Environnements d'exécution pour passerelles domestiques

Yvan Royon

▶ To cite this version:

Yvan Royon. Environnements d'exécution pour passerelles domestiques. Réseaux et télécommunications [cs.NI]. INSA de Lyon, 2007. Français. NNT : . tel-00271481

HAL Id: tel-00271481
https://theses.hal.science/tel-00271481

Submitted on 9 Apr 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Nº d'ordre : 2007-ISAL-0104 Année 2007

Thèse

Environnements d'exécution pour passerelles domestiques

Présentée devant

l'Institut National des Sciences Appliquées de Lyon
École Doctorale Informatique et Information pour la Société

pour l'obtention du

Grade de Docteur

spécialité informatique

par

Yvan ROYON

soutenue le 13 Décembre 2007

Devant le jury composé de

Président :	Didier Donsez,	Professeur,	Université Grenoble I
Directeurs :	Stéphane Frénot,	Maître de conférences,	INSA de Lyon
	Stéphane Ubéda,	Professeur,	INSA de Lyon
Rapporteurs :	Olivier Festor,	Directeur de recherche,	LORIA
	Gilles Muller,	Professeur,	École des Mines de Nantes
Examinateurs :	Gilles Grimaud,	Maître de conférences,	Université Lille I
	Nicolas Le Sommer,	Maître de conférences,	Université de Bretagne Sud

Thèse effectuée au sein du Centre d'Innovation en Télécommunications et Intégration de Services (CITI) de l'INSA de Lyon et de l'équipe Architecture Réseaux et Systèmes (Ares) de l'INRIA Rhône-Alpes

Résumé

Le marché des passerelles domestiques (les modems intelligents) évolue vers un nouveau modèle économique. Le modèle d'aujourd'hui, appelé triple play, amène la connectivité IP à nos maisons, ainsi que deux services additionnels : la téléphonie et la vidéo sur IP. Ces trois services sont contrôlés et gérés par une seule entité économique : le fournisseur d'accès.

Le modèle de remplacement proposé par les constructeurs et les opérateurs, appelé multiplay, ouvre le marché à de multiples fournisseurs de services spécialisés dans la voix ou la vidéo. Ceci permet aux usagers domestiques d'accéder à des contenus plus variés, ou encore de choisir une ou plusieurs offres concurrentielles.

Un troisième modèle économique devient dès lors plausible. Puisque le marché est ouvert à de multiples acteurs, et puisque les passerelles domestiques deviennent de plus en plus puissantes, pourquoi limiter les offres de services à la voix et à la vidéo ? Il en existe d'autres en pleine expansion, comme l'aide aux personnes âgées, la domotique ou la télésécurité. Il faut donc donner un sens très générique au terme service, et permettre à la passerelle domestique d'héberger n'importe lequel de ces services. Nous appelons ce modèle multi-services.

La nouveauté technique des passerelles multi-services est qu'elles doivent supporter le déploiement, l'exécution et la gestion de plusieurs éléments logiciels (modules), en provenance de fournisseurs différents. Ceci se traduit par des besoins en terme d'isolation d'exécution locale, de gestion à distance, d'infrastructure de déploiement et de modèle de programmation.

L'isolation d'exécution est un compromis entre le coût en performance et le niveau d'isolation obtenu. Les passerelles domestiques étant limitées en mémoire, disque et processeur, la performance et la complexité sont les critères prépondérants. Il est requis de pouvoir distinguer les modules logiciels selon leur fournisseur. La technique d'isolation choisie donne des variantes du modèle économique ; par exemple, il sera possible ou non de partager certains modules à la demande entre fournisseurs, comme un codec vidéo ou un serveur Web.

De la même manière, chaque fournisseur doit pouvoir gérer ses propres services à distance. Une variante est que chaque fournisseur choisit sa propre technologie de gestion.

L'infrastructure de déploiement pose des problèmes de passage à l'échelle et de mécanismes de mise à jour des modules. Plusieurs niveaux de granularité sont possibles dans la gestion de leur cycle de vie, comme la mise en pause ou l'intégration de la sécurité.

Enfin, le modèle de programmation choisi a un impact fort sur le découplage entre modules, leur réutilisabilité et la durée de leur développement.

Pour implémenter ces quatre familles de besoins sur des cibles à ressources limitées, nous retenons deux environnements d'exécution : Java/OSGi et C/Linux. OSGi a l'avantage d'imposer un modèle propre de programmation orientée service. Cette plate-forme gère déjà le déploiement et le cycle de vie des modules ; il lui manque cependant les notions de multi-utilisateurs et d'isolation. GNU/Linux dispose de nombreux outils d'isolation et de déploiement de modules. Il lui faut par contre une intégration cohérente et une gestion unifiée de tous ces outils. Nous proposons donc de combler les fonctionnalités manquantes à ces deux environnements, afin d'obtenir un système conforme au modèle multi-services.

Summary

The home gateway market evolves towards new business models. Today's model, called triple play, brings Internet connectivity to our homes, along with two additional services : telephony and video over IP. These three services are managed by a single business entity : the access provider.

Constructors and operators propose a replacement model, called multi-play, which opens the market to multiple voice and video providers. This allows end-users to access more content and to choose between competing service plans.

A third business model then becomes possible. Since the market is open to multiple business actors, and since home gateways become more and more powerful, why should we limit services to voice and video ? An increasing number of other services exist, such as health care, support for the disabled, home automation and telesecurity. The notion of service must be broadened, and the home gateway must be enhanced to be able to host any kind of service. We call this model multi-service.

Technical novelties with multi-service home gateways are that they support deploying, executing and managing several software modules that come from different providers. This translates into specific needs in terms of execution isolation, remote management, deployment infrastructure, and programming model.

Execution isolation is a compromise between the level of isolation and the impact on performance. Since home gateways have limited hardware resources, performance and complexity are key factors. They need a level of isolation that allows to separate software modules based on their providers. When choosing how this separation is enforced, we create variants in the business model ; for instance, some modules, such as a video codec or a web server, may or may not be shared among service providers.

Each provider manages his own services. Similarly, variants are possible : one is that each provider chooses his own management technology, another is that the access provider dictates one.

The deployment infrastructure must scale, both service-wise and user-wise, and propose update mechanisms. Variants lie in life cycle management granularity, e.g. pausing services may or may not be possible.

Lastly, the programming model impacts decoupling between modules, their reusability, and their time-to-develop.

To implement these four families of needs on resource-constrained targets, we focused on two execution environments, which allow different aforementioned variants : Java/OSGi and C/Linux. OSGi boasts a clean, service-oriented programming model, and already manages deployment and life cycle of modules. However, it lacks multi-user features and the related isolation mechanisms. GNU/Linux offers many tools for isolation and module deployment. However, it needs a tight integration and a unified management of such tools. Therefore, we propose to fill these lacking features on these two environments, so they can conform to the multi-service model.

Remerciements

Plusieurs personnes ont eu un rôle important dans la rédaction de ce mémoire, dont notamment celles qui figurent en première page de couverture.

Je tiens ainsi à remercier mes rapporteurs, Olivier Festor et Gilles Muller, pour avoir accepté d'évaluer mon travail, ainsi que pour leurs commentaires pertinents.

Je remercie également les membres du jury Didier Donsez, Gilles Grimaud et Nicolas Le Sommer, pour avoir accepté de subir ma soutenance. J'ajoute un second merci à Nicolas pour son retour attentif sur le présent document.

Ces travaux n'auraient ni abouti ni même débuté sans le maître d'orchestre, Stéphane Frénot, qui m'a encadré pendant ma thèse et mon DEA. J'en garde de bons souvenirs de discussions scientifiques, de séances de code à deux ou encore de débats œnologiques, parce que Pasteur avait raison.

Le suivant dans la ligne de mire est Stéphane Ubéda, mon directeur de thèse, qui gère ses thésards tout comme son laboratoire avec une grande ouverture d'esprit et une honnêteté qui force le respect. La bonne ambiance qui règne au CITI est, je pense, largement due aux libertés dont nous disposons.

Enfin, je remercie Christèle Bouchat et son équipe, pour m'avoir accueilli durant trois mois chez Alcatel, et pour m'avoir laissé carte blanche dans la gestion de mon travail.

```
 ----------------------------------
/ moi, je serais chercheur, ben je \
\ saurais pas quoi chercher        /
 ----------------------------------
   \
    \  \_\_    _/_/
     \     \__/
           (oo)\_______
           (__)\       )\/\
               ||----w |
               ||     ||
```

Sur une note plus personnelle, je remercie les arbres dont vous tenez les cadavres, alors qu'ils n'ont rien demandé. Mes salutations à la coinche et au picon ; comme on dit, leur inventeur, c'était pas la moitié d'un con.

Merci aux ami(e)s qui se reconnaîtront.

Merci enfin à mes parents que j'admire.

Table des matières

1 Introduction **1**
1.1 Constat de départ : évolution de l'accès à Internet pour la maison 1
1.2 Cadre des travaux : la passerelle domestique 4

2 État de l'art **7**
2.1 Écosystème des services Internet à domicile 7
2.1.1 Réseau opérateur et réseau Internet 8
2.1.2 Réseau domestique 10
2.1.3 Jonction des deux mondes : la passerelle domestique 13
2.1.4 Bilan et plan du chapitre 15
2.2 Protocoles de gestion pour passerelles domestiques 16
2.3 Environnements d'exécution existants liés aux passerelles domestiques 19
2.3.1 GNU/Linux 19
2.3.2 Java/OSGi 21
2.3.3 Environnements d'exécution connexes 24
2.4 Modèles de programmation 28
2.4.1 Langages natifs pour GNU/Linux 28
2.4.2 Objets, composants, services 28
2.5 Mécanismes d'isolation 32
2.5.1 Terminologie 32
2.5.2 Types de virtualisation 33
2.5.3 Tableau synthétique 40

3 Environnements d'exécution pour passerelles domestiques **43**
3.1 Expression des besoins 43
3.1.1 Notion d'environnement d'exécution 43
3.1.2 Besoins des passerelles domestiques 44
3.1.3 Besoins des passerelles domestiques multi-services 45
3.2 Spécification des aspects déploiement 45
3.2.1 Unités de déploiement 45
3.2.2 Cycle de vie 46
3.2.3 Gestion des dépendances 47
3.3 Spécification des aspects isolation multi-acteurs 49
3.4 Spécification des aspects gestion 50
3.5 Modèle d'environnement d'exécution pour passerelles domestiques 52
3.6 Styles d'implantation 53

4 Mise en œuvre pour Java/OSGi **55**
4.1 Java/OSGi et le modèle multi-services 55

4.2 Proposition . . . 56
4.2.1 Support local d'exécution . . . 56
4.2.2 Gestion des passerelles logicielles . . . 59
4.3 Implantation . . . 60
4.3.1 VOSGi : Virtual OSGi . . . 60
4.3.2 MOSGi : Managed OSGi . . . 63
4.4 Cas d'utilisation : services e-frigo et UPnP/AV . . . 66
4.5 Expérimentations . . . 67
4.5.1 Environnement de test et méthodologie d'évaluation . . . 67
4.5.2 Coût mémoire . . . 68
4.5.3 Comportement de la couche JMX . . . 69
4.6 Discussion . . . 71
4.6.1 Phase de login . . . 71
4.6.2 Performances . . . 72
4.6.3 Partage de services . . . 74

5 Mise en œuvre pour GNU/Linux 75
5.1 GNU/Linux et le modèle multi-services . . . 76
5.1.1 Fonctionnalités existantes . . . 76
5.1.2 Outils existants et leurs limites . . . 77
5.2 Proposition : HGL . . . 80
5.2.1 Architecture . . . 80
5.2.2 Aspects déploiement . . . 81
5.2.3 Aspects multi-acteurs . . . 82
5.3 Implantation . . . 83
5.3.1 Aspects déploiement . . . 83
5.3.2 Cycle de vie . . . 84
5.3.3 Aspects multi-acteurs . . . 85
5.3.4 Interface de gestion . . . 86
5.4 Expérimentations . . . 87
5.4.1 Intégration avec TR-069 . . . 87
5.4.2 Mesures de performance . . . 87
5.5 Discussion . . . 89
5.5.1 Fonctionnalités . . . 89
5.5.2 Choix d'implantation . . . 90
5.5.3 Partage de services dans HGL . . . 91
5.5.4 Modèle de programmation : services et méta-données . . . 91

6 Synthèse 93
6.1 Bilan des implantations . . . 93
6.2 Bilan des styles d'implantation . . . 94
6.2.1 Différences de performances . . . 94
6.2.2 Différences de fonctionnalités . . . 95
6.2.3 Dépendances exprimées et dépendances réelles . . . 97
6.2.4 Tableau comparatif . . . 100
6.3 Autres styles d'implantation possibles . . . 101
6.3.1 GNU/Linux et SOA . . . 101

6.3.2 GNU/Linux et SOP 102
6.4 Conclusion 103

Table des figures

1.1 Modèle économique connectivité 2
1.2 Modèle économique *triple play* 2
1.3 Modèle économique *multi-play* 2
1.4 Modèle économique multi-services 2
1.5 Royaumes multiples de gestion autour de la passerelle domestique 3

2.1 Bilan des normes et standards par couches de communication 15
2.2 Bilan des normes et standards par centres d'intérêt 16
2.3 Royaumes de gestion et technologies associées 17
2.4 Cycle de vie d'un *bundle* OSGi 22
2.5 Cycle de vie d'une Xlet dans Java ME 27
2.6 API offerte par l'environnement d'exécution 31
2.7 Machines virtuelles mono-tâches et multi-tâches 37
2.8 Bilan des techniques de virtualisation 41

3.1 Cycle de vie minimal d'une unité de déploiement 46
3.2 Fonctionnalités d'un environnement d'exécution multi-services 53
3.3 Vue en couches d'un environnement d'exécution multi-services 53

4.1 OSGi : fonctionnalités multi-services existantes 56
4.2 OSGi pour passerelles multi-services 58
4.3 OSGi : diagramme architectural 60
4.4 Hiérarchies de chargeurs de classes 62
4.5 JMXconsole : console de gestion distante 65
4.6 VOSGi/MOSGi : Consommation mémoire sur 3 environnements de test 69
4.7 Comportement typique d'un système sous stress 70
4.8 MOSGi : temps de réponse de l'agent JMX 71

5.1 GNU/Linux : fonctionnalités multi-services existantes 76
5.2 HGL : architecture multi-services pour GNU/Linux 81
5.3 HGL : diagramme d'état pour un bundle 81
5.4 HGL : vue en couches logicielles 82
5.5 HGL : vue gestion des acteurs en présence 83
5.6 HGL : vue processus des coins utilisateur 85
5.7 HGL : consommation mémoire 88

Liste des tableaux

2.1 Support multi-applications dans Java 38

4.1 Caractéristiques techniques des environnements de test 67

5.1 HGL : utilisation mémoire du processus hgl_init 87
5.2 HGL : taille disque du binaire 88

6.1 Comparaison entre VOSGi/MOSGi et HGL 101

Extraits de code

3.1 Interface de gestion pour un utilisateur 51
3.2 Interface de gestion pour l'administrateur 52

4.1 Commande shell pour VOSGi 60

5.1 HGL : commandes pour l'interface de gestion 86

1 Introduction

1.1 Constat de départ : évolution de l'accès à Internet pour la maison 1
1.2 Cadre des travaux : la passerelle domestique 4

1.1 Constat de départ : évolution de l'accès à Internet pour la maison

En 2006, 44% des ménages européens sont équipés d'un accès au réseau Internet[1]. Ces dernières années, les offres des fournisseurs d'accès à Internet pour les particuliers ont évolué, non seulement en performance, mais surtout sur le plan économique.

Depuis la fin des années 1990 et jusqu'à il y a quelques années, les abonnements des particuliers auprès des fournisseurs d'accès à Internet incluaient, comme le nom l'indique, l'accès au réseau des réseaux. En d'autres termes, il s'agissait d'assurer la connectivité aux niveaux 3-4 du modèle OSI [1] (TCP/IP). Dans la maison, un *modem* faisait l'interface entre le réseau opérateur (via le « dernier kilomètre ») et un micro-ordinateur, comme représenté sur la figure 1.1.

Aujourd'hui, les mêmes fournisseurs d'accès proposent également des services de téléphonie et de télévision sur IP. Dans ce modèle économique nommé *triple play* (figure 1.2), le fournisseur d'accès passe un accord commercial avec un fournisseur de contenu télévisé et un fournisseur de téléphonie. L'usager final, que nous appellerons usager au domicile, ne choisit pas son fournisseur pour ces deux services additionnels. Le marché est donc verrouillé par les fournisseurs d'accès, qui fixent leurs conditions sur le contenu multimédia et sur les tarifs.

L'étape suivante est alors de casser ce verrou, et d'ouvrir le marché à des fournisseurs de services multimédia concurrents. L'objectif est de dynamiser le marché en multipliant les offres, en donnant le choix à l'usager au domicile, et en intégrant des offres spécialisées telles que la vidéo à la demande. Ce modèle économique, nommé *multi-play*, est représenté

[1] Commission Européenne, Eurobaromètre spécial n° 259 « L'Union européenne et ses voisins », 2006. `http://ec.europa.eu/public_opinion/archives/ebs/ebs_259_fr.pdf`

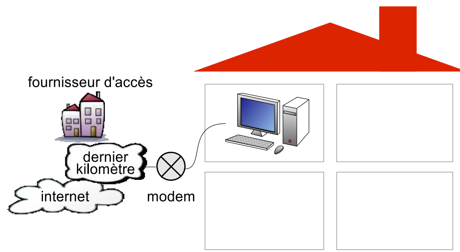


Fig. 1.1: Modèle économique connectivité

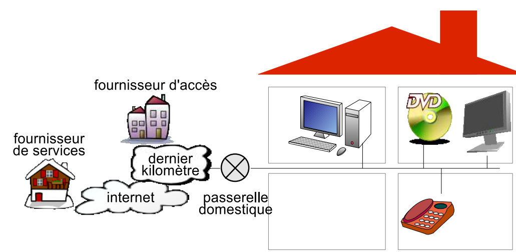


Fig. 1.2: Modèle économique *triple play*

en figure 1.3. Il est poussé par les acteurs économiques actuels, opérateurs et constructeurs inclus [2].

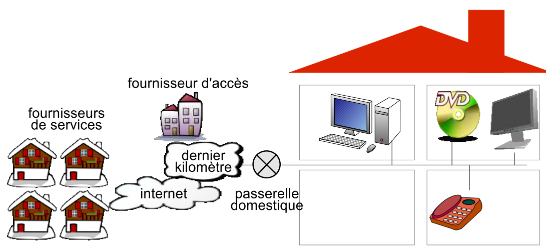


Fig. 1.3: Modèle économique *multi-play*

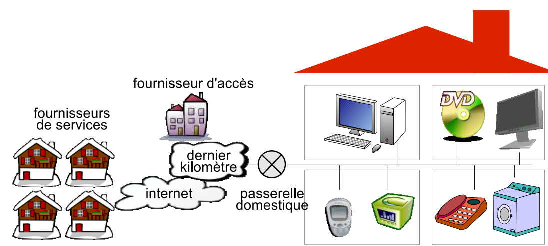


Fig. 1.4: Modèle économique multi-services

Cette thèse se focalise sur un modèle économique ultérieur au *multi-play*. Nous proposons en effet d'intégrer d'autres services que la voix et la vidéo sur IP (figure 1.4). En effet, de nombreux projets de recherche actuels étudient les services d'aide au domicile pour les enfants en bas âge, les personnes âgées ou les personnes handicapées, ce que l'on appelle *qualité de vie*. À titre d'exemple, une personne qui souffre d'insuffisances cardiaques utilise un *pacemaker* ; si un rythme cardiaque anormal est détecté, une alarme est remontée à l'hôpital le plus proche, ainsi qu'à la famille, pour demander une intervention rapide. Également, si un enfant en bas âge se déplace dans une pièce dite à risque comme la cuisine, un mécanisme de télésurveillance déclenche une alarme pour la personne en charge, par le biais de l'écran, du haut-parleur ou de l'équipement mobile le plus proche.

D'autres projets se concentrent sur l'intégration de la domotique et la création d'environnements domestiques communicants [3]. Ainsi, un répondeur téléphonique, un magnétoscope ou encore un réfrigérateur communicant peuvent être programmés et consultés, localement et à distance. En cas de panne, le réparateur le plus proche est notifié. Si un compartiment du réfrigérateur est vide, une liste de courses est préparée automatiquement.

Nous voyons là l'opportunité de généraliser la notion de *service*, afin de préparer le marché à intégrer toutes les extensions possibles : celles imaginées dans les projets sus-cités, et celles encore à venir. Nous appelons ce modèle économique *multi-services*.

La fourniture de services Internet au domicile fait interagir trois théâtres d'opérations, tant techniques qu'économiques : le réseau distant, le réseau domestique et la passerelle domestique. Le réseau distant est composé du réseau opérateur géré par le fournisseur d'accès, du « dernier kilomètre » qui connecte le réseau opérateur au domicile de l'usager, et des interconnexions vers d'autres réseaux qui forment Internet. Le réseau domestique est un réseau local qui interconnecte les équipements communicants de la maison. Enfin, la passerelle domestique est le point de jonction entre le réseau opérateur et le réseau domestique.

Sur ces trois théâtres d'opération du modèle économique multi-services, trois familles d'acteurs interviennent. Le fournisseur d'accès permet la connectivité réseau entre la passerelle domestique et le réseau opérateur. Il est également responsable du bon fonctionnement global de la passerelle domestique. Les fournisseurs de services proposent des services en ligne, de type contenu multimédia mais aussi de type supervision et configuration d'un équipement domestique. Finalement, l'usager au domicile utilise ses équipements communicants et les services qu'ils offrent. Il obtient ces équipements ou ces services auprès des fournisseurs de services.

