, [2]IBM, [3]Sun Microsystems, [4]BEA Systems, [5]SAP
http://www.w3.org/Submission/ws-addressing/

[10.] « **Web Services Dynamic Discovery** »
Auteurs : John Beatty[4], Gopal Kakivaya[1], Devon Kemp[2], Thomas Kuehnel[1], Brad Lovering[1], Bryan Roe[3], Christopher St. John[5], Jeffrey Schlimmer[1], Guillaume Simonnet[1], Doug Walter[1], Jack Weast[3], Yevgeniy Yarmosh[3], Prasad Yendluri[5]
Date : avril 2005
Organisme de standardisation : W3C

[1]Microsft, [2]Canon, [3]Intel, [4]BEA Sytem, [5]webMethods

http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf

[11.] « **Web Services Eventing** »
Auteurs : Don Box[1], Luis Felipe Cabrera[1], Craig Critchley[1], Francisco Curbera[2], Donald Ferguson[2], Steve Graham[2], David Hull[3], Gopal Kakivaya[1], Amelia Lewis[3], Brad Lovering[1], Peter Niblett[2], David Orchard[4], Shivajee Samdarshi[3], Jeffrey Schlimmer[1], Igor Sedukhin[5], John Shewchuk[1], Sanjiva Weerawarana[2], David Wortendyke[1]
Date : 15 mars 2006
Organisme de standardisation : W3C

[1]Microsoft, [2] IBM, [3]TIBCO Software, [4]BEA Systems, [5]Computer Associates

http://www.w3.org/Submission/WS-Eventing/

[12.] « **Web Services Description Language (WSDL) 1.1** »
Auteurs : Erik Christensen[1], Francisco Curbera[2], Greg Meredith [1], Sanjiva Weerawarana[2]
Date : 15 mars 2001
Organisme de standardisation : W3C

[1]Microsoft, [2]IBM Research

http://www.w3.org/TR/wsdl

[13.] « $OSGi^{TM}$ **Service Platform : Core Specification** » ; Release 4
Date : août 2005
Organisme de standardisation : $OSGi^{TM}$ Alliance

[14.] « $OSGi^{TM}$ **Service Platform : Service Compendium** » ; Release 4
Date : août 2005
Organisme de standardisation : $OSGi^{TM}$ Alliance

5.3 Référence technique

[15.] http://www.service-architecture.com

[16.] http://msdn.microsoft.com/webservices/webservices/understanding/specs/default.aspx

[17.] http://www.orchestranetworks.com/fr/soa/standardsws.cfm

[18.] http://msdn2.microsoft.com/fr-fr/webservices/Aa740689.aspx

[19.] http://www.service-architecture.com/

[20.] http://www.humbertocervantes.net/
[21.] http://www.artima.com/spontaneous/upnp_digihome.html
[22.] http://www-adele.imag.fr/users/Didier.Donsez/cours/upnposgi/tutorial.htm
[23.] http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc
[24.] http://www.upnp-ic.org/home/
[25.] http://xml.coverpages.org/ni2003-08-22-a.html
[26.] http://www.osgi.org/
[27.] http://www.humbertocervantes.net/
[28.] http://oscar.objectweb.org/
[29.] http://cwiki.apache.org/FELIX/index.html
[30.] http://www.zeroconf.org/
[31.] http://www.dns-sd.org/
[32.] http://www.apple.com/fr/macosx/features/bonjour/