Ces trois types d'acteurs économiques sont susceptibles d'accéder à la passerelle domestique, d'y configurer des préférences, et de déployer des services. Nous utiliserons le terme de *royaume* de gestion. Un royaume englobe un type d'acteurs économiques, ainsi que leurs actions de gestion sur les trois théâtres d'opérations.

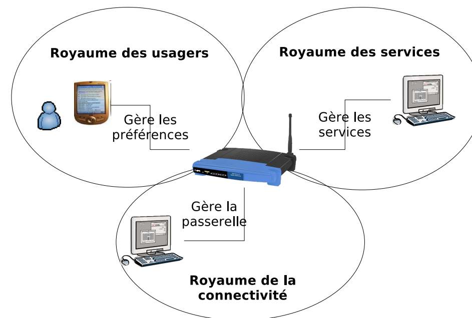


Fig. 1.5: Royaumes multiples de gestion autour de la passerelle domestique

Comme représentés en figure 1.5, ces royaumes sont le fournisseur d'accès, les fournisseurs de services, et les usagers au domicile. Le fournisseur d'accès est responsable de la connectivité aux couches 1 à 4 ; il doit donc configurer les paramètres de connexion (p.ex. DHCP). Il peut aussi configurer le pare-feu ou tout autre dispositif agissant sur les mêmes couches OSI. Dans le royaume des *services*, chaque fournisseur déploie ses services puis les gère. Enfin, dans le royaume des usagers, les services sur la passerelle domestique peuvent enregistrer des préférences et des options de configuration via une interface utilisateur.

1.2 Cadre des travaux : la passerelle domestique

Les travaux de cette thèse se focalisent sur la passerelle domestique. Plus précisément, nous nous intéressons à la manière dont les services y sont exécutés et gérés. Un service est un agrégat de code et de données. Quel que soit son royaume de gestion ou son théâtre d'opérations (la maison, le réseau Internet ou la vie en ligne de l'usager), chaque service contient une logique métier, ainsi que des paramètres et préférences en provenance des trois royaumes de gestion. La logique métier est déployée sur la passerelle domestique depuis l'un des trois royaumes, et les paramètres sont surveillés et manipulés depuis les trois royaumes.

Les nouveaux défis lancés par le modèle économique multi-services sont doubles. Le premier défi est que la passerelle domestique fonctionne sans interruption, même quand l'usager au domicile souscrit à un nouveau service ou en annule un ancien. Dans les modèles économiques antérieurs, si un service est mis à jour, l'usager doit redémarrer la passerelle domestique pour le prendre en compte. Dans le cas du multi-services, ce redémarrage perturberait les services des autres fournisseurs, dont certains sont importants pour la qualité de vie de l'usager. Le redémarrage systématique est donc à proscrire. À la place, la passerelle multi-services doit supporter le déploiement dynamique de services, leur installation, leur démarrage, leur arrêt et leur suppression. Dans la littérature, ce caractère dynamique est fourni par un système d'exploitation ou un intergiciel hébergé sur la passerelle domestique.

Le second défi est d'identifier les responsabilités de chacun. En effet, puisque les services sont déployés et gérés par plusieurs acteurs économiques venant de plusieurs royaumes de gestion, la passerelle multi-services est un environnement multi-acteurs, ou multi-utilisateurs. En cas de dysfonctionnement sur la passerelle domestique, il est important d'identifier l'acteur économique fautif. Pour cela, la passerelle domestique doit identifier et isoler les services en fonction de leur fournisseur. Dans la littérature, les mécanismes d'isolation multi-utilisateurs sont également le fait d'un système d'exploitation ou d'un intergiciel.

En plus de ces deux défis, la passerelle multi-services offre aux multiples acteurs économiques un accès distant pour que chacun contrôle et agisse sur ses services. Cette fonction est remplie par une infrastructure de gestion au-dessus du système d'exploitation ou de l'intergiciel.

Nous proposons donc d'identifier les contraintes des environnements d'exécution pour passerelles domestiques multi-services. Ces environnements d'exécution comportent un système d'exploitation ou un intergiciel pour déployer des services et les isoler selon leur fournisseur, ainsi qu'une infrastructure de gestion locale et distante. Nous étudions la représentation logicielle des services et leurs relations de dépendance. Nous montrons également le modèle de programmation associé, les opérations de gestion des services, et le mode de fonctionnement dû aux multiples acteurs économiques en présence. Nous proposons deux implantations de passerelle multi-services, l'une basée sur Java/OSGi, l'autre sur GNU/Linux.

Le chapitre 2 présente les travaux existants autour des passerelles domestiques. Nous abordons les standards des réseaux opérateurs, des réseaux domestiques et des passerelles domestiques ayant attrait à la fourniture de services au domicile. Sur les passerelles domestiques mêmes, nous présentons les travaux existants liés aux activités de gestion ainsi qu'aux environnements d'exécution. Nous montrons que la passerelle domestique fait le lien entre plusieurs technologies de gestion, dues aux trois royaumes. Nous montrons également que l'environnement d'exécution le plus utilisé par les passerelles domestiques est GNU/Linux, et que OSGi est le candidat le plus probable pour le remplacer. Nous notons également les avantages que présentent ces deux technologies en termes de fonctionnalités et de modèle de programmation pour le modèle multi-services.

Le chapitre 3 page 43 définit les besoins des passerelles domestiques multi-services en terme d'environnement d'exécution. Nous proposons une architecture répondant à ces besoins, dont la mise en œuvre est décrite dans les deux chapitres suivants. Ces besoins concernent le déploiement des services, l'isolation des acteurs économiques et l'intégration d'une infrastructure de gestion locale et distante.

Notre démarche est ensuite de partir d'environnements d'exécution existants et de les adapter au modèles multi-services. Le chapitre 4 page 55 propose d'adapter Java/OSGi pour répondre aux contraintes multi-services, tandis que le chapitre 5 page 75 en fait de même pour GNU/Linux. Java/OSGi présente l'intérêt de gérer de manière cohérente une implantation partielle d'environnement multi-services, tandis que GNU/Linux dispose d'une implantation presque complète mais éclatée en de multiples outils non cohérents. Ces deux chapitres décrivent également l'implantation réalisée durant ces travaux de thèse. Ils présentent les performances obtenues, en particulier la consommation mémoire et disque, puis discutent des choix d'implantation que nous avons faits.

Le chapitre 6 page 93 compare les deux implantations réalisées. Il discute de leurs différences, notamment en termes de fonctionnalités et de modèle de programmation. Il présente également des alternatives envisageables, pour conclure par une synthèse des résultats et des directions pour de futurs travaux.

État de l'art

2

2.1 Écosystème des services Internet à domicile 7
2.1.1 Réseau opérateur et réseau Internet 8
2.1.2 Réseau domestique 10
2.1.3 Jonction des deux mondes : la passerelle domestique 13
2.1.4 Bilan et plan du chapitre 15
2.2 Protocoles de gestion pour passerelles domestiques 16
2.3 Environnements d'exécution existants liés aux passerelles domestiques . . . 19
2.3.1 GNU/Linux 19
2.3.2 Java/OSGi 21
2.3.3 Environnements d'exécution connexes 24
2.3.3.1 .NET CLR 24
2.3.3.2 Java ME 25
2.4 Modèles de programmation 28
2.4.1 Langages natifs pour GNU/Linux 28
2.4.2 Objets, composants, services 28
2.5 Mécanismes d'isolation 32
2.5.1 Terminologie 32
2.5.2 Types de virtualisation 33
2.5.2.1 Virtualisation au niveau ISA 33
2.5.2.2 Virtualisation au niveau HAL 34
2.5.2.3 Para-virtualisation au niveau HAL 34
2.5.2.4 Systèmes d'exploitation à conteneurs 35
2.5.2.5 Support matériel pour la virtualisation 35
2.5.2.6 Virtualisation en espace utilisateur 36
2.5.2.7 Virtualisation au niveau librairie 36
2.5.2.8 Virtualisation au niveau intergiciel 36
2.5.2.9 Virtualisation niveau application 40
2.5.3 Tableau synthétique 40

2.1 Écosystème des services Internet à domicile

Pour amener des services Internet au domicile des usagers, deux réseaux distincts interviennent : le réseau dans la maison et le réseau du fournisseur d'accès. Le premier relie les

équipements domestiques communicants et leur permet d'interagir. Le second donne accès au réseau mondial, qui relie les divers fournisseurs de services avec le réseau domestique. Pour chaque réseau, un ensemble de normes et de standards définit ses multiples aspects techniques, comme la connectique, la couche de transport, les formats d'échanges multimédia, ou encore les protocoles de gestion. En plus des deux réseaux, la passerelle domestique qui les relie dispose de ses propres normes et standards, qui traitent en plus de l'hébergement de services applicatifs.

Le début de ce chapitre présente ces normes, standards et autres travaux, existants ou en cours d'élaboration. Nous les aborderons séparément pour les réseaux opérateurs, les réseaux domestiques et les passerelles domestiques, dans cet ordre. Par la suite, nous recentrerons l'étude sur les passerelles domestiques, en particulier pour leur gestion et l'hébergement de services applicatifs.

2.1.1 Réseau opérateur et réseau Internet

Le réseau distant, soit le réseau extérieur à la maison, comprend trois parties : le « dernier kilomètre », qui relie la maison de l'abonné aux équipements du fournisseur d'accès, le réseau opérateur lui-même, et les liens vers d'autres réseaux pour former l'interconnection Internet.

Terminologie Le terme **NGN** (*Next-Generation Network*, réseau prochaine génération) désigne l'évolution conjointe de ces trois parties du réseau distant, dans l'objectif de fournir des services à nos domiciles. Dans ce contexte, un **service** est un flux de données ou un flux multimédia sur le réseau, comme c'est le cas dans les modèles économiques *triple play* et *multi-play*. Du côté de la maison, le point de terminaison d'un service est appelé un **CPE** (*Customer Premises Equipment*, équipement chez le client). Concrètement, un CPE peut être la passerelle domestique, un ordinateur, ou un équipement dédié au décodage de la télévision sur IP (*set-top box*) par exemple.

Objectifs Le défi des NGN est d'assurer une qualité de service (QoS) de bout en bout, via un réseau moderne tout IP, sur des flux réseau de type voix, vidéo et données. L'expression « de bout en bout » signifie que les services sont maîtrisés depuis le fournisseur jusqu'au CPE au domicile de l'usager. Outre mettre en place des solutions de QoS réseau, les NGN nécessitent conjointement de maîtriser les flux de gestion et la sécurité de bout en bout. Enfin, les CPE pouvant être mobiles dans et hors de la maison, il faut intégrer le facteur « bout en bout » et le facteur mobilité ; on appelle cela la convergence fixe-mobile.

Un ensemble de normes et de standards existants ou en cours d'élaboration abordent ces problèmes au niveau des couches 1 à 4 du modèle OSI [1].

Normes Les principaux travaux de normalisation internationaux proviennent du NGN Focus Group de l'ITU-T [4], lancé en juin 2004, et duquel les acteurs européens sont absents. La QoS est réalisée par IP sur ATM (IPoA). Les accents sont mis d'une part sur le fonctionnement du réseau, avec les flux de contrôle et de signalisation, et d'autre part sur l'authentification et la « vie privée » des utilisateurs. L'administration du réseau est réalisée par CMIP [5].

Du côté européen, l'organisme de normalisation ETSI, via son entité TISPAN [6], définit en particulier une connectique (DTM : *Dynamic synchronous Transfer Mode*, sur fibre optique), ainsi qu'un modèle économique multi-fournisseurs. L'ETSI définit également le protocole GAT (Generic Addressing & Transport), dans l'espace de signalisation du réseau. GAT sert à amener des services au domicile, en insistant sur le choix multi-fournisseurs, la convergence fixe-mobile et l'utilisation des mécanismes des réseaux intelligents.

Standards L'ETSI TISPAN, le 3GPP CN5 et le groupe Parlay proposent OSA (*Open Service Access*) [7], un intergiciel pour opérateurs de télécommunications dédié aux réseaux fixes, mobiles et d'entreprise. L'implantation d'OSA, nommée Parlay d'après l'entité qui l'a créée, est basée sur Java. Des équipements d'un réseau de télécommunications hébergent une passerelle OSA, qui est un serveur d'applications. Chaque passerelle OSA héberge des services OSA, qui correspondent à une partie de l'interface de programmation (API) de Parlay. Par exemple, on trouve des services pour la facturation, pour la localisation de l'utilisateur, le contrôle d'appels multimédia ou d'appels de type conférence. Chaque service est exprimé en fonction d'interfaces Java. Les passerelles OSA détectent la présence de nouveaux services, et fournissent ces interfaces aux applications que l'opérateur de télécommunications déploie pour gérer son réseau.

Projets européens Le projet européen MUSE [2] (2004–2007), regroupant entre autres de nombreux opérateurs et constructeurs, propose des travaux assez similaires à ceux de TISPAN en insistant sur le fonctionnement du réseau ainsi que la QoS sur IP ou Ethernet. Cependant, MUSE étudie aussi la question du dernier kilomètre et du point de jonction chez l'usager au domicile.

NGN Initiative (NGNi) [8] (2001–2004), comme son nom l'indique, travaille sur les réseaux d'accès, la mobilité et la qualité de service dans les réseaux NGN.

Un dernier projet que nous citerons à titre d'exemple est MediaNet [9] (2001–2004). Il se focalise sur les communications multimédia et l'échange de données audio et vidéo. Les délivrables de MediaNet s'intéressent donc à l'acheminement, au stockage et au décodage de flux multimédia. Ces opérations sont réparties entre le réseau du fournisseur de flux, le réseau de l'opérateur et le réseau domestique.

2.1.2 Réseau domestique

Le réseau domestique interconnecte tous les équipements communicants de la maison. L'objectif est que les ordinateurs, mais aussi les consoles de jeux, la hi-fi, la domotique, l'e-électroménager, la téléphonie sur IP ou encore les équipements mobiles puissent s'échanger des informations.

Les deux problèmes majeurs sont l'interconnection physique des équipements et leur interopérabilité au niveau applicatif. Schématiquement, les normes actuelles s'attaquent plutôt au premier problème, et les efforts de standardisation se concentrent sur le second.

Normes Les travaux de normalisation du CENELEC (TC 205 - *Home and Building Electronic Systems*) [10], tout comme les standards produits par le CEBus *Industry Council* (CIC) [11], s'intéressent en particulier aux caractéristiques physiques des appareils communicants. Ils en définissent les normes de compatibilité électrique.

La norme KNX [12] (EN 50090, ISO/IEC 14543) permet d'interconnecter les équipements domotiques dans une maison ou un bâtiment. La couche application est spécifique au cas d'utilisation (chauffage, pilotage des volets), et la couche de lien physique est rendue transparente.

L'ITU-T, via la recommandation J.190 MediaHomeNet [13], définit les couches basses d'un réseau domestique. Cependant, J.190 n'étant pas accessible librement, nous n'en connaissons pas les détails. Il faut y ajouter les recommandations J.112, J.122 et J.160 qui concernent la couche physique.

Le Digital Home-network Forum (DHF), toujours de l'ITU-T, veut s'attaquer à toutes les couches (du matériel à l'intergiciel et aux applications) de la maison communicante, à l'exception des protocoles pour PC et de la voix sur IP. Typiquement le lien de la passerelle domestique vers le réseau opérateur est sur IP, alors que l'accès vers les équipements communicants de la maison se fait via un protocole non spécifié, par exemple propriétaire. Les travaux du DHF sont toujours en cours et ne sont pas accessibles publiquement.

Standards Pour avoir un aperçu des standards existants, observons-les en fonction des couches du modèle OSI auxquels ils correspondent. Nous les listons en partant des couches basses pour arriver aux couches hautes, comme précédemment. Tout d'abord, citons X10 [14], qui est un protocole de communication pour la domotique. Il utilise les prises secteur de la maison comme couche physique, via la technologie CPL (courant porteur en ligne). X10 cible les capteurs et récepteurs domotiques assez simples, comme des capteurs de température ou des actionneurs de volets.

HAVi (*Home Audio/Video Interoperability*) [15] connecte spécifiquement les équipements audio et vidéo de la maison. La connectique adoptée est celle de FireWire (IEEE 1394).

Nous avons vu que le CEBus Industry Council travaille sur la couche électrique. Les équipements certifiés CEBus *Home Plug and Play* communiquent par CPL. Aussi, le CIC définit un protocole de niveau applicatif nommé CAL (*Common Application Language*), dont l'adoption reste très limitée.

Sur le même modèle, LonWorks [16] propose une couche physique CPL, des couches réseau basées sur le protocole LonTalk (ANSI/EIA 709.1) [17] ainsi qu'une couche applicative spécifique. La cible de LonWorks est le bâtiment, comme par exemple l'immeuble d'une entreprise, plutôt que le domicile.

Citons également l'Alliance Zigbee, menée par des constructeurs. Zigbee [18] cible des réseaux sans fil à bas débit et basse consommation d'énergie, du type domotique, réseaux de bâtiment et réseaux industriels. Trois couches sont définies : réseau, sécurité et application. La couche application impose un schéma d'échange et d'adressage ; elle est donc complètement intrusive dans l'écriture des applications fonctionnant sur Zigbee.

Le *Digital Living Network Alliance* (DLNA) [19] regroupe des acteurs de l'électronique grand public, des ordinateurs et des équipements mobiles. Comme beaucoup de groupes d'industriels dans le domaine, leurs travaux sont axés autour de cas d'utilisation. Ici, les cas d'utilisation sont en premier lieu le partage de contenu multimédia entre plusieurs équipements dispersés dans la maison. Pour qu'un équipement soit certifié DLNA, il doit être conforme à d'autres normes ou standards existants : Ethernet et IP, WiFi le cas échéant, et UPnP pour la partie applicative et pour la découverte d'équipements en particulier. Des appareils conformes DLNA existent, mais en nombre et variété encore très restreints.

Le Forum UPnP (*Universal Plug and Play*) [20] définit des protocoles applicatifs pour les appareils communicants de la maison, en se basant sur des protocoles ouverts et bien connus tels que IP, TCP, UDP et HTTP. Les appareils UPnP savent s'auto-décrire et décrire leurs services en diffusant des messages XML sur le réseau local. Des entités nommées points de contrôle interceptent ces descriptions et enregistrent appareils et services dans le but de les mettre en relation. Des protocoles additionnels comme UPnP/AV (Audio/Vidéo) se spécialisent pour certains cas d'utilisation, en l'occurrence pour l'envoi de flux vidéo entre un émetteur (*media server*) et un diffuseur (*media renderer* : télévision, écran, hi-fi). UPnP et ses dérivés sont en passe de devenir les standards les plus utilisés pour créer des réseaux locaux pervasifs.

Jini [21] est une surcouche à Java pour organiser des systèmes distribués, qui sont des fédérations de machines virtuelles Java partageant des services. La notion de service dans Jini est vaste : il peut s'agir d'un équipement physique (imprimante, écran), d'une application ou de données. Pour déclarer son arrivée sur le réseau Jini, un nouveau service publie un proxy sur un annuaire. Les autres services présents peuvent alors le découvrir et l'utiliser via l'objet proxy, qui masque la communication distante. Malgré la richesse des principes de Jini, notamment la variété des services qu'il peut fédérer, son adoption est toujours restée limitée,

partiellement à cause de la démocratisation d'UPnP.

Le projet *Digital Video Broadcasting* (DVB) est également un consortium d'industriels, qui se focalise sur la fourniture de services vidéo. DVB définit le standard DVB-MHP (*Multimedia Home Platform*) [22], qui contient :

- Les protocoles de transport utilisables ;
- Les codecs et les types de compression à utiliser pour les flux vidéo et les images ;
- Le format d'échange pour le contenu non vidéo, en particulier les interfaces graphiques, basé sur HTML ou XML ;
- Un serveur d'applications basé sur Java, qui s'exécute chez l'usager domestique, par exemple sur une *set-top box*. Ce serveur d'applications permet de déployer de nouveaux codecs ou le code gérant de nouveaux formats d'échange.

Une application MHP peut ainsi être démarrée, arrêtée et mise en pause sans redémarrer la *set-top box*. La connectique adoptée par les équipements DVB-MHP est celle définie par HAVi, vue plus haut. Notons enfin que *OpenCable Application Platform* (OCAP) [23], standardisé par CableLabs, reprend et adapte MHP pour les fournisseurs d'accès par câble Nord-Américains.

MExE (*Mobile Execution Environment*) [24] provient du 3GPP, le groupe de standardisation pour les réseaux de téléphonie mobile. MExE définit un environnement d'exécution sur les téléphones mobiles, basé sur Java, dans lequel s'exécutent des services de type transferts multimédia ou interfaces graphiques. Les télécommunications en revanche sont gérées par un système d'exploitation sous-jacent.

Enfin, oBIX (*open Building Information eXchange*) [25] est une proposition récente du consortium OASIS (*Organization for the Advancement of Structured Information Standards*) qui facilite la gestion d'équipements domotiques. Ces équipements sont contrôlés par une passerelle oBIX en lisant et écrivant des valeurs et en routant des alarmes. La passerelle oBIX expose également ces opérations via des services Web, pour faciliter la coopération avec des applications utilisateur. L'implantation de référence d'oBIX est réalisée en Java.

Projets européens Plusieurs projets, regroupant chercheurs et industriels en Europe, participent aux efforts de standardisation sus-cités en apportant leurs propres nouveautés.

Ainsi, le projet européen Amigo [3], qui a débuté en 2004 pour finir en 2007, vise à créer un réseau domestique intégré, ambiant, et facile à utiliser. Il s'agit alors de spécifier les moyens (protocoles, middleware) utilisés par les appareils pour communiquer, en intégrant la notion d'utilisateur.

Le projet MediaNet, dans ses délivrables traitant du réseau domestique, propose des services interactifs sur la télévision, la propagation sans fil des flux audio et vidéo, et adopte l'utilisation de MHP comme serveur d'application.

2.1.3 Jonction des deux mondes : la passerelle domestique

Le réseau domestique et le réseau opérateur sont reliés par la passerelle domestique, aussi appelée passerelle résidentielle. Dans sa forme la plus simple, il s'agit d'un modem, qui assure la traduction des couches 1 et 2 entre les deux réseaux hétérogènes. Dans ses formes plus évoluées, elle opère également sur les couches réseau et transport, par exemple en traduisant des adresses réseau (NAT, *Network Address Translation*).

Vers la maison, la passerelle domestique est reliée aux équipements multimédia, électroménager, domotique, etc. Elle pilote ces équipements en utilisant des technologies des constructeurs, les réseaux domestiques présentés ci-dessus, et des préférences utilisateur.

Vers le réseau distant, la passerelle domestique est reliée au réseau de distribution de l'opérateur, généralement ADSL ou fibre optique. Avec le modèle économique *multi-play* et les réseaux NGN, les services multimédia et données sont contrôlés et garantis depuis le fournisseur jusqu'au CPE (passerelle domestique ou *set-top box*).

Normes L'ITU-T travaille sur les passerelles domestiques par le biais du DHF (*Digital Home-network Forum*). Le groupe de travail commun à l'ISO et à l'IEC s'appelle HES (*Home Electronic System*). DHF et HES ont pour ambition de normaliser les couches d'interconnexion physique, réseau et application sur et autour des passerelles domestiques. Ces travaux sont toujours en cours mais ne sont pas accessibles au public.

Le groupe de travail Megaco [26] de l'IETF, lui, s'intéresse aux passerelles dédiées à la téléphonie sur IP. Les RFC 2805, 3435 et 3525 définissent le protocole asymétrique MGCP (*Media Gateway Control Protocol*), qui est utilisé par des fournisseurs d'accès à Internet aujourd'hui. Il est dit asymétrique car il fonctionne en mode client-serveur, contrairement aux protocoles SIP ou H323 par exemple. Megaco vise à poursuivre les travaux de MGCP.

Standards Le DSL Forum a été créé en 1994 et compte environ 200 membres. En plus des technologies xDSL, il se penche sur les bonnes pratiques de l'auto-configuration, de la fourniture de services et du déploiement en masse. Le DSL Forum produit des rapports techniques, dont notamment le TR-069, aussi appelé CWMP (*CPE WAN Management Protocol*) [27]. TR-069 est une technologie de gestion de passerelles domestiques et de *set-top boxes*, basée sur XML/SOAP et adaptée au monde xDSL. D'autres documents viennent compléter CWMP, comme le TR-098 qui définit le modèle de données associé au TR-069.

HGI (*Home Gateway Initiative*) a été créée en 2004 par des opérateurs européens et japonais, et a été depuis rejointe par des constructeurs européens, américains et chinois. La vision qu'a HGI de la passerelle domestique [28] est celle d'une passerelle dans le sens usuel : elle traduit ou adapte des protocoles entre réseaux hétérogènes. Cette adaptation comprend les notions de qualité de service (QoS) ou encore de sécurité. Sur le plan de la gestion, les équipements et les

services dans la maison sont supervisés depuis un serveur d'auto-configuration (ACS, *Auto-Configuration Server*). La passerelle fait suivre les flux de gestion vers le réseau domestique, ou joue le rôle d'intermédiaire entre les protocoles UPnP, DHCP ou TR-069.

Standards et technologies pour passerelles de services Nous avons vu les normes et des standards définissant des passerelles domestiques avec une vision centrée sur le réseau. D'autres efforts définissent les passerelles domestiques comme des passerelles hébergeant des services, avec une vision centrée sur l'environnement d'exécution de ces services, c'est-à-dire le système d'exploitation ou l'intergiciel.

JES (*Java Embedded Server*) était la proposition de Sun dans cette direction. L'idée était que les domiciles peuvent être connectés à Internet et accéder à des services sans qu'un ordinateur de type PC soit nécessairement requis. JES se voulait donc un serveur d'applications léger, tournant sur la passerelle de services. Le développement de JES a une histoire intimement liée avec la naissance d'OSGi (*Open Service Gateways initiative*), qui partage des objectifs similaires mais qui regroupe plusieurs acteurs économiques, dont Sun. Aujourd'hui, JES est abandonné et a disparu du catalogue de Sun, mais OSGi perdure.

Les spécifications OSGi [29] définissent une plate-forme de services ouverte qui permet de déployer et gérer des services à chaud, c'est-à-dire sans redémarrer la passerelle domestique. Un service OSGi s'exprime sous la forme d'une interface Java, qui symbolise un contrat fonctionnel. Depuis quelques années, la technologie OSGi ne vise plus seulement le monde des passerelles domestiques, mais aussi celui de l'informatique pour automobiles, des serveurs d'application d'entreprise, et plus généralement de tout système à composants. L'Alliance OSGi, fondée en 1999, a donc décidé que le nom OSGi n'est plus un acronyme ; le nouveau mot d'ordre est celui d'« intergiciel universel ».

OpenWings [30] est une technologie aux principes similaires basée sur Jini [31], menée par Motorola et Sun. Par comparaison avec OSGi, OpenWings ciblait initialement les environnements demandeurs de fiabilité, comme les applications militaires, les réseaux d'hôpitaux ou l'automobile. Cependant, la communauté d'OpenWings ne semble plus active depuis 2003 ; ce projet est donc soit arrêté soit entièrement dédié aux applications militaires.

Enfin, l'intergiciel Parlay/OSA présenté plus haut se qualifie également comme plate-forme de services, mais s'exécute sur les équipements des réseaux d'opérateurs de télécommunications, et non sur les passerelles domestiques. Son comportement reste assez proche du principe d'OSGi par exemple, étant donné que OSA découvre automatiquement les nouveaux services disponibles, et exprime ces services sous la forme d'interfaces Java.

2.1.4 Bilan et plan du chapitre

Nous avons vu qu'un ensemble de normes, de standards et de travaux en cours régissent l'écosystème des passerelles domestiques. Certains s'attachent à définir des réseaux opérateur nouvelle génération, avec des caractéristiques comme la qualité de service de bout en bout et la convergence fixe-mobile, le tout sur des réseaux hétérogènes. D'autres avancent vers des réseaux domestiques pervasifs, où les équipements interopèrent aux niveaux physique ou applicatif. Une synthèse des centres d'intérêt est représentée en figure 2.2, tandis que la figure 2.1 fait le bilan des couches de communication en jeu. Ces travaux sont nombreux et pour beaucoup toujours actifs, preuve que la fourniture de services au domicile est un sujet d'actualité, qui nécessite encore d'être travaillée.

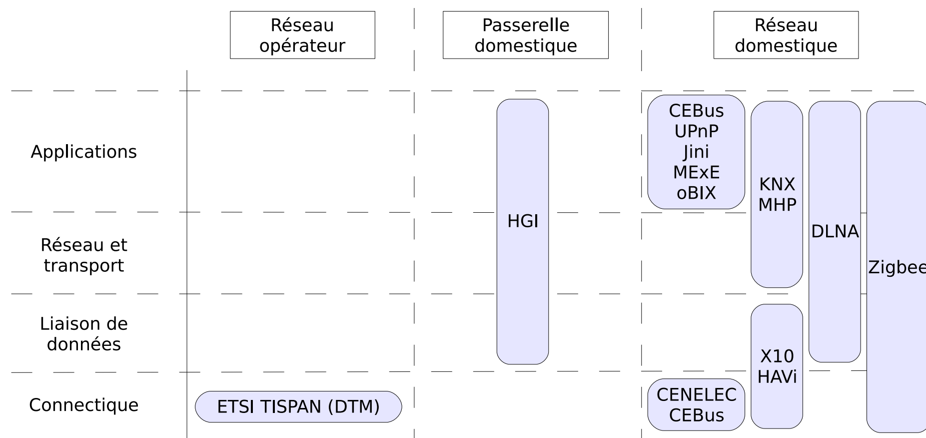


FIG. 2.1: Bilan des normes et standards par couches de communication. Ce tableau vise à montrer que les travaux sur les protocoles de communication, dans et autour de la maison, sont nombreux et variés. Il dénote ainsi un écosystème actif et vivant.

Sur les passerelles domestiques même, la plupart des travaux proposent un comportement de point de jonction entre réseaux hétérogènes, que ce soit au niveau physique, réseau ou protocole de gestion. Là encore, les différents groupes de travail sont très actifs et ont un calendrier chargé pour les années à venir. L'autre aspect des passerelles domestiques, introduit en particulier par le modèle économique *multi-play*, est qu'elles doivent disposer d'un environnement d'exécution ouvert et dynamique permettant de gérer des services.

La suite de ce chapitre se recentre donc sur les passerelles domestiques, et examine plus en détail les aspects gestion et hébergement de services. Nous allons aborder, dans l'ordre :

- Les technologies de gestion connexes aux passerelles domestiques ;
- Les environnements d'exécution pour passerelles domestiques ou adaptables à ce cas d'utilisation ;
- L'impact de ces environnements d'exécution sur le modèle de programmation ;
- Les mécanismes permettant d'isoler les multiples fournisseurs de services sur les passe-

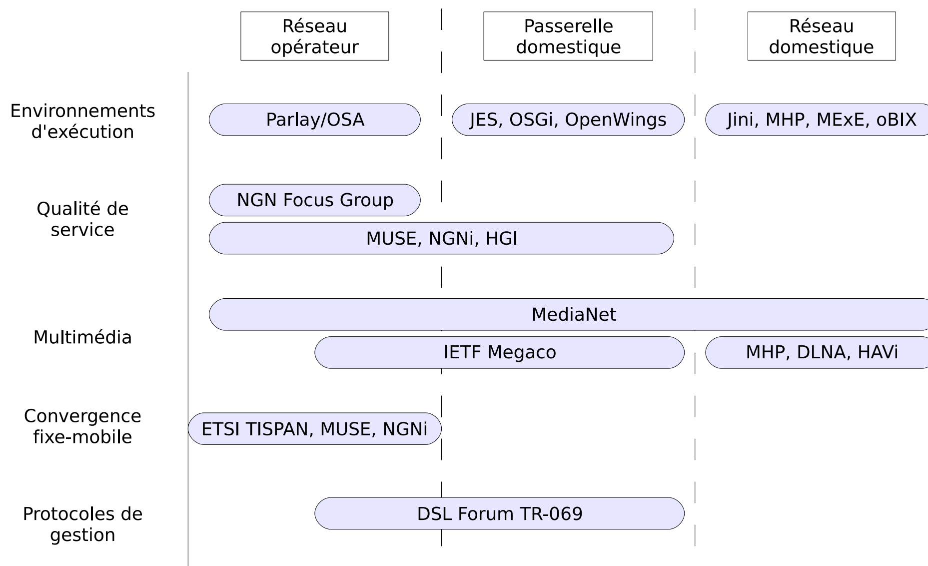


Fig. 2.2: Bilan des normes et standards par centres d'intérêt. Ce tableau regroupe les travaux les plus marquants présentés plus haut. Par la suite, nous allons retenir deux centres d'intérêt (colonne de gauche) que nous focaliserons sur la passerelle domestique : les environnements d'exécution et les infrastructures de gestion. On relèvera que tous les environnements d'exécution représentés sont basés sur Java.

relles domestiques.

2.2 Protocoles de gestion pour passerelles domestiques

Nous avons vu en introduction (figure 1.5 page 3) que les acteurs autour de la passerelle domestique peuvent être regroupés en trois *royaumes* : l'accès (ou connectivité), les services et les usagers au domicile. Ces royaumes délimitent les acteurs économiques en présence, leurs activités liées à la gestion de services, les protocoles et les architectures de gestion mis en œuvre [32].

Ainsi, pour chacun de ces trois royaumes, un ensemble de technologies de gestion est disponible, comme représenté ci-dessous en figure 2.3. Cette section présente les infrastructures et les protocoles qui ont un lien direct ou potentiel avec les passerelles domestiques.

SNMP et CMIP Les protocoles traditionnels de gestion ont été créés en vue de superviser et configurer les équipements réseau tels que les routeurs et les commutateurs. Ils proposent une architecture asymétrique : un superviseur concentre les données de gestion en provenance de multiples équipements supervisés. Le superviseur interroge les équipements du réseau pour obtenir des mesures comme leur débit instantané ou leur taux de retransmission ; il écoute également d'éventuelles alarmes levées par les équipements supervisés. Le but du superviseur

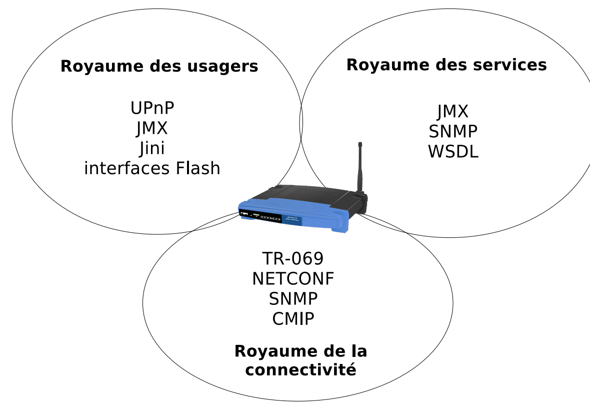


Fig. 2.3: Royaumes de gestion et technologies associées

est de concentrer et présenter ces informations à la personne en charge de l'administration, d'automatiser leur traitement et de proposer des mesures correctrices en cas de défaillance.

Les équipements supervisés disposent d'un logiciel dit agent, qui dialogue avec le superviseur. L'agent puise les données de gestion à remonter dans une base de données de gestion locale (MIB, *Management Information Base*). Cette MIB elle-même est peuplée par des sondes, qui relèvent l'information demandée à leur source, par exemple le pilote de la carte réseau ou le système d'exploitation.

SNMP [33] de l'IETF est utilisé principalement sur des réseaux locaux ou métropolitains. La MIB SNMP est structurée sous forme d'arbre, chaque noeud définissant un sous-espace de nommage unique. Elle peut ainsi être utilisée pour gérer des services applicatifs, à condition que l'éditeur logiciel des services l'étende. Le format d'échange entre superviseur et agent est ASN.1, une syntaxe de description de données indépendante du protocole de transport sous-jacent.

CMIP [5] est une autre technologie de gestion basée sur des principes similaires. CMIP n'est principalement utilisé que dans des réseaux de télécommunications, du fait de sa complexité. En effet, CMIP sur TCP/IP est composé de 4 protocoles additionnels, décrits dans 7 documents différents. L'avantage de CMIP sur SNMP est en particulier la représentation de sa MIB sous forme d'objets, encapsulant chacun ses propres attributs, comportements et notifications.

TR-069 et Netconf TR-069, présenté dans la section précédente, a été créé par le DSL Forum spécialement pour les équipements de type passerelles domestiques. Netconf (RFC 4741) [34], de son côté, a été créé pour gérer des équipements réseau.

Ces deux technologies partagent des ressemblances techniques, et en particulier leur format d'échange de messages basé sur SOAP/XML, plus expressif qu'ASN.1. Tout comme SNMP, ces protocoles sont asymétriques. Dans le cas de TR-069 par exemple, l'équipement

supervisé (CPE) entame toute transaction de gestion en se déclarant auprès de son serveur d'auto-configuration (ACS). Les avantages d'utiliser TR-069 plutôt que SNMP sont un jeu de paramètres (la MIB) dédié aux passerelles domestiques, des appels de méthodes (*reboot*, *download*) adaptés au cas d'utilisation, et l'intégration de la sécurité à plusieurs niveaux : transport des données (via SSL/TLS), et secrets partagés au niveau applicatif. Netconf, en plus, propose un mécanisme de session et de *rollback* : l'ACS peut modifier les valeurs de tout un groupe de paramètres d'un coup ; en cas d'erreur, tout le groupe de paramètres garde les valeurs initiales.

WSDM *Web Services Distributed Management* est le standard du consortium OASIS pour utiliser des services Web comme plate-forme de gestion. Il est composé de deux sous-parties. *Management Using Web Services* (MUWS) [35] sert à gérer des équipements divers via des services Web. *Management Of Web Services* (MOWS) [36] sert à gérer des services Web via des services Web. Dans WDSM, le contenu des MIB est directement représenté comme des *ressources* dans des services Web. Ceci permet d'abstraire les notions d'équipements physiques, d'agents de gestion ou de localisation géographique et de les noyer derrière les mécanismes de découverte de services Web.

JMX *Java Management eXtensions* [37] est une solution de gestion spécifique à Java. La MIB est représentée sous la forme d'interfaces Java, appelées MBeans, sur lesquelles le superviseur appelle des méthodes. Les MBeans sont implantés par des sondes, qui peuvent donc indifféremment fournir des informations sur l'état physique du système, sur le système d'exploitation, sur l'état de la machine virtuelle Java, ou encore sur les applications elles-mêmes. Du point de vue architectural, JMX ajoute une nouvelle notion à la distinction entre superviseur et supervisé. En effet, contrairement à SNMP où le protocole de communication est figé, JMX introduit la notion de connecteur. Situé sur l'équipement supervisé, l'objectif du connecteur est d'assurer la traduction entre le protocole choisi par le superviseur et l'agent JMX localement présent. On trouve ainsi des connecteurs pour RMI [38], HTTP et services Web (via WS-Management du DMTF). Il est donc techniquement possible d'implanter un connecteur JMX pour TR-069 par exemple.

UPnP UPnP a été présenté en section 2.1.2 comme un protocole de communication entre équipements dans la maison basé sur XML/HTTP. Les équipements UPnP pouvant diffuser et découvrir les services de leur choix, ces services peuvent être des interfaces de gestion. UPnP est ainsi utilisable pour gérer des appareils électroménagers [39], depuis un PC ou depuis un téléphone mobile [40].

2.3 Environnements d'exécution existants liés aux passerelles domestiques

Les modems et les premières générations de passerelles domestiques étaient équipés de systèmes d'exploitation propriétaires pour assurer les fonctions de codage et de transport réseau (couches 1–4). Avec le modèle économique *triple play*, les constructeurs ont migré vers des solutions plus standardisées [41] ; les passerelles domestiques d'aujourd'hui sont équipées d'un système GNU/Linux. L'intérêt de cette migration est de pouvoir puiser dans le jeu d'applications déjà existantes, de réutiliser leurs configurations, d'avoir des outils déjà longuement testés, et de trouver des développeurs rapidement.

D'autres technologies émergentes peuvent jouer le rôle d'environnement d'exécution multi-services, en particulier Java/OSGi. Cette section présente les objectifs de ces solutions existantes, ainsi que leurs avantages pour le modèle économique multi-services.

2.3.1 GNU/Linux

Présentation GNU/Linux remplace peu à peu les systèmes d'exploitation propriétaires des constructeurs sur nos passerelles domestiques. C'est le cas par exemple pour les Freebox et autres Livebox en France. En effet, ce système d'exploitation supporte une large gamme d'architectures processeur et de pilotes matériels ; les constructeurs peuvent donc réutiliser le même système d'exploitation sur plusieurs générations de passerelles domestiques. Le coût de maintenance s'en trouve réduit. De même, la pile réseau, les fonctionnalités de filtrage et de pare-feu sont déjà implantées. Elles ont été testées par de nombreux développeurs, sur divers environnements, et sont en principe moins sujettes à des erreurs de programmation, comme exprimé par l'adage « *many eyes make all bugs shallow* » [42].

GNU/Linux s'obtient sous forme de *distribution* : des développeurs choisissent un système d'exploitation (noyau Linux et des outils GNU), un jeu d'applications et de librairies ainsi que la documentation, qu'ils fournissent sur un support quelconque (CD-ROM, téléchargement). Les développeurs s'assurent que l'ensemble est cohérent, en particulier via un choix judicieux des numéros de version des logiciels.

Paquetages Les applications GNU/Linux sont généralement livrées sous forme de *paquetages*. Il s'agit d'un fichier archive qui contient le code et les données de l'application ainsi que des méta-données. Suivant ce que supporte le gestionnaire de paquetages utilisé, les méta-données peuvent contenir notamment le numéro de version de l'application, les autres paquetages dont il dépend, sa maturité (version stable ou version en cours de développement), ainsi que des commandes qui seront exécutées automatiquement lors de l'installation ou de la désinstallation du paquetage.

Les gestionnaires de paquetages des distributions modernes gèrent automatiquement les dépendances, c'est-à-dire qu'à l'installation d'une application, tous les paquetages nécessaires à son fonctionnement sont automatiquement installés également.

Dépôts de paquetages Les distributions GNU/Linux comptent plusieurs milliers de paquetages. Ceux-ci proviennent de développeurs différents, et sont disponibles sous des conditions très variées : serveur d'hébergement personnel ou dédié, archive pré-empaquetée ou code source depuis le gestionnaire de configuration (SCM), lois du pays d'hébergement, etc. Pour simplifier le problème et garantir un accès stable et permanent à l'ensemble des paquetages, les distributions préparent les paquetages et les déposent sur des serveurs répliqués. Les gestionnaires de paquetage s'adressent à ces serveurs pour obtenir une liste versionnée des paquetages existants.

Services ou scripts rc Les paquetages sont les unités d'installation et de désinstallation pour les applications et les librairies du système. De la même manière, il existe des unités de lancement et d'arrêt pour les applications, appelés scripts `rc` ou plus simplement services. On les trouve souvent dans `/etc/init.d/` ou `/etc/rc.d/`, selon la distribution. Les scripts `rc` prennent en paramètre des commandes comme `start` et `stop`, et mettent en œuvre le démarrage et l'arrêt des services en prenant garde de positionner des variables d'environnement ou de vérifier l'intégrité de l'application. Dans certaines distributions, les scripts `rc` renseignent des dépendances sur d'autres scripts `rc`, résolues à l'invocation de `start` ou `stop`. Par exemple, le service de journalisation `syslog` a besoin d'une horloge système fonctionnelle, représentée par le service `clock`. Si l'administrateur démarre `syslog`, `clock` devrait démarrer automatiquement. Les services GNU/Linux sont souvent des processus démon (serveur Web, serveur SSH).

Multi-applications et multi-utilisateurs Un autre avantage des systèmes GNU/Linux est de supporter l'exécution pseudo-parallèle de plusieurs applications, ce qui est nécessaire pour une passerelle domestique multi-services. Les applications s'exécutent sous couvert d'un compte *utilisateur*, qui est une abstraction pour des mécanismes d'authentification, de droits d'accès et d'utilisation des ressources physiques. Ainsi, si chaque fournisseur de services dispose de son propre compte utilisateur, les applications des uns n'auront qu'un accès limité à celles des autres. Sur le système de fichiers en particulier, les risques de conflits d'accès s'en trouvent réduits.

2.3.2 Java/OSGi

Présentation OSGi a été présenté plus haut comme un environnement d'exécution initialement créé pour les passerelles domestiques. Il s'agit d'une surcouche à la machine virtuelle Java (JVM) qui facilite le déploiement et la gestion d'applications dans une même instance de JVM.

Bundles Les applications se présentent sous la forme de *bundles*, c'est-à-dire d'archives `jar` contenant des classes Java, du code natif, des fichiers divers dits de ressources et des méta-données. En cela, un *bundle* est similaire à un paquetage dans GNU/Linux : il peut être installé et désinstallé, et la plate-forme OSGi vérifie que les dépendances nécessaires sont présentes. Cependant, les méta-données d'un *bundle* font apparaître une classe particulière dans l'archive : l'activateur. Celui-ci joue le même rôle que les scripts `rc` dans GNU/Linux, en ce qu'il contient le code à exécuter pour démarrer et arrêter l'application. Le *bundle* est donc une unité cohérente qui sert à la fois aux étapes d'installation/désinstallation et de démarrage/arrêt.

Un bundle correspond généralement à une application ou à un sous-ensemble d'une application (l'équivalent d'une module en langage C). Ce découpage facilite et impose la décomposition d'une application en unités logiques les plus autonomes possibles.

Déploiement Pour installer un *bundle*, il faut le rapatrier localement depuis une URL, par exemple depuis le Web (schéma `http://`) ou depuis le système de fichier (schéma `file://`). Suivant le même principe que les dépôts de paquetages GNU/Linux, un dépôt de *bundles* tel que spécifié par OBR (*OSGi Bundle Repository*) [43] peut faciliter cette étape. Si des dépendances sont absentes de l'environnement OSGi local, OBR les rapatrie automatiquement, tandis qu'OSGi sans OBR indique une erreur.

Une fois le rapatriement effectué, un *bundle* peut être démarré, arrêté et mis à jour depuis une URL, sans redémarrer la plate-forme OSGi. La figure 2.4 décrit plus précisément ce cycle de vie.

Dépendances Un *bundle* peut dépendre d'autres ressources logicielles pour s'exécuter :

- un *package* Java particulier, contenant des classes Java et/ou du code natif ;
- un *bundle* particulier, par exemple contenant des fichiers son ou image, qui n'appartiennent pas à un *package* Java.

Ceci s'exprime par des paires clé-valeur dans les méta-données. Un *bundle* importe les ressources dont il a besoin (`Import-Package`, `Require-Bundle`) et il exporte les *packages* Java qu'il veut rendre accessibles aux autres *bundles* (`Export-Package`). Lorsqu'un *bundle* est installé ou mis à jour, la plate-forme OSGi résout ses dépendances, c'est-à-dire qu'elle

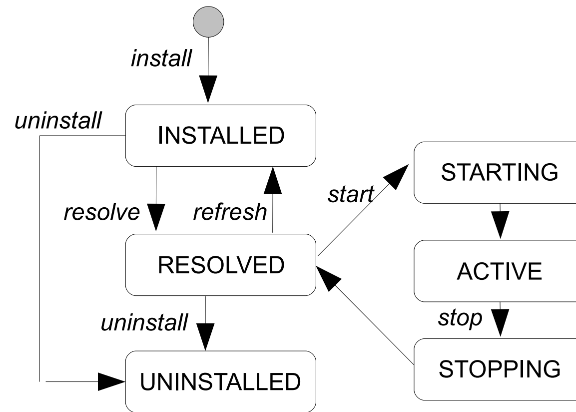


Fig. 2.4: Cycle de vie d'un *bundle* OSGi

vérifie que les ressources demandées sont présentes sur le système.

Si toutes les dépendances du *bundle* nouvellement installé ou mis à jour sont résolues, celui-ci passe à l'état RESOLVED.

Chargement de classes Une classe Java est une suite d'octets lus depuis un médium quelconque (disque local, réseau) et transformée en une instance de la classe `java.lang.Class`. Trouver la suite d'octets, la transformer et la charger en mémoire est le rôle des chargeurs de classes (*classloaders*). Il en existe plusieurs, hiérarchisés sous forme d'arbre. Au sommet de l'arbre, le chargeur de classes de démarrage (*bootstrap class loader*) charge les classes standard Java, comme `java.*`, depuis le répertoire d'installation de la JVM. Pour l'implantation de Sun, il s'agit du fichier `jre/lib/rt.jar` ; dans GNU Classpath, ce fichier s'appelle `glibj.zip`. Ce chargeur de classes a un fils : le chargeur de classes d'extensions, qui charge les classes implantées pour un système d'exploitation ou une cible matérielle particuliers (`jre/lib/ext/*.jar` pour les JVM de Sun). Enfin, le chargeur de classes d'extensions a lui-même un fils, qui est le chargeur de classes applicatif. Son rôle est de charger les classes présentes dans le *classpath* tel que passé en paramètre au lancement de la machine virtuelle Java. Une application Java peut exécuter des instructions du type `new ClassLoader()`, en utilisant une implantation existante (`SecureClassLoader`, `URLClassLoader`) ou une implantation spécifique ; ceci donne naissance à la structure en arbre.

La structure hiérarchique des chargeurs de classe joue un rôle lorsqu'il faut trouver une classe, avant de pouvoir la charger. En effet, le chargeur de classes demande à son parent de trouver et charger la classe demandée ; on dit qu'il délègue. La délégation est récursive : le chargeur de classes parent interroge son propre parent, et ainsi de suite jusqu'à remonter au chargeur de classes principal, dit de *bootstrap*. Si le parent répond par la négative, le chargeur de classes courant tente lui-même de charger la classe. S'il échoue également, on obtient une `ClassNotFoundException`. Notons que la délégation du chargement de classes apporte

plusieurs avantages, le premier étant un élément de sécurité. En effet, on garantit que les classes standard comme `java.net.Socket` sont toujours chargées depuis le `rt.jar` installé par l'administrateur système ; une application ne doit pas pouvoir remplacer l'implantation standard connue par une implantation non sûre. Le deuxième avantage est de pouvoir décharger (ou supprimer) des classes, et par conséquent de pouvoir les mettre à jour. Lorsque l'on décharge une classe, toutes les autres classes ayant une référence sur elle (par exemple ayant un attribut de ce type) doivent être déchargées également, sous peine de subir des erreurs à l'exécution. Trouver toutes ces références parmi toutes les classes chargées est fastidieux. La solution des chargeurs de classes permet de confiner l'espace des classes visibles, dans le sens où un chargeur de classes permet de voir les classes connue de son parent, mais pas de ses fils, ni de ses frères. Ainsi, une classe chargée par le chargeur de classes de démarrage ne peut pas avoir de référence sur une classe chargée par le chargeur de classes applicatif. Il est donc possible de décharger une classe du chargeur de classes applicatif sans perturber celles du chargeur de classes de démarrage. Pour éviter la recherche fastidieuse de toutes les références existantes, décharger une classe revient à décharger son chargeur de classes ainsi que tous ses fils.

Isolation entre bundles OSGi tire grandement parti du mécanisme de chargement de classes. En effet, chaque *bundle* « résolu » dispose de son propre chargeur de classes. Ceci confine la visibilité des classes contenue dans un *bundle* à ce *bundle* même, et permet de mettre à jour une unité de déploiement sans impacter les autres et sans nécessiter de redémarrage. Le seul moyen de franchir ce confinement est par le biais du mécanisme explicite d'import/export.

Quand un *bundle* nouvellement installé importe un *package* Java, le *package* est recherché parmi les exports connus. S'il est trouvé, il est ajouté (avec son chargeur de classes) à une liste de liens (*wires*). Si tous les imports sont satisfaits, le *bundle* passe à l'état RESOLVED et un chargeur de classes est créé, prenant en paramètre la liste de liens.

Les chargeurs de classes OSGi diffèrent donc des chargeurs de classes standard car ils acceptent cette liste de liens vers des classes venant de chargeurs de classes frères, cassant alors l'isolation entre chargeurs de classes sur demande explicite.

La procédure pour mettre à jour un *bundle* est de l'arrêter, d'arrêter ses dépendances, de supprimer les chargeurs de classes associés, de changer l'ancienne version du *bundle* par la nouvelle, puis de le redémarrer avec ses dépendances.

Services En plus des *bundles* et des *packages*, OSGi propose une troisième couche : les services. Un service OSGi est une interface Java, fournie par un *bundle* à sa discrétion lorsqu'il est démarré, et enregistrée auprès d'un annuaire. Les spécifications OSGi définissent un ensemble de services standards, comme par exemple :

- un service HTTP (serveur Web) ;

- un service de journalisation (*Log Service*) ;
- un service de gestion des permissions (*Permission Admin*) pour implanter des politiques de sécurité ;
- des gestionnaires d'URL, pour prendre en charge des schémas URL tels que les bien connus `ftp://` et `http://`, mais aussi des schémas personnalisés comme `p2p://` [44].

Un service OSGi peut être implanté de plusieurs manières, et peut disposer de plusieurs implantations au même instant. Lorsqu'un service particulier est demandé, l'annuaire renvoie la liste des implantations disponibles. Le *bundle* demandeur choisit alors une implantation selon des critères exprimés par un filtre LDAP.

Ainsi, si les dépendances entre *bundles* permettent de partager des classes Java, les services OSGi permettent de partager des objets. Le modèle de programmation associé aux services OSGi est présenté plus loin en section 2.4.

2.3.3 Environnements d'exécution connexes

2.3.3.1 .NET CLR

Présentation .NET est la réponse de Microsoft à Java de Sun. CLR (*Common Language Runtime*) est une machine virtuelle exécutant le code compilé .NET, tout comme la JVM exécute le *bytecode* Java. La différence principale est que le code compilé pour CLR, et représenté en langage MSIL (*Microsoft Intermediate Language*), peut provenir de plusieurs langages de programmation : C#, J#, ASP.NET, mais aussi Haskell, PHP, Python ou même Ada, pourvu que les compilateurs nécessaires soient disponibles. Notons que Java dispose maintenant d'interpréteurs pour Ruby ou encore Python, et donc que cette différence s'estompe.

Assemblies et AppDomains Les unités de déploiement dans .NET partagent également des ressemblances avec celles de Java/OSGi. Une *assembly* .NET est un ensemble de code MSIL, de ressources et de méta-données. Sa granularité est proche d'un *bundle* OSGi, mais son fonctionnement est plus proche d'une classe Java : on peut charger une *assembly* en mémoire, mais on ne peut pas la décharger directement.

Un *AppDomain* est un espace de nommage dont le rôle est de charger les *assembly* (on parle aussi d'*Add-in*). Sa granularité est celle d'un groupe de *bundles*, mais son fonctionnement est plus proche de celui d'un *bundle* : on ne peut pas décharger une *assembly*, mais on peut décharger un *AppDomain*.

Dans .NET, un processus a un *AppDomain* par défaut, qui ne peut pas être déchargé. Pour comparaison, en Java, il n'est pas possible de mettre à jour les classes standard Java que l'on trouve dans le `rt.jar`, à moins d'utiliser une implantation de JVM basée sur OSGi [45] ; il est cependant possible de mettre à jour la plate-forme OSGi durant son exécution. L'*AppDomain*

par défaut peut créer d'autres *AppDomain* dans son processus, qui eux pourront être déchargés et mis à jour.

Coopération et communication Les *AppDomains*, qu'ils soient sur des machines différentes ou dans le même processus, utilisent des espaces d'adressage différents. Par conséquent, un communication entre différents *AppDomains* passe par une « frontière » (*boundary*). Franchir cette frontière revient à réaliser un appel de fonction distant : l'appel ainsi que ses paramètres subissent une opération de *marshalling*, où les objets ou données sont transformées en un train d'octets (on parle de sérialisation). Alternativement, la communication entre *AppDomains* peut se faire via un service Web, mais le même problème de sérialisation des paramètres se pose. Ces opérations de sérialisation et de désérialisation ont un coût de traitement non négligeable, que les applications .NET vont payer à chaque communication inter-*AppDomain*.

Par opposition, OSGi définit des interactions locales entre *bundles*, qui se font par appel direct à une méthode d'un service OSGi. Les interactions distantes ne sont pas spécifiées, mais des technologies dédiées à cela peuvent être utilisées : RMI pour les appels distants, DOSGi [46] (anciennement M-OSGi) ou R-OSGi [47] pour transformer des plates-formes OSGi en systèmes distribués. Les communications via services OSGi ont donc l'avantage de ne pas faire payer le coût de la sérialisation.

.NET et passerelles domestiques Pour réaliser une passerelle domestique multi-services, .NET a l'avantage d'être multi-langages et de supporter la mise à jour dynamique de code. Cependant, la granularité de ces mises à jour est mal positionnée. Si un *AppDomain* correspond à une application d'un fournisseur, alors les communications entre applications du même fournisseur doivent être limitées, sous peine de dégradation des performances. Si un *AppDomain* correspond à un fournisseur de services, alors la mise à jour d'une application implique l'arrêt et le redémarrage de toutes les applications de ce fournisseur.

En résumé, la plupart des avantages de .NET pour les passerelles domestiques se retrouvent aussi dans OSGi, mais les facilités de mise à jour, la coopération entre unités de déploiement et la gestion du cycle de vie d'OSGi semblent mieux adaptées au contexte multi-services.

2.3.3.2 Java ME

présentation Java ME (*Micro Edition*) est la version de Java dédiée aux petits équipements. Java ME se dérive en deux « configurations » : CDC (*Connected Device Configuration*) et CLDC (*Connected Limited Device Configuration*). CDC est plutôt dédié aux assistants personnels (PDA) et aux *set-top boxes*, tandis que CLDC vise équipements plus petits du type téléphones mobiles. Pour cela, CLDC se base sur la KVM (*K Virtual Machine*), dont la conception diffère des spécifications JVM de Sun, et dont l'objectif est de tenir en quelques

centaines de Ko de mémoire.

Une *configuration* Java ME est complétée par un *profil*, qui contient le jeu de classes Java disponible. Des *packages* optionnels peuvent également être ajoutés. CDC propose trois profils :

- Le profil de base *Foundation*, qui fournit une partie des *packages* de Java SE, sans interface graphique. Les cibles visées sont les imprimantes réseau, les routeurs ou encore les passerelles domestiques ;
- Le profil *Personal Basis*, qui y ajoute une partie des classes AWT pour interfaces graphiques. Sun cite comme exemple des applications pour automobiles ;
- Le profil *Personal*, qui supporte la totalité d'AWT et les applets. CDC/PP est recommandé pour des navigateurs Web embarqués et des PDA.

En revanche, sur la configuration CLDC, le profil le plus répandu est MIDP (*Mobile Information Device Profile*). On notera le support d'interfaces graphiques minimales, de bases de données persistantes légères, ainsi que des applications sous forme de MIDlets, présentées ci-dessous. Un second profil, IMP (*Information Module Profile*), est destiné aux machines sans écran. Á l'exception du support graphique, IMP est semblable à MIDP.

Java ME et passerelles domestiques Sun présente les environnements CLDC/MIDP et CDC/Foundation comme adaptés aux passerelles domestiques. L'avantage notable de CLDC/-MIDP est la notion de MIDlet. « MID » symbolise les équipements mobiles (*mobile information devices*). Une MIDlet est une application MIDP contrôlée par un gestionnaire d'application. Le cycle de vie d'une MIDlet est composé de trois états : active, en pause, et détruite. Le changement d'état de la MIDlet est commandé soit par le gestionnaire d'applications (`startApp()`, `pauseApp()`, `destroyApp()`), soit par la MIDlet elle-même (`resumeRequest()`, `notifyPaused()`, `notifyDestroyed`). En revanche, la configuration CLDC est exempte de chargeurs de classes. Il n'est donc pas prévu de décharger des classes Java ou de les mettre à jour.

De son côté, CDC/Foundation est plus proche de Java SE, et supporte les mécanismes de chargement/déchargement de classes. L'équivalent CDC d'une MIDlet CLDC s'appelle une Xlet, et est également contrôlé par un gestionnaire d'applications. Le cycle de vie d'une Xlet (figure 2.5) est composé de trois états : chargée, en pause, démarrée et détruite. En comparaison avec les MIDlets, l'état supplémentaire « chargée » est dû au support des chargeurs de classes ; ceci impose l'appel supplémentaire `initXlet()`. En dehors de cet appel, le changement d'état de la Xlet se fait comme précédemment. CDC/Foundation demande plus de ressources matérielles que CLDC ; en particulier, il faut 2 Mo de mémoire au minimum, ce que les passerelles domestiques d'aujourd'hui satisfont.

Les Xlets sont utilisées par les fournisseurs de télévision numérique, par exemple dans le

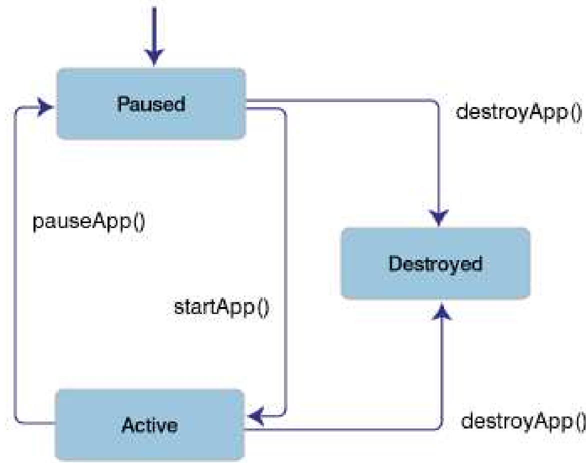


Fig. 2.5: Cycle de vie d'une Xlet dans Java ME

standard *Multimedia Home Platform* présenté plus haut en section 2.1.2.

Java ME et OSGi Nous avons vu que les différentes versions de Java ME disposent d'un gestionnaire d'applications, dont le but est de gérer le cycle de vie des éléments logiciels installés. En revanche, les applications Java ME ne savent pas exprimer leurs dépendances comme savent le faire les *bundles* OSGi. L'environnement Java ME ne résout donc pas les dépendances applicatives ; il ne permet pas de découper facilement applications et librairies en modules. De plus, il manque à Java ME la couche de *services* pour faciliter la coopération entre modules logiciels.

On peut donc noter que le modèle Java ME fonctionne de manière satisfaisante sur un petit équipement contrôlé par une seule entité économique. Par contre, sur une passerelle domestique multi-services où plusieurs fournisseurs installent et gèrent des applications, Java ME se révèle moins adapté qu'OSGi.

Pour combler ces lacunes, il est possible d'exécuter OSGi au-dessus de Java ME. D'après les spécifications OSGi, un environnement CDC/Foundation permet d'exécuter une plate-forme OSGi minimale [48], surnommée « profil OSGi/Minimum ». L'environnement CLDC n'est par contre pas suffisant, principalement parce qu'il lui manque le mécanisme de chargeurs de classes, crucial pour le cycle de vie des *bundles*. Le groupe de travail *Mobile Expert Group*, mandaté par l'alliance OSGi, travaille cependant à adapter une version minimaliste d'OSGi pour CLDC/MIDP.

2.4 Modèles de programmation

Dans les sections précédentes, nous avons vu deux grandes familles d'environnements d'exécution adaptés ou adaptables aux passerelles domestiques : les systèmes d'exploitation et les intergiciels. Leurs principaux représentants sont respectivement GNU/Linux et Java/OSGi. Ces environnements proposent à leur administrateur des « prises » ou outils leur permettant de contrôler et de gérer les applications hébergées. Nous décrivons ci-dessous ce que ces environnements proposent aux développeurs, en terme de paradigmes de programmation, de décomposition d'applications en modules et de bonnes pratiques en général.

2.4.1 Langages natifs pour GNU/Linux

Un système d'exploitation GNU/Linux n'impose pas de limite quant aux langages de programmation employés. Cependant, à l'usage, la plupart des applications sont écrites en C ou en langage interprété de type shell. Nous allons donc nous restreindre aux paradigmes de programmation associés.

Le modèle de programmation usuel pour le noyau Linux et les applications GNU est impératif et procédural. Une application C peut être conçue sous forme de modules, mais donnera lieu à un seul exécutable après les phases de compilation et d'édition des liens. Lors de l'exécution, des applications peuvent communiquer soit par le système de fichiers, soit par des communications inter-processus (IPC), soit par des *sockets*. Ces communications consistent en des échanges de données entre modules ou applications. Pour qu'un module puisse appeler directement une fonction partagée par un autre module sans recopier ce dernier dans l'application, il faut utiliser des librairies liées dynamiquement (fichiers `.so`). L'utilisation de librairies partagées permet la réutilisation de code, qui fait partie des bonnes pratiques de l'ingénierie logicielle.

L'interface de programmation (API) offerte aux applications natives est à quelques détails d'implantation près celle de POSIX [49]. Pour les applications C, l'API POSIX est implantée par la librairie C (`glibc` ou autre).

2.4.2 Objets, composants, services

La plupart des intergiciels sont associés à un paradigme de programmation plus avancé que la programmation procédurale. Bien sûr, cette relation n'est ni universelle ni réciproque, mais elle est une simplification que nous utiliserons pour mettre en relief l'information contenue dans cette section. En particulier, les environnements Java/OSGi et .NET CLR sont de bons représentants des paradigmes de programmation composants et services.

Objets Par opposition à la programmation procédurale, la programmation orientée objet vise à représenter un modèle mental du monde réel. Un objet représente donc une personne, une pièce de moteur ou encore une entité administrative. Il a un état interne, représenté par ses *attributs*, et sait réaliser des actions, représentées par ses *méthodes* ; on parle d'*encapsulation*. Les attributs correspondent à des noms (nom, âge d'une personne), et les méthodes à des verbes (manger, dormir). Un objet est typé : il dérive d'une classe ou d'un prototype, selon les langages.

Les objets interagissent via des références : si l'objet A a une référence sur l'objet B, il peut demander à B d'exécuter une de ses méthodes.

Composants Une autre ambition des objets était la réutilisation de code. Pour composer une application, on peut en théorie grouper des types d'objets déjà existants dans d'autres projets, et ne développer que la logique qui lie les objets (comme le fil d'exécution principal). En pratique, cette réutilisation pose cependant plusieurs problèmes :

- Réutiliser un type nécessite de réutiliser ses dépendances, c'est-à-dire les types que lui-même utilise. Il est difficile de prévoir et borner le nombre et la taille des dépendances ramenées, ce qui risque de rendre l'application complexe ;
- Un même type d'objet aura différents objectifs fonctionnels selon l'application visée. Par exemple, une « Personne » dans une application bancaire aura un numéro de compte, un solde ou encore un revenu moyen, mais ces informations sont inutiles dans un simulateur de tennis. Il y a une différence de sémantique entre ces Personnes, que le concepteur de l'application connaît mais que le langage ne permet pas d'exprimer ;
- La logique liant les objets est mêlée aux objets eux-mêmes. Les références vers d'autres objets sont parsemées dans le code, et pointent vers des implantations particulières, comme vu dans les points ci-dessus. La notion d'interface permet d'abstraire partiellement l'implantation du type référencé, mais elle n'est pas utilisée partout car le modèle de programmation ne l'impose pas.

La programmation orientée composant propose de combler ces déficits. Un composant dispose d'une implantation, qui ne communique avec l'extérieur que par des interfaces. L'implantation interne du composant peut être un ou plusieurs objets, ou encore un sous-composant, selon les modèles. Les interfaces sont soit fonctionnelles, c'est-à-dire les entrées/sorties vers d'autres composants, soit non fonctionnelles, c'est-à-dire des prises pour que l'environnement gère des caractéristiques non métier comme la persistance, le cycle de vie ou la sécurité. Ceci permet de séparer le « contrat » du composant, ou ce qu'il déclare savoir faire, de la manière dont il est réalisé.

De plus, en imposant que toutes les interactions inter-composants passent par des interfaces, on explicite les dépendances entre composants. En effet, ce paradigme suggère très fortement

aux concepteurs et développeurs de découper les applications en modules les plus autonomes possible. L'intérêt est double : d'une part, il devient facile de réutiliser un composant dans une nouvelle application. D'autre part, lorsqu'une application est modifiée, mise à jour, corrigée, la portée des modifications est limitée. On limite alors le risque de régressions et le nombre de tests unitaires à lancer pour valider la modification.

Selon les modèles, un composant peut être vu à la fois comme une unité d'installation et une unité d'exécution. Les exemples les plus courants de plates-formes à composants sont *CORBA Component Model* (CCM) [50], COM [51] de Microsoft, J2EE/EJB [52] de Sun, les implantations du modèle Fractal [53] d'ObjectWeb telles que Think [54] ou Julia [55], et le niveau *bundle* dans OSGi. Ils diffèrent par la simplicité de leurs interfaces de programmation, par la souplesse du modèle et par les possibilités de gestion offertes par l'environnement d'exécution.

Services La programmation orientée service insiste sur la notion de contrat (ou service) en tant qu'interface. En cours d'exécution, des services sont enregistrés et désenregistrés auprès de l'environnement d'exécution. Inversement, l'environnement d'exécution est interrogé pour savoir si un service est disponible. Cet échange peut se faire sous la forme d'événements, sur le modèle des écouteurs (*listeners*) ou du tableau blanc (*whiteboard pattern* [56]). Un service est donc une interface (et son implantation), accessible par intermittence durant le cycle de vie de l'environnement d'exécution. Cette définition est celle adoptée par OpenWings [57] et par OSGi via les services déclaratifs [58].

D'autres propositions indiquent quelle implantation de service utiliser non pas dynamiquement, mais dans une phase préalable, dans un fichier de configuration. C'est le cas de iPOJO [59] pour OSGI, de Spring [60] et de Pico Container [61].

Un service est donc une entité d'exécution, et non de déploiement comme un composant. Ces deux entités peuvent être complémentaires, comme c'est le cas dans OSGi avec la couche *bundle* et la couche service. On parle parfois de modèles à composants à services. Alternativement, Spring intègre la notion de service mais pas de composant, car la notion de déploiement à chaud d'applications est ignorée.

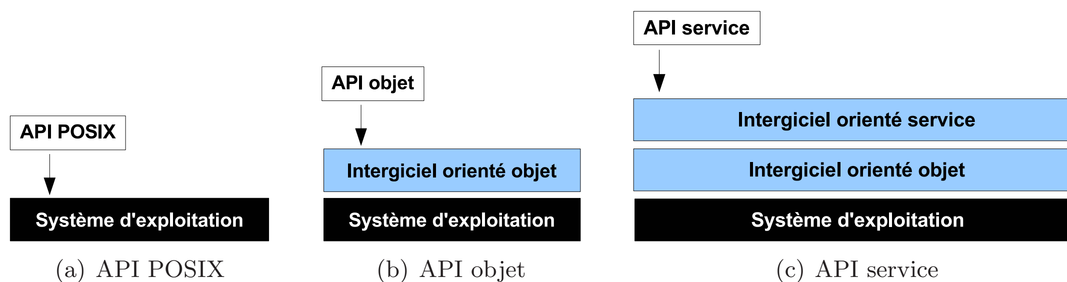
Programmation orientée service et architectures orientées service La notion de service diffère beaucoup selon les domaines et les personnes ; ceci cause une certaine confusion dans des termes de plus en plus usités aujourd'hui. En effet, il nous faut distinguer la programmation orientée service (SOP) des architectures orientées service (SOA). SOP et SOA s'attaquent toutes deux au problème de la coopération entre différents modules logiciels. Cependant, la SOP a une approche programmatique, tandis que les SOA ont une approche réseau.

Le but ultime de la SOP est trouver une adresse mémoire dans son propre espace d'adressage. Dans OSGi, les *bundles* peuvent coopérer en partageant des *packages* Java ou en utilisant

des services OSGi ; dans les deux cas, le client demandeur obtient une référence directe sur la classe ou l'objet demandé. La SOP ne résout qu'un problème de références locales à un seul environnement d'exécution. Notons qu'à la fois le client et le fournisseur d'un service OSGi doivent être écrits dans le langage Java.

Par opposition, l'objectif des SOA est de faire coopérer des modules logiciels locaux ou distants, écrits dans un ou plusieurs langages de programmation. Pour pallier le problème de la localité, on utilise des mécanismes d'annonce et de découverte. Pour résoudre le problème des langages hétérogènes, les modules logiciels utilisent un protocole d'échange commun. L'idée n'est pas nouvelle : ce que font les services Web à l'échelle d'Internet est ce que font les bus logiciels comme CORBA [62] à l'échelle du réseau d'entreprise.

Pour conclure, SOP et SOA résolvent des problèmes à des échelles différentes. On peut dire que la SOP propose une coopération « orientée code » alors que les SOA proposent une coopération « orientée données ». La coopération orientée données a pour avantages de fonctionner entre différents espaces d'adressage et entre des langages et des environnements d'exécution hétérogènes. Son inconvénient est d'ajouter un coût à cette coopération, car il y a toujours une étape intermédiaire de transformation des données ou de stockage. Les communications inter-processus (IPC), les appels distants (RPC) et les bus logiciels proposent une coopération orientée données, avec des surcoûts très variables. La coopération orientée code, elle, fonctionne dans des cas d'utilisation bien plus restreints, mais n'a pas ce surcoût car il n'y a pas d'étape intermédiaire à chaque appel.



(a) API POSIX (b) API objet (c) API service

Fig. 2.6: API offerte par l'environnement d'exécution. Les API objet (2.6(b)) et service (2.6(c)) sont généralement fournies par des intergiciels. La programmation orientée service étant un incrément de la programmation orientée objet, l'API service est fournie par un intergiciel orienté service (p.ex. OSGi, OpenWings), lui-même basé sur un intergiciel orienté objet (p.ex. machine virtuelle Java).

Interfaces de programmation (API) En conclusion de cette section, les environnements d'exécution adaptés ou adaptables aux passerelles domestiques offrent deux grandes familles d'interfaces de programmation : l'API POSIX ou une API orientée service ou composant (figure 2.6). La première a l'avantage d'être implantée et testée sur de nombreuses cibles

matérielles, et dispose d'un grand nombre d'applications déjà existantes. La seconde s'accompagne en règle générale d'un intergiciel particulier qui offre des services non fonctionnels tels que la gestion du cycle de vie des applications. Cette surcouche impose un découpage « propre » des applications en modules et permet d'externaliser des services communs tels que la journalisation ou encore un serveur Web. La combinaison de ces avantages fait qu'une application orientée service ou composant contient presque exclusivement du code métier, et est donc plus rapide à développer que sa contrepartie pour API POSIX.

2.5 Mécanismes d'isolation

Le modèle économique multi-services décrit que plusieurs entités économiques cohabitent sur le même environnement d'exécution, chacune exécutant ses propres services. Intuitivement, la passerelle domestique multi-services doit isoler les services de différents fournisseurs, afin de garantir la confidentialité du code et des données, ou simplement pour imposer des relations de « bon voisinage ». Cette section présente les différentes interprétations de ce que signifie « isoler » dans la littérature.

2.5.1 Terminologie

Les techniques d'isolation se classent en deux familles : l'isolation de nommage et l'isolation de ressources. L'isolation de nommage utilise des espaces ou dictionnaires de noms pour définir à quoi une entité logicielle, application ou autre, peut accéder. Les exemples les plus courants sont :

- Les permissions des utilisateurs Unix. Une opération quelconque sur le système de fichiers réussira ou échouera en fonction des droits de l'utilisateur Unix exécutant l'opération ;
- Les espaces `chroot`. Le répertoire racine « `/` » est changé pour un processus particulier. Le résultat de la commande `ls /` varie alors selon l'espace `chroot` depuis lequel elle a été issue ;
- Des *tags* ou catégories qui partitionnent un registre ou un dictionnaire. Pour reprendre l'exemple des utilisateurs Unix, la commande `ps -u 'whoami'` ne liste que les processus de l'utilisateur courant ;
- De manière générale, tout mécanisme de contrôle d'accès qui répond de manière binaire à la question « ai-je le droit d'accéder à cette ressource ? ».

Par opposition, l'isolation de ressources implique des limites sur la quantité des ressources utilisées, en particulier des ressources matérielles : taux maximum d'utilisation du processeur, espace mémoire accessible. On parle de quotas pour les limites maximum et de garanties pour les limites minimum. Usuellement, les garanties de ressources sont étudiées par les chercheurs des communautés informatique temps réel et embarquée ; nous nous centrerons par la suite sur

les techniques de quotas. La mise en œuvre d'isolation de ressources se traduit communément par :

- Des techniques de comptabilité, insufflées par exemple dans l'ordonnanceur ou dans le gestionnaire de mémoire. Ceci permet d'accepter ou non les demandes d'allocation de ressources, que ce soit selon le processus, l'application ou l'utilisateur ;
- Des techniques de virtualisation, présentées ci-après.

On parle aussi de droits qualitatifs pour l'isolation de nommage et de droits quantitatifs pour l'isolation de ressources.

2.5.2 Types de virtualisation

La notion de virtualisation n'est pas nouvelle. Née dans les années 1970 [63, 64, 65], ses objectifs premiers étaient de protéger les utilisateurs de *mainframes* les uns des autres, et de consolider plusieurs serveurs sur un même mini-ordinateur [66].

Délaissé durant les années 1980 à cause de l'avènement des micro-ordinateurs personnels, la virtualisation vit depuis les années 2000 une seconde jeunesse, due à l'augmentation de la puissance de calcul des PC.

Cependant, les passerelles domestiques d'aujourd'hui disposent de ressources un ordre de grandeur plus petites que celles de nos PC, en terme de puissance de calcul, de mémoire et de stockage en particulier. L'intérêt de la virtualisation pour les passerelles multi-services est de séparer les fournisseurs de services sur une même passerelle domestique. Nous devons donc étudier le compromis entre le degré d'isolation fourni et le coût en performances.

Le terme de virtualisation sous-entend que l'on simule la présence d'une certaine couche matérielle ou logicielle [67]. Les techniques de virtualisation se distinguent par la couche qu'elles virtualisent ; la suite de ce chapitre en propose une classification. Le terme de machine virtuelle désigne une instance de la couche virtualisée (par exemple le système d'exploitation) ainsi que les couches logicielles supérieures que cette dernière exécute (par exemple les applications).

2.5.2.1 Virtualisation au niveau ISA

Le jeu d'instructions (ISA, *Instruction Set Architecture*) est l'ensemble des opérations que le processeur peut exécuter. La virtualisation au niveau ISA s'appelle aussi émulation. Son objectif est plus souvent de pouvoir réutiliser des applications écrites pour d'autres architectures matérielles, voire pour des architectures matérielles dépréciées, plutôt que d'isoler des applications. Par exemple, MAME (Multiple Arcade Machine Emulator) [68] émule les jeux de bornes d'arcades sur toute machine supportant MS Windows, Mac OSX, Linux ou BSD. Bochs [69] émule des machines x86 et QEMU [70] émule des machines x86, PPC, Sparc ou ARM, tous deux sur des système d'exploitation MS Windows, Mac OSX ou Linux. PearPC [71] émule des machines PowerPC sur des systèmes d'exploitation pour x86, et Parallels [72] fait l'inverse.

Une machine virtuelle niveau ISA contient donc l'architecture matérielle émulée (dont le processeur), une émulation des périphériques, un système d'exploitation (noyau inclus) et des applications. Parmi les différents niveaux de virtualisation présentés ici, le niveau ISA est le seul qui permet d'utiliser du code compilé pour une autre architecture matérielle. C'est également la solution la plus gourmande en ressources, et donc la plus lente en théorie.

2.5.2.2 Virtualisation au niveau HAL

La couche d'abstraction du matériel (HAL, *Hardware Abstraction Layer*) masque les différences de matériel, comme les périphériques et la carte mère (gestionnaire mémoire, bus d'entrées/sorties), et offre une interface d'accès unique au noyau du système d'exploitation. Il s'agit d'une interface logicielle, par opposition à l'ISA qui est matériel. Cette abstraction permet d'écrire des noyaux portables sur divers types de matériel, la HAL étant le principal code à adapter.

VMware [73] et VirtualBox [74] sont des exemples de virtualisation au niveau HAL. Les machines virtuelles VMware et VirtualBox hébergent un système d'exploitation (dit « invité » ou « hébergé ») pour architecture x86, sur un système d'exploitation dit « hébergeur » pour architecture x86 également. Virtual PC et Virtual Server [75] sont des technologies de virtualisation au niveau HAL rachetées par Microsoft.

2.5.2.3 Para-virtualisation au niveau HAL

Dans le but d'optimiser le coût de la virtualisation, certaines solutions modifient la HAL virtuelle, par exemple avec des instructions virtuelles ou des registres virtuels. Le système d'exploitation hébergé doit alors être légèrement modifié pour prendre en compte ces différences ; il sera donc informé qu'il s'exécute dans une machine virtuelle.

On parle alors de *para-virtualisation* au niveau HAL, par opposition à la *virtualisation complète* au niveau HAL de VMware et de VirtualBox. Les machines virtuelles ainsi exécutées contiennent chacune un noyau, modifié pour supporter l'interface binaire du para-virtualisateur, ainsi qu'un jeu d'applications.

Xen [76] supporte des noyaux hébergeurs Linux et NetBSD, et des noyaux hébergés Linux, *BSD, Minix, OpenSolaris et autres dérivés d'Unix. Denali [77, 78] a pour objectif de supporter des centaines voir des milliers de machines virtuelles simultanément. Lguest [79] lance des noyaux Linux au-dessus d'un noyau Linux, et met en avant la simplicité : lancer une machine virtuelle revient à exécuter une commande `modprobe` pour charger le module lguest.

Les plates-formes permettant de virtualiser ou para-virtualiser au niveau HAL (VMware, Xen) sont aussi appelées des hyperviseurs, ou des moniteurs de machines virtuelles (VMM, *Virtual Machine Monitor*). On distingue les hyperviseurs de type 1, qui s'exécutent direc-

tement au niveau ISA (en lieu et place du système d'exploitation), et les hyperviseurs de type 2, qui ont besoin d'un système d'exploitation hébergeur. z/VM [80] et son prédécesseur VM/370 [63] d'IBM, VMware ESX Server et les *Logical Domains* [81] de Sun sont des hyperviseurs de type 1 ; Xen, Denali, les autres produits VMware et VirtualBox sont des hyperviseurs de type 2.

User Mode Linux (UML) [82] n'est pas à proprement parler un hyperviseur : il permet d'exécuter des noyaux Linux dans l'espace utilisateur. UML est un moyen simple de lancer des machines virtuelles, mais ses performances sont moins bonnes que celles des autres technologies présentées ici. Mais surtout, UML n'intègre pas de mécanismes avancés d'isolation de ressources ou de nommage.

2.5.2.4 Systèmes d'exploitation à conteneurs

Une autre technique de virtualisation est d'utiliser un système d'exploitation à conteneurs. Le noyau d'un tel système permet de créer des compartiments pour exécuter des *serveurs virtuels*. Ces machines virtuelles contiennent la partie *userspace* du système d'exploitation ainsi que les applications. Par opposition aux hyperviseurs, les conteneurs n'ont pas un noyau par machine virtuelle. L'avantage est de répliquer moins de code pour chaque machine virtuelle, et donc d'espérer un meilleur passage à l'échelle [83] ; l'inconvénient est de ne pas pouvoir tester ou supporter différents noyaux ou différentes versions du même noyau.

Un conteneur dispose de ses propres quotas d'utilisation de ressources (processeur, disque, bande passante) et de son propre espace de nommage (système de fichiers, liste de processus, interfaces réseau, *sockets*).

Les exemples les plus représentatifs sont VServer [83] et Virtuozzo/OpenVZ [84] pour Linux, les *Zones* dans Solaris 10 [85] et les *jails* de FreeBSD [86].

2.5.2.5 Support matériel pour la virtualisation

Certaines architectures matérielles supportent des extensions pour la virtualisation. L'objectif est d'aider à exécuter des systèmes non modifiés dans des machines virtuelles. On passe ainsi de la para-virtualisation à la virtualisation « complète », en limitant l'impact sur les performances.

Sur processeurs x86, Intel VT [87] et AMD-V [88] sont les propositions des deux constructeurs majeurs. Ces supports sont utilisés par KVM (*Kernel Based Virtual Machine*) [89] pour virtualiser complètement les systèmes d'exploitation x86 usuels. Xen version 3 les utilise également pour virtualiser des systèmes d'exploitation propriétaires tels que MS Windows.

Sun offre un support pour hyperviseurs dans ses machines UltraSPARC, appelé sun4v [90]. Solaris 10 utilise sun4v via Logical Domains, un hyperviseur qui héberge des systèmes Solaris

et Linux. IBM propose également ses propres solutions pour architecture POWER, comme Advanced POWER Virtualization pour machines System p [91].

2.5.2.6 Virtualisation en espace utilisateur

Nous avons vu des technologies qui virtualisent du niveau matériel à l'interface fournie par le noyau du système d'exploitation. D'autres se cantonnent à l'espace utilisateur.

C'est le cas de `chroot` (*change root*), qui limite la visibilité qu'a une application du système de fichiers. Il s'agit donc d'une isolation de nommage, et non de ressources. `chroot` limite l'incidence que peut avoir une application non sûre en changeant le répertoire racine « `/` » pour cette application. En revanche, les appels système, l'accès aux entrées/sorties (IOMMU), l'espace disque et le temps processeur ne sont pas virtualisés. D'autres outils peuvent combler certains de ces manques, comme `quota` pour l'espace disque. Cependant, le degré d'isolation sera toujours plus faible qu'avec les technologies citées plus haut.

2.5.2.7 Virtualisation au niveau librairie

La virtualisation au niveau librairie consiste à reproduire le comportement des librairies d'un système d'exploitation particulier vers un autre système. Par exemple, Wine [92] est une implantation de l'API Windows sur les dérivés libres d'Unix. Ces librairies servent principalement pour des raisons de portabilité entre systèmes d'exploitation ; il s'agit donc plus d'émulation que d'isolation.

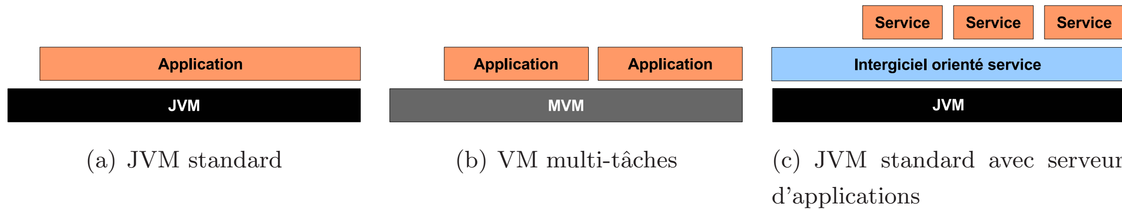
2.5.2.8 Virtualisation au niveau intergiciel

Java et virtualisation Une machine virtuelle Java isole une application dans un « bac à sable », duquel la JVM peut contrôler l'accès de l'application aux ressources du système d'exploitation : système de fichiers, *sockets*, etc. Cependant, selon les spécifications émises par Sun [93], une JVM est mono-tâche. Au lancement de la JVM, celle-ci charge l'application demandée et exécute sa méthode `main()`. La JVM et l'application chargée peuvent démarrer et arrêter des fils d'exécution légers (*threads*), personnifiés par la classe `java.lang.Thread` et l'interface `java.lang.Runnable`. Selon l'implantation de la JVM, un ou plusieurs fils d'exécution légers Java sont projetés sur un ou plusieurs processus au niveau du système d'exploitation ; il s'agit cependant toujours d'une seule application (un seul `main()`) d'un point de vue conceptuel.

Lancer plusieurs applications Java revient à lancer plusieurs machines virtuelles. Beaucoup de ressources sont ainsi gaspillées, car non seulement la JVM elle-même est dupliquée, mais les classes standard Java le sont aussi. Il est possible de partager une partie de la représentation des classes entre JVM [94], mais la taille des JVM même reste prohibitive. Pour passer le

facteur d'échelle, il vaut donc mieux diminuer la duplication de code en n'instanciant qu'une seule JVM.

Pour exécuter plusieurs applications dans une même JVM, deux solutions sont possibles : modifier la machine virtuelle pour qu'elle sache exécuter plusieurs `main()`, ou ajouter une surcouche à la JVM pour le faire. La première approche est celle de la MVM (*Multi-task Virtual Machine*) [95] de Sun ; la deuxième approche correspond aux serveurs d'applications tels que J2EE et OSGi.



(a) JVM standard (b) VM multi-tâches (c) JVM standard avec serveur d'applications

Fig. 2.7: Machines virtuelles mono-tâches et multi-tâches. Une JVM standard (2.7(a)) est mono-application. Pour héberger plusieurs applications dans une même JVM, deux solutions existent. On peut modifier le coeur de la machine virtuelle (2.7(b)). On obtient alors la meilleure isolation entre applications, mais l'effort de développement de la machine virtuelle est important, et doit être dupliqué pour chaque cible matérielle. La deuxième solution est d'utiliser un intergiciel (2.7(c)). L'isolation entre applications est potentiellement plus faible qu'avec la MVM, et repose souvent sur des chargeurs de classes spéciaux ; l'intergiciel est en revanche portable sur toutes les JVM standard.

La MVM est la suite des travaux menés dans le projet Barcelona de Sun. Elle réutilise la notion d'*isolate* [96], qui décrit comment isoler plusieurs applications Java dans une même machine virtuelle, et l'interface de gestion de ressources [97] qui permet de les piloter. Chaque application s'exécute dans sa propre *isolate*, avec ses propres objets et variables de classes. En revanche, les classes elles-mêmes (à l'exception de leurs variables) sont partagées entre les *isolates*. La MVM comptabilise également la consommation mémoire et processeur par *isolate*. Les travaux ultérieurs sur la MVM incluent une optimisation du gestionnaire mémoire multi-tâches [98], la consolidation de la représentation des classes dupliquées entre *isolates* [99], et surtout une extension multi-utilisateurs appelée MVM-2 [100]. L'idée de MVM-2 est de faire correspondre un compte utilisateur du système d'exploitation avec une ou plusieurs *isolates*, par le biais d'une interface de login écrite en C. L'inconvénient de la MVM est que, n'étant par nature pas une JVM standard mais un projet de recherche, elle n'est implantée que pour machines SPARC, ce qui limite son utilisation aux « gros » systèmes.

Par opposition à la MVM, un serveur d'application comme OSGi fonctionne sur une JVM standard. L'avantage est de fonctionner sur toutes les machines disposant d'une JVM, des

environnements embarqués aux serveurs d'entreprise. Un serveur d'application typique charge chaque application dans un chargeur de classes différent [101] ; ce mécanisme a été décrit de manière générique au paragraphe 2.3.2 page 22 et sera détaillé spécifiquement pour OSGi au paragraphe 4.3.1 page 61. Chaque application dispose alors de son propre espace de nommage. En revanche, les serveurs d'application usuels ne proposent pas d'isolation de ressources. En effet, les mécanismes impliqués se situent dans la machine virtuelle elle-même : gestionnaire mémoire, ordonnanceur pour les fils d'exécution légers.

Enfin, d'autres travaux comme KaffeOS [102], Jnode [103] et JX [104] mélangent la machine virtuelle Java au système d'exploitation, ce que certains appellent un *JavaOS*. Ces environnements d'exécution intègrent une gestion des processus, des pilotes de périphériques et autres fonctionnalités traditionnellement fournies par le système d'exploitation, et exécutent du *bytecode* Java. Les JavaOS évitent de dupliquer des fonctionnalités comme la gestion de la mémoire, et permettent une isolation de nommage et de ressources entre processus. Leur inconvénient majeur est de nécessiter un portage vers tous les équipements matériels connus, alors qu'ils sont loin d'être aussi répandus que les JVM standard.

Le tableau 2.1 récapitule les avantages et inconvénients de ces deux solutions pour exécuter plusieurs applications Java dans une même machine virtuelle.

	Isolation de nommage	Isolation de ressources	Optimisations de performance	Cibles et disponibilité
plusieurs JVM	oui	selon système d'exploitation	non	inutilisable
JavaOS	oui	oui	oui	implantations rares
VM spécialisée	oui	oui	oui	gros systèmes
Surcouche à la JVM	oui	non	non	toutes JVM

TAB. 2.1: Support multi-applications dans Java

Java et contrôle de ressources La MVM et les serveurs d'application permettent d'exécuter plusieurs applications Java dans une même machine virtuelle. Ils offrent pour cela une isolation de nommage, voire une isolation de ressources entre applications. D'autres travaux s'attaquent au contrôle de ressources dans la machine virtuelle, sans s'attaquer à la notion de multi-applications.

RAJE [105] modifie la JVM pour comptabiliser l'utilisation de ressources par les objets Java. L'approche est donc similaire à celle de l'interface de gestion de ressources de la MVM.

Les travaux autour de la machine virtuelle virtuelle [106, 107, 108] permettent de spécialiser la JVM pour une application donnée. Cette spécialisation peut se faire au préalable ou en cours d'exécution, et concerne principalement le domaine métier de l'application. Un exemple donné par les auteurs est celui de protocoles pour réseaux actifs, la machine virtuelle elle-

même se transformant alors en réseau actif. Cette technique colle parfaitement aux besoins d'applications spécialisées, mais sont à l'opposé des besoins exprimés dans le modèle économique multi-services, à savoir un environnement d'exécution hébergeant et gérant divers types d'applications simultanément.

J-SEAL2 [109] implante un « micro-noyau » au-dessus de la machine virtuelle Java, c'est-à-dire sans modifier la JVM ni les classes standard. J-SEAL2 réimplante un ordonnanceur et un gestionnaire de mémoire au-dessus de la JVM, et procède par injection préalable de *bytecode* dans les applications qui devront les utiliser. Ceci permet de mesurer la consommation de ressources d'applications Java et de leur fixer des quotas d'utilisation maximaux, le tout uniquement via du code Java portable.

JRes [110] ne modifie pas la machine virtuelle Java elle-même, mais modifie les classes standard Java. JRes comptabilise l'allocation mémoire d'un *thread* ou d'un groupe de *threads* ; pour cela, la méthode standard `loadClass()` a été réécrite pour insuffler dynamiquement le *bytecode* effectuant cette comptabilité dans chaque classe Java lors de son chargement initial en mémoire. JRes comptabilise également le nombre d'octets lus et écrits sur le réseau, en réécrivant le *package* java.net, et fixe des limites de taux d'utilisation du processeur, toujours par *thread* ou groupe de *threads*. Le code comptabilisant l'utilisation du processeur fait appel à du code natif spécifique au système d'exploitation, et n'est donc pas portable.

Ces propositions souffrent de performances moins bonnes que celles de la MVM, partiellement dues à la phase d'injection de *bytecode*, pour une précision moindre dans les mesures. En revanche, leur avantage est un degré variable de portabilité, qui est nulle dans le cas de la MVM.

Java et métriques de ressources Les techniques de contrôle de ressources permettent de remonter des informations sur la consommation de ressources, mais également d'agir sur les consommateurs de ressources (objets, processus légers) en leur fixant des quotas ou en les arrêtant. D'autres techniques permettent uniquement de relever ces mesures, sans moyen de rétroaction. Elles permettent donc de comptabiliser l'utilisation de ressources, mais pas de les isoler.

J-RAF [111] comptabilise l'utilisation du processeur par *thread* Java. Il procède par injection de *bytecode*, et, comme JRes, ne nécessite pas de modifier la JVM. L'unité de comptabilité est l'instruction (*opcode*) de la JVM. Les mesures ainsi effectuées par J-RAF sont donc indépendantes du système d'exploitation sous-jacent.

On peut également citer *Java Platform Debugger Architecture* (JPDA) [112], l'infrastructure de débogage des machines virtuelles de Sun 1.4 et plus. JPDA contient une interface de bas niveau en langage C, pour écrire des sondes de débogage, une interface de haut niveau en Java, permettant d'écrire un débogueur, ainsi qu'un protocole d'échange entre le débogueur

et le processus débogué. JPDA donne accès à potentiellement toutes les statistiques de fonctionnement de la JVM, allant du gestionnaire mémoire aux chargeurs de classes. Cependant, ces mesures ont un coût important, et ne sont utilisées qu'avec parcimonie lors des phases de débogage. JPDA n'est pas conçu pour être utilisable en environnement de production.

Java ME et isolation Java pour le monde mobile s'intéresse également aux techniques d'isolation entre applications Java. Le cas d'utilisation porteur est de déployer plusieurs applications sur un téléphone mobile. L'environnement Java ME concerné est donc en priorité la configuration CLDC avec le profil MIDP (voir la partie 2.3.3.2 en page 25).

Le projet PhoneME Feature [113] de Sun est une machine virtuelle multi-tâches pour ce type de cible. PhoneME isole des MIDlets comme la MVM isole des applications Java. Il s'agit donc d'une machine virtuelle à isolation de ressources, qui a les avantages et les inconvénients d'une Java ME CLDC : légèreté, embarquabilité, mais aussi support limité du cycle de vie des MIDlets et absence de gestion des dépendances.

2.5.2.9 Virtualisation niveau application

Un dernier niveau où la virtualisation peut être mise en œuvre est au sein d'une application de type serveur. Ainsi, les *virtual hosts* du serveur Web Apache [114] permettent d'héberger plusieurs sites Web, correspondant à plusieurs noms de domaine DNS, dans une même instance d'Apache. Le client Web demandant un document en ligne sera redirigé vers le site demandé par discrimination suivant le nom DNS demandé (si plusieurs noms DNS pointent vers l'adresse IP du serveur Web) ou suivant l'adresse IP pointée par le client Web (si la machine serveur dispose de plusieurs adresses IP).

La virtualisation au niveau application sert donc à consolider plusieurs serveurs logiciels sur une même machine physique. Elle n'adresse en revanche pas les problèmes d'isolation.

2.5.3 Tableau synthétique

Les techniques d'isolation et de virtualisation sont une brique importante pour mettre en œuvre des passerelles multi-services. Il est en effet nécessaire d'apporter des éléments de sécurité et de confidentialité aux différents fournisseurs de services. Seulement, le degré d'isolation obtenu est un compromis avec le coût en ressources de l'isolation. Les quantités limitées d'espace disque et de mémoire sont les principaux freins aux techniques d'isolation qu'une passerelle domestique peut adopter. Selon la classification en figure 2.8, l'indice de légèreté de la technique d'isolation doit être de rang A.

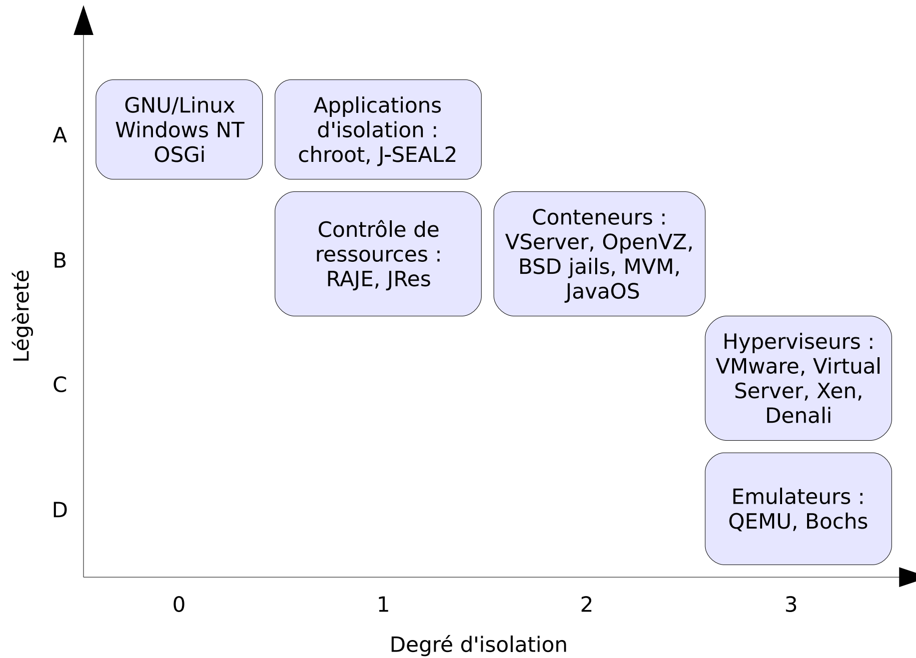


FIG. 2.8: Bilan des techniques de virtualisation. L'axe des abscisses représente le niveau d'isolation fourni : simple notion d'utilisateur (0), isolation de nommage ou de ressources (1), combinaison des deux (2), et exécution de différents noyaux (3). L'axe des ordonnées indique, dans une échelle arbitraire, ce qui est répliqué pour chaque service que l'on veut isoler : depuis l'architecture matérielle HAL (D), depuis le jeu d'instructions ISA (C), une sous-partie de l'environnement d'exécution (B), une sous-partie de l'application d'isolation (A).

Environnements d’exécution pour passerelles domestiques

3

3.1 Expression des besoins 43
3.1.1 Notion d’environnement d’exécution 43
3.1.2 Besoins des passerelles domestiques 44
3.1.3 Besoins des passerelles domestiques multi-services 45
3.2 Spécification des aspects déploiement 45
3.2.1 Unités de déploiement 45
3.2.2 Cycle de vie 46
3.2.3 Gestion des dépendances 47
3.3 Spécification des aspects isolation multi-acteurs 49
3.4 Spécification des aspects gestion 50
3.5 Modèle d’environnement d’exécution pour passerelles domestiques 52
3.6 Styles d’implantation 53

3.1 Expression des besoins

3.1.1 Notion d’environnement d’exécution

Nous avons vu au chapitre 2 qu’une passerelle domestique a besoin d’un environnement d’exécution dynamique et multi-utilisateurs. Nous utilisons le terme générique d’environnement d’exécution pour désigner indifféremment un système d’exploitation, un intergiciel ou un canevas logiciel permettant l’exécution d’applications ou services. Ceci inclut la phase préalable qui est le déploiement d’un service, ainsi que la phase concurrente au déploiement et à l’exécution qui est la gestion du service. À titre d’exemple, dans le cas d’OSGi présenté au chapitre précédent, l’environnement d’exécution est constitué de la plate-forme OSGi, de la machine virtuelle Java dans laquelle elle s’exécute, ainsi que du système d’exploitation sous-jacent.

Un environnement d’exécution classique supporte des fonctionnalités de base permettant à des applications de s’exécuter :

- Gestion des tâches, ou processus, ou processus légers. Un ordonnanceur répartit l’accès au processeur entre ces unités d’exécution ;

- Gestion de la mémoire. Chaque processus dispose de son propre espace d'adressage. Il s'agit d'adresses virtuelles, que le *Memory Management Unit* (MMU) traduit en adresses mémoire physiques. Ceci empêche un processus de lire et modifier impunément les zones mémoire de son voisin ;
- Système de fichiers. L'accès à l'espace de stockage, structuré en répertoires et fichiers, est géré par un système de fichiers : NTFS pour Windows, ZFS pour Solaris, ou encore ext3 pour Linux ;
- Support réseau. Les applications peuvent communiquer via des *sockets* TCP/IP, UDP/IP, ou autres protocoles implantés ;
- Sécurité. En particulier, la notion d'utilisateur système permet de restreindre l'accès aux fichiers et aux processus par un mécanisme de droits ;
- Pilotes de matériel. La diversité des périphériques matériels, comme différentes imprimantes ou différentes cartes vidéo, est rendue transparente par des pilotes spécifiques.

3.1.2 Besoins des passerelles domestiques

Les passerelles domestiques ont des besoins particuliers que les fonctionnalités de base des environnements d'exécution ne satisfont pas. Les points ci-dessus s'appliquent toujours, mais ne suffisent pas. La *Home Gateway Initiative* a listé des besoins additionnels ([115] pages 36–37), largement inspirés d'OSGi. Ils sont répartis en trois catégories :

1. Gestion des logiciels :
   - Configuration d'éléments logiciels appelés modules. Un module dans HGI est semblable à un *bundle* OSGi ;
   - Gestion du cycle de vie des modules : installation, mise à jour, désinstallation ;
   - Sécurité pendant le cycle de vie, aux étapes d'enregistrement, vérification et résolution des dépendances ;
   - Remise à zéro de la passerelle, pour revenir aux paramètres par défaut et pour déboguer le *firmware*.
2. Performance et diagnostics :
   - Tests diagnostiques à distance, pour le matériel et pour les modules logiciels ;
   - Monitorage des performances ;
   - Mécanisme d'événements, envoyés de la passerelle vers son serveur d'auto-configuration (ACS).
3. Utilisateurs en présence :
   - Un super-utilisateur, l'ACS, qui contrôle tout ce qui peut être supervisé et géré ;
   - Un administrateur local, qui contrôle la gestion locale, par exemple le pare-feu ;
   - Les utilisateurs finaux, avec des permissions données par l'administrateur local.

3.1.3 Besoins des passerelles domestiques multi-services

Les besoins exprimés par HGI sont suffisants pour les modèles économiques *triple play* et *multi-play* exprimés au chapitre 1. Cependant, Le modèle économique multi-services impose d'élargir la notion d'utilisateur pour intégrer le partage de l'environnement d'exécution par de multiples fournisseurs de services. En effet, un module HGI peut contenir du code ou des données critiques, ou tout simplement laisser transpirer des informations à propos de son fournisseur : base clientèle, configurations, performance des implantations. Aussi, un module peut contenir du code bogué ou malicieux, qui mettrait en danger tous les autres modules présents. Au vu de ces risques, et en nous inspirant des besoins exprimés par HGI, nous représentons les fonctionnalités des passerelles multi-services sous trois grandes familles :

1. Le déploiement de services. L'usager au domicile peut s'abonner à de nouveaux services, sans redémarrer sa passerelle domestique ;
2. Des interactions avec plusieurs acteurs économiques. En plus de l'usager au domicile, nous différencions le fournisseur d'accès et les multiples fournisseurs de services ;
3. La gestion locale et distante des deux points précédents. Chaque acteur économique interagit avec la passerelle domestique pour déployer et superviser ses propres services.

Ce chapitre explicite tout d'abord chacun de ces trois points, sous la forme de besoins pour les passerelles multi-services. La partie 3.5 exprime, à partir de ces besoins, un modèle informel d'environnement d'exécution pour passerelles multi-services. Enfin, la partie 3.6 discute des différents styles d'implantations possibles de ce modèle, ainsi que ceux que nous avons choisis. Les implantations elles-mêmes font l'objet des deux chapitres suivants.

3.2 Spécification des aspects déploiement

3.2.1 Unités de déploiement

Le terme de déploiement regroupe plusieurs activités, qui sont le rapatriement d'éléments logiciels, leur installation, leur démarrage et arrêt, leur mise à jour, et la gestion de leurs dépendances. Le terme d'élément logiciel, utilisé ici librement, peut signifier un système entier, une application, un fichier, ou ce que la littérature désigne par composant, service, paquet ou encore module. Pour caractériser cette granularité, nous utiliserons le terme générique d'« unité de déploiement » lorsque nous voudrons faire abstraction d'une implantation particulière.

Dans notre cas, une unité de déploiement est une archive qui contient du code (généralement compilé ou prêt à être interprété), des librairies, des fichiers divers dits de ressources (images, documentation), et un fichier de description dit *manifeste*. La granularité visée est celle de l'application ou de la sous-partie d'une application. Pour référence, ceci correspond à ce que OSGi nomme un *bundle*, et à ce que les distributions GNU/Linux nomment un *package*.

3.2.2 Cycle de vie

Une unité de déploiement est amenée à passer par plusieurs états sur une passerelle domestique. Au minimum, elle peut être rapatriée, installée, démarrée, arrêtée et désinstallée, comme représenté en figure 3.1. Pour chaque état et transition, l'environnement d'exécution est susceptible d'être modifié. Par exemple, un processus est créé ou arrêté, ou un fichier est lu ou modifié.

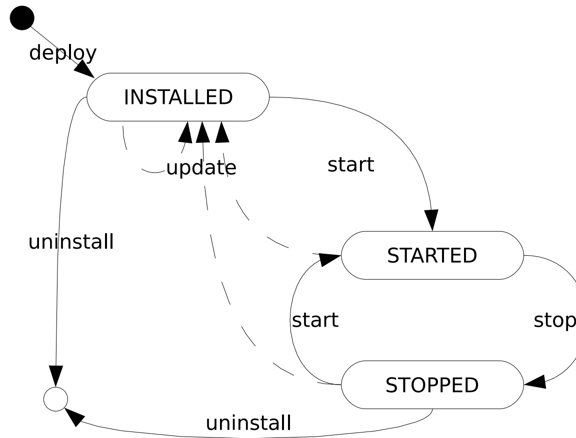


FIG. 3.1: Cycle de vie minimal d'une unité de déploiement

Rapatriement Une unité de déploiement doit pouvoir être rapatriée sur l'environnement d'exécution local. Dans le cas des applications classiques qui nous intéresse, il s'agira de télécharger l'unité de déploiement depuis un dépôt distant, contrôlé par le fournisseur de services ou par un acteur économique tiers. Une fois rapatriée, une unité de déploiement est prête à être installée.

Installation Une unité de déploiement doit être installée avant de pouvoir être démarrée. Suivant les implantations, l'archive déjà rapatriée est décompressée vers un répertoire destination, mise en cache disque ou mémoire, ou laissée telle quelle. D'éventuels scripts de configuration sont lancés, par exemple pour positionner des variables d'environnement.

Démarrage et arrêt Démarrer une unité de déploiement consiste à démarrer l'*activateur* indiqué dans son fichier *manifeste*. Cet activateur peut être un exécutable, une classe Java, un script de lancement ou autre, répondant à une fonction `start()`. Le cycle de vie de l'unité de déploiement est alors représenté par un processus, un processus léger ou une variable d'état. Son arrêt s'effectue en appelant une fonction `stop()`, qui termine le processus ou processus léger concerné ou modifie la variable d'état. Facultativement, l'unité de déploiement

démarrée peut aussi être mise en pause et reprise, sans passer par les appels d'initialisation et de destruction `start()` et `stop()`.

Notons que cette distinction entre l'état installé et démarré n'existe pas dans les EJB ou dans les Xlets (figure 2.5 page 27), mais existe ici et dans OSGi. La logique derrière cette distinction est d'expliciter les phases de résolution des dépendances, décrites ci-dessous.

Mise à jour Une unité de déploiement dispose d'un numéro de version. Elle peut être mise à jour sans avoir à redémarrer la passerelle domestique. Pour cela, elle est arrêtée. Dans le cas d'un service à état, ce dernier est sauvegardé. Le fichier archive de la nouvelle version est rapatrié, installé, puis démarré. C'est à l'activateur de gérer la récupération de l'état précédent le cas échéant.

3.2.3 Gestion des dépendances

Une unité de déploiement peut dépendre de la présence d'autres unités de déploiement pour fonctionner. Ces dépendances peuvent intervenir à plusieurs états du cycle de vie :

- au **rapatriement**. Pour récupérer le fichier archive de l'unité de déploiement, l'environnement d'exécution s'adresse à un dépôt d'archives. Celui-ci fonctionne avec un protocole particulier : HTTP, FTP, CVS, `rsync`. L'environnement d'exécution doit donc disposer des outils adéquats pour utiliser ces protocoles. Dans le cas des passerelles domestiques, ces outils dépendent des fournisseurs de services et des dépôts qu'ils utilisent, et sont donc fournis par eux.
- à l'**installation**. Pour décompresser une archive `.zip` ou `.gz` et pour configurer les paramètres et préférences, d'autres outils sont nécessaires. Dans le cas d'un système dit « orienté source » comme FreeBSD ou Gentoo Linux, c'est-à-dire un système qui rapatrie le code source d'applications pour le compiler localement, d'autres outils comme `make` ou `gcc` sont nécessaires. Les passerelles domestiques sont plutôt des systèmes « orientés binaires », par souci de simplicité de gestion, de puissance de calcul disponible et de garantie de fonctionnement des services livrés par les fournisseurs.
- au **démarrage**. Pour pouvoir s'exécuter, une interface graphique utilisateur a besoin de librairies graphiques ; un point de contrôle UPnP/AV a besoin du pilote approprié ; un service voulant notifier un administrateur a besoin d'un service d'envoi de courriels. On distingue les dépendances obligatoires des dépendances facultatives. Une application qui surveille un frigo UPnP a besoin du pilote UPnP pour fonctionner, mais n'utilise la fonctionnalité de notification par e-mail que si celle-ci est présente localement.

Sur une passerelle domestique, les dépendances au démarrage sont les plus courantes et les plus variées. En effet, un fournisseur décide souvent d'employer un seul protocole pour rapatrier ses unités de déploiement, et de les installer selon une méthode bien définie. Les

dépendances de rapatriement et d'installation d'unités de déploiement provenant du même fournisseur devraient donc être largement similaires.

Dépendances avec état Nous avons vu qu'une unité d'exécution A peut dépendre d'une unité d'exécution B, ce que nous noterons $A \rightarrow B$. Nous venons aussi d'exprimer que A peut dépendre de B dans une certaine phase de son cycle de vie : lorsque A est rapatrié, installé ou démarré. Nous noterons cette phase x, ce qui donne $A_x \rightarrow B$. Il existe cependant un troisième paramètre à prendre en compte : A peut avoir besoin que B soit dans une phase spécifique y de son cycle de vie. Nous avons alors $A_x \rightarrow B_y$. y est la phase minimale que le cycle de vie de B doit avoir atteint pour vérifier la condition. En effet, si B est dans l'état démarré, il vérifie aussi les états rapatrié et installé. De même, si B est installé, il vérifie aussi l'état rapatrié.

Prenons un exemple hypothétique où B est le *package* FUSE (*Filesystem in User Space*) pour GNU/Linux. Si une application native APP quelconque est liée dynamiquement à la librairie partagée `/usr/lib/libfuse.so`, cette librairie doit être présente sur le système de fichiers au démarrage de APP, sans quoi la phase de résolution des symboles échoue. On a donc x = STARTED (l'étape du cycle de vie que l'on souhaite atteindre), et y = INSTALLED (FUSE doit être installé). Avec le formalisme précédent, on note $APP_{STARTED} \rightarrow FUSE_{INSTALLED}$. Prenons un second cas d'exemple : l'application APP' n'est pas liée à la `libfuse`, mais plutôt a besoin d'utiliser FUSE pour monter un système de fichiers quelconque. Dans ce cas, la dépendance signifie que le script `/etc/init.d/fuse` doit être démarré. S'il ne l'est pas, l'application APP' peut démarrer, mais va produire une erreur à l'exécution. Cette dépendance est donc $APP'_{STARTED} \rightarrow FUSE_{STARTED}$.

En pratique, sur des environnements d'exécution comme GNU/Linux ou OSGi, les fournisseurs peuvent prendre soin de séparer en deux *packages* ou *bundles* la partie installation (librairies natives, classes Java) de la partie démarrage (exécutables, services OSGi). Cependant, rien dans le modèle de programmation ou de découpage en unités de déploiement ne les y oblige. Au minimum, l'environnement d'exécution de la passerelle domestique doit prendre en compte le cas où $x = y$, ce que font GNU/Linux et OSGi.

Profils Un fournisseur de services est susceptible de vouloir installer voire démarrer plusieurs unités de déploiement, correspondant à un ou plusieurs services. Ainsi, une offre de services particulières, ou un cas d'utilisation particulier, se traduit par une liste d'unités de déploiement installées et une liste d'unités de déploiement démarrées. Plutôt que de traiter chaque élément de cette liste séparément, on les regroupe parfois sous forme d'un *profil*, appelé aussi configuration ou *runlevel*.

Selon l'implantation, le profil peut être sélectionné au démarrage ou même changé en cours d'exécution. Changer de profil signifie installer et/ou démarrer les unités de déploiement du nouveau profil qui ne le sont pas encore, et arrêter les unités de déploiement démarrées dans

l'ancien profil mais pas dans le nouveau.

3.3 Spécification des aspects isolation multi-acteurs

Taxonomie Plusieurs acteurs économiques sont susceptibles de déployer des services sur une même passerelle domestique. Le code et les données de ces acteurs, ainsi que les données et préférences venant de l'usager à domicile, ne doivent être accessibles que par les entités autorisées. Pour cela, un mécanisme d'isolation est nécessaire. Il en existe cependant plusieurs degrés. On distingue en particulier l'isolation de nommage et l'isolation de ressources.

Isolation de nommage Chaque acteur économique est considéré au niveau système d'exploitation comme un *utilisateur*. Un utilisateur du système ne doit pas pouvoir accéder aux données et aux services des autres ; sont en jeu le respect de la vie privée et de la loi Informatique et Libertés lorsque l'usager au domicile renseigne des préférences, ainsi que la protection de données entre entreprises potentiellement concurrentes. Ce mode d'isolation est dit à espaces de nommage ; on le trouve par exemple dans l'abstraction « utilisateur » des systèmes Unix. Il peut concerner l'accès aux fichiers, la visibilité des processus, les droits de lecture ou d'écriture de données applicatives, l'accès à une interface réseau, ou encore le droit d'utiliser une librairie ou un programme particuliers.

Isolation de ressources Le second mode d'isolation consiste à allouer des quotas d'utilisation de ressources physiques ou bas niveau, comme un espace disque, un nombre de processus, une bande passante ou un pourcentage de temps processeur maximaux. On le trouve par exemple dans les techniques de virtualisation vues au chapitre 2.5. Notons que l'isolation de ressources, appelée aussi isolation forte, n'implique pas forcément l'isolation de nommage. Par exemple, on peut fixer un nombre maximum de processus à un utilisateur Unix, mais cela ne l'empêche pas de voir la liste des processus lancés par les autres utilisateurs.

Degré d'isolation pour passerelles domestiques Les machines de type passerelle domestique sont le théâtre d'opérations à caractère économique ; une isolation de nommage est donc nécessaire. Cependant, elles sont aussi relativement contraintes en ressources physiques. L'isolation de ressources n'aura de sens que si la technique utilisée a un impact négligeable sur l'espace disque, la mémoire et le temps processeur utilisés. Selon la classification donnée en figure 2.8 page 41, la technologie d'isolation choisie doit être de rang A, soit la moins intrusive et la plus légère possible ; la même classification indique que les techniques d'isolation actuelles sont capables à ce prix de fournir ou une isolation par espaces de nommage, ou une isolation par contrôle de ressources.

Pour qualifier le degré d'isolation obtenu, qui est fonction de la puissance de la cible matérielle, nous utiliserons l'expression suivante : chaque utilisateur du système lance ses services et entrepose ses données « *dans son coin* » (*nook* en anglais). Les *coins* utilisateurs sont isolés les uns des autres par la technique d'isolation choisie. Au sein d'un même *coin*, la technique d'isolation n'entre pas en jeu : un fournisseur de services est responsable du bon fonctionnement et de la bonne coopération de ses propres services.

Administrateur Le gestionnaire de passerelle est aussi l'administrateur de l'environnement d'exécution. Ce rôle économique est, dans le cas le plus courant, tenu par le fournisseur d'accès. Le rôle de l'administrateur est de permettre aux autres utilisateurs du système, donc aux fournisseurs de services, de lancer et utiliser leur propre *coin*. L'administrateur dispose lui aussi de son propre *coin* pour héberger ses propres unités de déploiement. Typiquement, le logiciel pare-feu et les éventuels proxys et caches sont hébergés par l'administrateur.

Notion de partage Il est parfois souhaitable de partager une unité de déploiement entre plusieurs utilisateurs du système. Certaines applications comme un serveur Web, certaines facilités comme un service de journalisation, ou encore des librairies telles que des codecs multimédia sont ainsi mutualisables. L'intérêt est d'éviter la duplication de code et de données pour économiser en espace disque et mémoire. Le partage d'unités de déploiement entre utilisateurs doit nécessairement être explicitement autorisé. Par défaut, l'isolation entre *coins* utilisateur doit être respectée.

Nous distinguons deux types de partage : de l'administrateur vers les utilisateurs, et d'un utilisateur à un autre. Au minimum, l'administrateur peut rendre les unités de déploiement de son choix visibles par tous. Les trois exemples cités plus haut (serveur Web, journalisation et codecs) entrent dans ce cas. Le partage explicite d'un utilisateur vers un autre a des cas d'utilisation plus flous ; ayant un grain de contrôle plus fin, il demande la mise en œuvre de mécanismes de partage plus avancés. Ce deuxième mode de partage est donc facultatif.

3.4 Spécification des aspects gestion

Interface unifiée de gestion Un utilisateur du système doit accéder à distance aux opérations de gestion locale, et en particulier aux activités de déploiement. Une session usuelle pour déployer l'unité de déploiement `x` est de la forme :

```
connecter passerelle
rapatrier x
installer x
démarrer x
```

Nous voulons éviter le cas typique des distributions GNU/Linux grand public, où cette session de déploiement demande de connaître plusieurs outils spécialisés, avec une syntaxe différente. Mais surtout, nous voulons éviter la duplication des méta-données que l'on trouve dans ces mêmes outils. Par exemple, les différents types de dépendance vus en 3.2 ne doivent pas être éclatés et partiellement dupliqués entre le paquetage d'installation et le script `rc` de démarrage, comme c'est le cas dans GNU/Linux. Elles doivent au contraire être écrites à un seul endroit et stockées dans un *registre* unique. Les méta-données sont toutes écrites dans le fichier *manifeste*, afin de faciliter le travail du développeur et du *packager*. Lors du déploiement, elles sont accessibles de manière uniforme dans un registre dédié aux unités de déploiement.

Architecture de gestion Chaque utilisateur dispose de sa propre interface de gestion, qui lui permet d'accéder à son propre *coin*. Qui plus est, chaque acteur économique autour de la passerelle domestique est libre de préférer une technologie de supervision particulière. Au minimum, chaque *coin* utilisateur dispose donc de ses propres connecteurs. Selon l'architecture choisie, la passerelle domestique héberge soit un unique agent de gestion, capable de différencier les acteurs économiques en fonction de leurs connecteurs, soit un agent de gestion par *coin* utilisateur. Il s'agit d'un compromis entre flexibilité et passage à l'échelle. Dans le premier cas, l'agent de gestion est plus complexe, mais les *coins* utilisateur nécessitent moins d'espace disque et mémoire. Dans le second cas, chaque acteur économique est libre d'utiliser un agent de gestion de la technologie de son choix, par exemple TR-069, Netconf ou JMX. Cet agent peut être simplifié et allégé selon les besoins. Laquelle de ces deux approches utiliser est déterminé par les choix d'implantation.

L'extrait 3.1, écrit en pseudo-code Java, représente une interface de gestion minimale pour un utilisateur du système.

```java
public interface DeploymentUnitManagement {
  public void    downloadUnit (String unitId) throws Exception;
  public void    installUnit (String unitId) throws Exception;
  public void    startUnit (String unitId) throws Exception;
  public void    stopUnit (String unitId)  throws Exception;
  public void    uninstallUnit (String unitId) throws Exception;
  public Vector  listUnits ();
  public boolean isAlive (String unitId);
}
```

Listing 3.1: Interface de gestion pour un utilisateur

Interface administrateur L'administrateur de la passerelle domestique est, selon le modèle économique multi-services, l'acteur appelé gestionnaire de passerelle. De manière générale, ce

rôle est assumé par le fournisseur d'accès. Il est chargé de gérer trois familles d'activités. Premièrement, tout comme les utilisateurs du système, l'administrateur peut déployer ses propres services. Ils concernent généralement la connectivité réseau : pare-feu, règles de routage, etc. Deuxièmement, l'administrateur contrôle le partage de services vu ci-dessus. Troisièmement, il gère les utilisateurs : il est chargé de créer les *coins* utilisateur, de les démarrer et de les arrêter. L'interface de gestion pour l'administrateur est donc un sur-ensemble de l'interface de gestion pour utilisateur. L'extrait 3.2 représente les fonctionnalités de gestion additionnelles pour l'administrateur. La première partie concerne les *coins* utilisateur ; la seconde concerne le partage d'unités de déploiement dans sa version minimale, c'est-à-dire que l'administrateur partage une unité de déploiement vers tous les autres utilisateurs.

```
public interface DeploymentUnitManagement {
  public void    downloadUnit (String unitId) throws Exception;
  public void    installUnit (String unitId) throws Exception;
  public void    startUnit (String unitId) throws Exception;
  public void    stopUnit (String unitId)  throws Exception;
  public void    uninstallUnit (String unitId) throws Exception;
  public Vector  listUnits ();
  public boolean isAlive (String unitId);
}
public interface NookManagement {
  public void    startNook (String providerId) throws Exception;
  public void    stopNook (String providerId)  throws Exception;
  public Vector  listNooks ();
  public boolean isAlive (String providerId);
}
public interface SharingManagement {
  public void    addSharedDeploymentUnit (String unitId) throws Exception;
  public void    removeSharedDeploymentUnit (String unitId)  throws Exception;
  public Vector  listSharedUnits ();
}
```

Listing 3.2: Interface de gestion pour l'administrateur

3.5 Modèle d'environnement d'exécution pour passerelles domestiques

Nous pouvons représenter l'environnement d'exécution pour passerelles domestiques selon plusieurs représentations graphiques. La figure 3.2 montre l'environnement d'exécution sous forme de briques de fonctionnalités. Certaines briques sont connexes ou complémentaires : celles à gauche concernent la gestion des unités de déploiement, et celles à droite concernent la gestion des utilisateurs, ou acteurs économiques. Ces briques diverses sont regroupées et unifiées sous la coupe d'une interface unifiée de gestion.

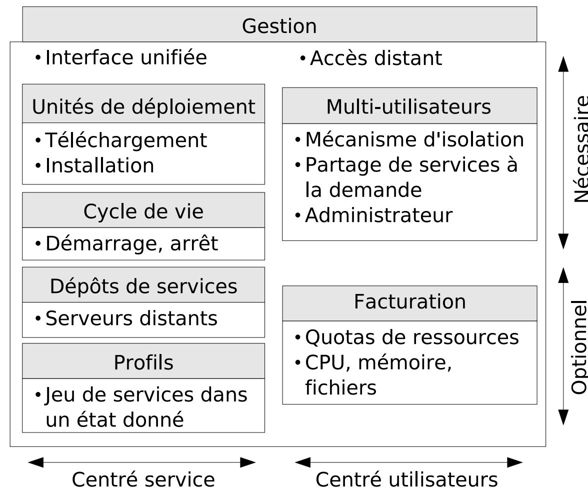


FIG. 3.2: Fonctionnalités d'un environnement d'exécution multi-services. L'interface unifiée de gestion contrôle toutes les autres briques. Les briques en haut sont nécessaires pour être conforme au modèle économique multi-services ; celles en bas sont optionnelles, selon les variantes du modèle économique.

La figure 3.3 montre l'environnement d'exécution sous forme d'architecture en couches. Différents *coins* hébergent un ensemble de services, présentés sous la forme d'unités de déploiement. Les *coins* utilisateur gèrent les services de cet utilisateur, et le *coin* administrateur gère les services communs et les autres *coins*.

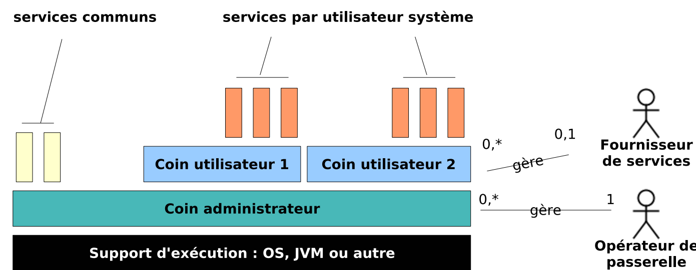


FIG. 3.3: Vue en couches d'un environnement d'exécution multi-services. Chaque utilisateur du système exécute ses unités de déploiement dans son *coin*. L'administrateur exécute les unités de déploiement partagées et gère les *coins* utilisateur. Les acteurs économiques, à droite, gèrent chacun le *coin* qui leur est réservé.

3.6 Styles d'implantation

Le modèle informel d'environnement d'exécution présenté dans ce chapitre peut être implanté de nombreuses manières. Ces styles d'implantation mettent l'accent sur des priorités différentes. Nous en avons retenu deux grands types. Le premier est d'utiliser un modèle de

programmation expressif, en particulier la programmation orientée service, pour faciliter le développement des services, leur coopération et leur gestion. Le second est d'utiliser l'environnement déjà existant qui dispose du plus large catalogue d'outils et d'applications disponibles, dans le but de mettre une barre d'entrée très basse à son adoption.

Ce sont les deux voies que nous avons emprunté. Le chapitre 4 présente notre implantation d'une passerelle multi-services basée sur Java/OSGi ; le chapitre 5 en présente une seconde basée sur GNU/Linux. Ces deux chapitres suivent un plan similaire. Nous comparons tout d'abord Java/OSGi, et GNU/Linux respectivement, au modèle d'environnement d'exécution multi-services présenté ici. Nous présentons ensuite une implantation pour combler les différences. Nous poursuivons par des expérimentations et des mesures de performances, pour conclure sur une discussion. La discussion porte sur l'adéquation de l'implantation obtenue par rapport au présent modèle, sur le jeu de fonctionnalités fourni, et sur des choix d'implantation que nous avons faits. Finalement, le chapitre 6 page 93 compare les deux implantations, propose des alternatives, et conclut le document.

4 Mise en œuvre pour Java/OSGi

4.1 Java/OSGi et le modèle multi-services 55
4.2 Proposition 56
4.2.1 Support local d'exécution 56
4.2.2 Gestion des passerelles logicielles 59
4.3 Implantation 60
4.3.1 VOSGi : Virtual OSGi 60
4.3.2 MOSGi : Managed OSGi 63
4.4 Cas d'utilisation : services e-frigo et UPnP/AV 66
4.5 Expérimentations 67
4.5.1 Environnement de test et méthodologie d'évaluation 67
4.5.2 Coût mémoire 68
4.5.3 Comportement de la couche JMX 69
4.5.3.1 Méthodologie d'évaluation 69
4.5.3.2 Résultats 70
4.6 Discussion 71
4.6.1 Phase de login 71
4.6.2 Performances 72
4.6.3 Partage de services 74

4.1 Java/OSGi et le modèle multi-services

OSGi offre plusieurs avantages pour implanter une passerelle domestique multi-services. Tout d'abord, OSGi impose un modèle de programmation orientée service, introduit au chapitre 2.4 page 28. Ceci permet aux développeurs de découper des applications en unités cohérentes, dont les interactions sont contraintes par des interfaces Java. De plus, la plate-forme OSGi gère le déploiement des applications. Les unités de déploiement, pour les phases de rapatriement, d'installation et de démarrage, sont les *bundles*. Les points d'interaction entre *bundles* sont renseignés lors des phases de résolution des dépendances. On en trouve à trois niveaux : dépendances au niveau *package* Java, dépendances au niveau *bundle*, et dépendances au niveau *service*. OSGi permet également de rapatrier des *bundles* depuis un dépôt OBR

(*OSGi Bundle Repository*). Enfin, toutes les activités de déploiement sont accessibles via la plate-forme OSGi, ce qui autorise une interface de gestion unifiée.

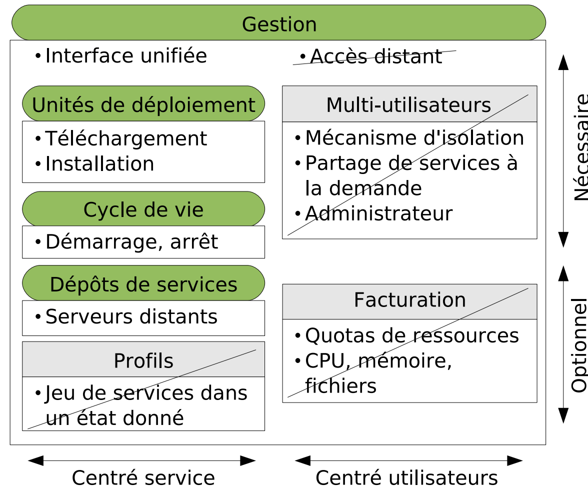


Fig. 4.1: OSGi : fonctionnalités multi-services existantes. Il manque à OSGi tout le support multi-utilisateurs. Les éléments de facturation dépendent de la présence de contrôle de ressources, et en particulier de son support dans la JVM ; ils ne sont donc pas spécifiés. La notion de profil ne fait pas partie des spécifications OSGi, mais apparaît dans certaines implantations. Enfin, l'accès distant aux fonctionnalités de gestion est laissé « ouvert ». Notons la présence de l'interface unifiée de gestion vers les briques de fonctionnalités existantes.

Pour obtenir un environnement d'exécution multi-services, il manque à OSGi tous les aspects multi-utilisateurs. Il manque également la gestion distante de la passerelle, que les spécifications OSGi définissent comme possible, sans toutefois indiquer une mise en œuvre particulière. Sur la figure 4.1, les fonctionnalités présentes dans OSGi apparaissent avec un bord arrondi, tandis que les fonctionnalités absentes sont barrées.

4.2 Proposition

4.2.1 Support local d'exécution

Nous proposons d'ajouter des fonctionnalités multi-utilisateurs à OSGi, sur deux niveaux : d'une part d'un point de vue support local d'exécution, d'autre part d'un point de vue gestion locale et distante. Nous partons du principe que les critères pour choisir une solution particulière sont d'être :

- La moins intrusive possible. La solution choisie doit pouvoir être adaptée aux différentes implantations d'OSGi, de la machine virtuelle Java et du système d'exploitation que les acteurs économiques choisiront. Idéalement, elle est indépendante du SE et de la JVM, et ne nécessite qu'une modification mineure de la couche OSGi ;

- Simple à utiliser. Créer et gérer un *coin* utilisateur doit se faire en quelques commandes seulement ;
- Légère. Les passerelles domestiques ont des ressources relativement limitées, incluant la puissance du processeur, la mémoire et l'espace disque.

Le support OSGi multi-utilisateurs d'un point de vue exécution locale peut être implanté selon trois familles de solutions. La première est de modifier l'API OSGi pour identifier l'utilisateur appelant, par exemple en remplaçant `getServiceReferences(String className, Filter filter)` par `getServiceReferences(UserId caller, String className, Filter filter)`. L'objectif est d'identifier les utilisateurs ayant le droit d'accéder à tel ou tel objet, *bundle* ou service. Cette solution a le défaut d'être intrusive, dans le sens où des *bundles* OSGi classiques ne seront pas compatibles avec OSGi multi-utilisateurs. Mais surtout, l'identification est laissée aux utilisateurs, ce qui est un défaut de sécurité dans la conception.

La deuxième famille de solutions est de laisser l'environnement tracer l'utilisateur appelant. On y gagne l'automatisation de la phase d'identification. Une implantation possible est d'utiliser la méthode `java.lang.Throwable.getStackTrace()` introduite dans Java 1.4. A chaque appel de méthode, l'environnement peut remonter à la classe appelante et vérifier ses droits. Ceci a l'inconvénient d'ajouter un surcoût important à chaque appel de méthode, en plus de remonter seulement au niveau classe et non au niveau objet. Une autre implantation possible est l'injection de *bytecode*. Il est ainsi possible de modifier les appels aux routines de résolution de dépendances OSGi pour y ajouter un identifiant d'utilisateur, et ce sans modifier l'API publique OSGi. Cette approche est peu intrusive et flexible ; son inconvénient est le coût de l'injection de code à chaque déploiement ou mise à jour de *bundle*, qui se traduit en coût de calcul et en mémoire, voire disque si les *bundles* sont mis en cache.

Enfin, la troisième famille de solutions pour rendre le support local d'exécution multi-utilisateurs est de virtualiser OSGi. L'idée est de dédier un environnement OSGi à chaque acteur du modèle économique. L'avantage est d'être simple et non intrusif sur le déploiement et la gestion des applications dans OSGi. Le coût est celui de virtualiser un environnement OSGi par acteur économique présent sur la passerelle domestique, qui se traduit en un surcoût en mémoire.

Apparaissant comme la solution la moins intrusive et la plus simple à utiliser, nous avons choisi de virtualiser OSGi. Le modèle en couches exprimé en section 3.5 page 52 devient donc comme représenté en figure 4.2 :

La passerelle domestique exécute une machine virtuelle Java, qui elle-même exécute une plate-forme OSGi dite passerelle principale. La passerelle principale est gérée par l'opérateur (ou fournisseur d'accès). Son rôle est de s'assurer du bon fonctionnement de la passerelle domestique. La passerelle principale exécute et gère des passerelles dites virtuelles, chacune étant une instance de plate-forme OSGi. Une passerelle virtuelle héberge les services d'un

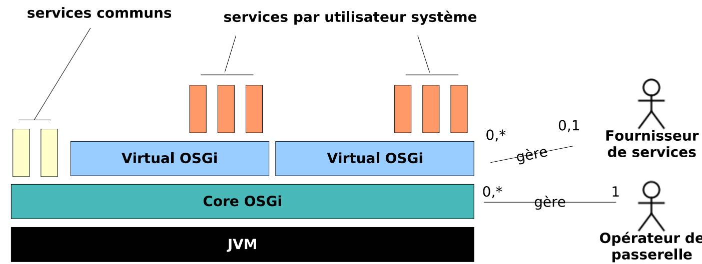


Fig. 4.2: OSGi pour passerelles multi-services. La machine virtuelle Java lance une instance d'OSGi dite principale, qui elle-même lance et gère des instances d'OSGi dites virtuelles. Les *bundles* qui s'exécutent dans différentes passerelles OSGi virtuelles sont isolés les uns des autres, c'est-à-dire qu'ils ne peuvent pas partager des *packages* Java ou des services OSGi. Un fournisseur de services ne voit et ne gère que les *bundles* dans sa propre passerelle virtuelle.

fournisseur particulier. Les passerelles virtuelles sont l'incarnation des « coins » utilisateur pour OSGi multi-services.

Par opposition aux passerelles domestiques qui sont matérielles, nous appellerons « passerelles logicielles » les passerelles principales et virtuelles.

Isolation entre services Chaque fournisseur de services exécute ses propres services dans sa passerelle virtuelle dédiée. Lorsqu'un *bundle* OSGi (ou un service OSGi) déclenche une phase de résolution de dépendances, seuls les *bundles* et services présents dans la passerelle virtuelle courante sont visibles. Les passerelles virtuelles sont donc bien isolées par espaces de nommage, et les services provenant de différents fournisseurs sont bien isolés par défaut.

Notons que l'opérateur, par le biais de la passerelle principale, n'est pas en mesure de lister le contenu des passerelles virtuelles. Ainsi, la passerelle principale contrôle les passerelles virtuelles avec un grain épais, tout en garantissant aux fournisseurs de services la confidentialité de leurs services et des données associées.

Coopération entre services Le cas se présente où certains services peuvent être partagés entre tous les utilisateurs du système, c'est-à-dire par toutes les passerelles virtuelles. Dans le cas d'OSGi, ces services peuvent être des services standard tels que la journalisation (*Log Service*) ou un serveur HTTP (*HTTP Service*). Ces services sont hébergés par la passerelle principale, et peuvent être exportés vers les passerelles virtuelles à la discrétion de l'opérateur. Ceci permet donc de partager des services à la demande, tout en tirant profit de la programmation orientée service offerte par OSGi.

En effet, ce mécanisme de partage est fortement typé et contrôlé par la plate-forme OSGi, contrairement aux mécanismes de partage usuels : *sockets*, IPC et fichiers permettent d'échanger des octets de données, mais leur typage n'est pas garanti et est reconstruit par les appli-

cations.

4.2.2 Gestion des passerelles logicielles

L'opérateur, par le biais de la passerelle principale, agit en quelque sorte comme l'utilisateur *root* (administrateur) des systèmes Unix. Il autorise les utilisateurs du système, ou fournisseurs de services, à lancer et utiliser leur propre environnement d'exécution, ou passerelle virtuelle. La passerelle principale étant chargée de gérer les passerelles virtuelles, elle doit disposer d'une interface de gestion structurée autour de 4 activités :

1. Gestion du déploiement : fournit un moyen de démarrer et arrêter des passerelles virtuelles ;
2. Gestion de la performance : fournit des informations sur l'état de fonctionnement d'une passerelle logicielle (principale ou virtuelle) ;
3. Sécurité : établit l'identification et l'authentification des acteurs économiques accédant aux passerelles logicielles ;
4. Comptabilité et journalisation : donne des informations sur l'utilisation de services ou de ressources par passerelle logicielle.

En plus de ces activités, la passerelle principale est susceptible d'héberger elle-même des services. Elle se conforme donc également aux mêmes interfaces de gestion que les passerelles virtuelles (ci-dessous).

Les fournisseurs de service accèdent à leur passerelle virtuelle via une interface de gestion distance. Selon le modèle économique multi-services, chaque fournisseur est responsable des services qu'il déploie. Il est donc possible et même encouragé de superviser l'exécution de ses propres services sur les passerelles domestiques des usagers au domicile. De plus, certains services ont par nature besoin d'être supervisés : qualité de vie, domotique, télésurveillance, etc. Ces multiples raisons invitent à définir une interface de gestion des services pour les passerelles logicielles : rapatriement, installation, démarrage et arrêt de *bundles*.

Acteurs et gestion dans les spécifications OSGi Les spécifications OSGi définissent les acteurs en présence comme :

- L'opérateur (*operator*), qui contrôle la plate-forme OSGi ;
- Les gérants de déploiement de services (*service deployment managers*), qui déploient des *bundles* avec l'aval de l'opérateur.

OSGi considère le cas où la plate-forme matérielle (*service platform server*) héberge plusieurs plates-formes de services. Cependant, il manque le cas où ces plates-formes multiples sont contrôlées par un utilisateur *root*. Pour intégrer ce concept, une modification mineure

est nécessaire aux spécifications OSGi, représentée par la flèche courbe sur la figure 4.3 : l'opérateur doit pouvoir contrôler le *service platform server* afin de commander le lancement de plusieurs plates-formes de service.

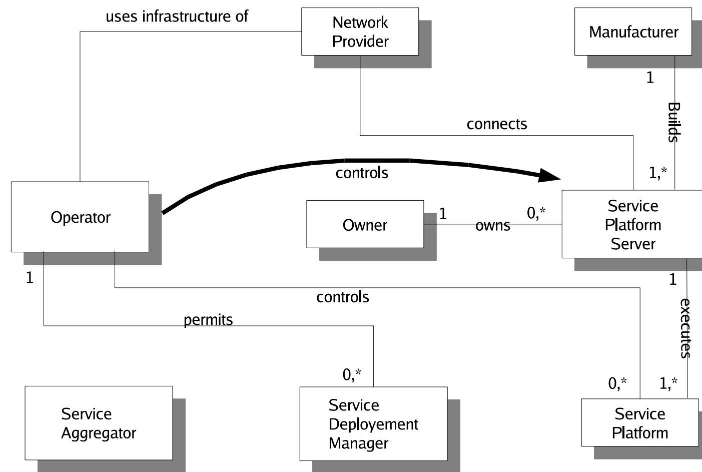


Fig. 4.3: OSGi : diagramme architectural

Nous faisons le choix d'utiliser un agent de gestion par passerelle logicielle. Ceci permet d'une part à chaque acteur économique de choisir la technologie de gestion qui lui convient le mieux, et d'autre part de réutiliser des agents de gestion existants.

4.3 Implantation

4.3.1 VOSGi : Virtual OSGi

Apache Felix Notre code est basé sur Felix [116], un projet de la fondation Apache distribué sous la licence *open source* ASL. Il s'agit d'une implantation de la plate-forme OSGi visant à être conforme à la version 4 des spécifications.

Bundle VOSGi La virtualisation d'OSGi [117] est faite à la granularité d'une instance de Felix. Notre implantation, appelée VOSGi pour *Virtual OSGi*, se présente sous la forme d'un *bundle*. Celui-ci permet de démarrer, arrêter et lister les passerelles virtuelles sur la passerelle principale courante. Ces trois opérations sont mises à disposition via une commande shell utilisable à partir du `shelltui` de Felix :

```
vgw start <profil>
vgw stop <profil>
vgw list
```

Listing 4.1: Commande shell pour VOSGi

Profils Une passerelle virtuelle est symbolisée par un profil particulier. La notion de profil introduite par Felix met un nom sur un répertoire privé, qui contient un cache des *bundles* et les fichiers de données enregistrés par ceux-ci. En démarrant Felix avec un profil particulier, on retrouve la liste des *bundles* déjà installés dans ce profil, dans leur état précédent (démarré, arrêté). De plus, si un *bundle* lit ou écrit dans un fichier qui lui est propre, en utilisant l'appel standardisé `bundleContext.getDataFile(fichier)`, ces données sont enregistrées dans le cache.

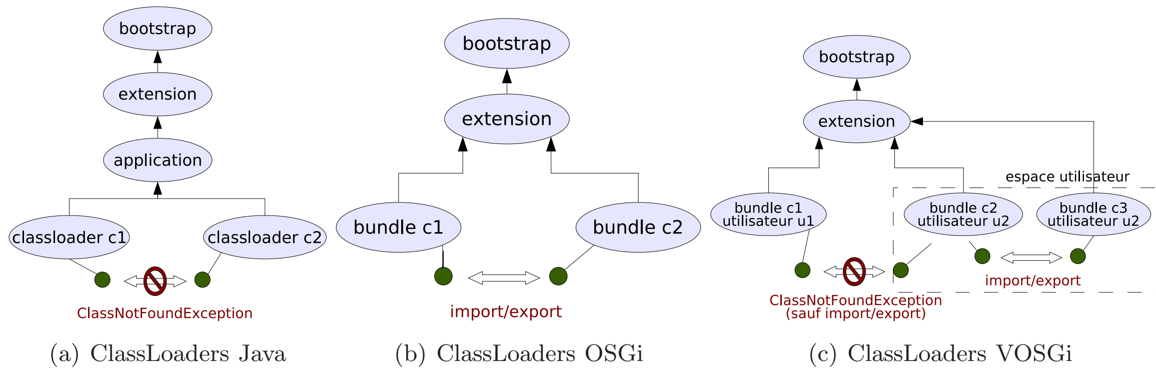
Chaque passerelle virtuelle, donc chaque instance de VOSGi, est associée avec un profil Felix, mais aussi avec une liste de propriétés. Cette liste indique quels *bundles* cette passerelle virtuelle doit lancer au démarrage, en plus des *bundles* présents dans le cache ; elle liste également des préférences pré-provisionnées, comme l'adresse d'un dépôt de *bundles*, le port où trouver un serveur Web, etc.

Isolation de services à l'exécution OSGi utilise un mécanisme à base de chargeurs de classes (*ClassLoaders*) pour contrôler les imports et exports entre *bundles*.

Dans le fonctionnement classique de Java, une *délégation* de chargeurs de classes est mise en place au démarrage de la machine virtuelle [101]. Au sommet de la hiérarchie, le chargeur de *bootstrap* lit les classes Java présentes dans l'installation de la JVM. Chez les JVM de Sun, il s'agit du fichier `rt.jar` ; pour JamVM, qui utilise l'implémentation GNU Classpath des classes standard, ce fichier s'appelle `glibj.zip`. Un second chargeur de classes charge les extensions spécifiques à la plate-forme matérielle et au système d'exploitation en place (répertoire `lib/ext` dans l'installation de Sun). Enfin, le chargeur de classes application (ou système selon les terminologies) charge les classes présentes sur le *classpath* à l'exécution. Lorsqu'un chargeur de classes doit trouver une classe, il commence par déléguer la tâche à son parent (figure 4.4(a)). Si son parent connaît la classe en question, sa définition sera utilisée. Sinon, le chargeur de classes courant essaie lui-même de trouver la classe. S'il ne la trouve pas, on obtient une `ClassNotFoundException`.

Une application Java peut elle-même créer de nouveaux chargeurs de classes, représentés par des ovales en figure 4.4(a). Dans ce cas, et à cause du mécanisme de délégation hiérarchique décrit ci-dessus, une classe chargée par c1 ne peut pas « voir » une classe chargée par c2. Ceci permet d'étendre le mécanisme de chargement de classes en ajoutant une isolation de nommage au sein d'une application.

OSGi fonctionne en cassant en quelque sorte cette hiérarchie (figure 4.4(b)). En effet, à chaque *bundle* est associé un chargeur de classes. Lorsqu'un *bundle* est installé, un chargeur de classes est créé et ajoute dans la machine virtuelle les classes contenues dans le *bundle* ; mais surtout, lorsqu'un *bundle* est désinstallé, son chargeur de classes est supprimé, ainsi que toutes les classes chargées par lui. Ce mécanisme est nécessaire pour pouvoir installer,



(a) ClassLoaders Java (b) ClassLoaders OSGi (c) ClassLoaders VOSGi

Fig. 4.4: Hiérarchies de chargeurs de classes. Les chargeurs de classes Java usuels ne permettent pas à des classes (les petits cercles) chargées par des chargeurs « frères » c1 et c2 de se référencer ; il faudrait que c1 et c2 aient une relation de filiation. Dans OSGi, les chargeurs de classes, ou chargeurs de modules, cassent cette isolation lorsque des imports et exports de *packages* Java sont spécifiés dans les méta-données. VOSGi propose le partage d'OSGi au sein d'un même *coin* utilisateur, et réinstaure l'isolation usuelle de Java entre espaces utilisateurs.

désinstaller et mettre à jour des *bundles* sans redémarrer la machine virtuelle. En revanche, à cause de l'isolation d'espaces de nommage entre chargeurs de classes (figure 4.4(a)), un *bundle* serait dans l'impossibilité d'utiliser les classes d'un autre *bundle*. OSGi définit donc un chargeur de classes particulier qui casse, à la demande, cette isolation. Ainsi, une classe chargée par c1 peut « voir » une classe chargée par c2 si les imports et exports explicitement indiqués correspondent. Ceci permet de partager des *packages* Java et des services OSGi entre *bundles*.

L'idée de VOSGi est d'obtenir un mécanisme de chargement de classes intermédiaire [118] entre le mode Java et le mode OSGi. En lançant des instances virtualisées d'OSGi, nous ajoutons un niveau *utilisateur* dans la hiérarchie des chargeurs de classes. Entre deux utilisateurs (figure 4.4(c)), les classes sont par défaut isolées, comme dans le cas usuel de Java. Par contre, le sous-arbre de chargeurs de classes de chaque utilisateur fonctionne comme en mode OSGi, à savoir que les classes de c2 et de c3 ont toujours la possibilité de se voir, sous réserve d'imports et exports corrects.

Partage de services La passerelle principale a la possibilité de partager des services OSGi vers les passerelles virtuelles. Pour ce faire, le *bundle* VOSGi passe une liste de propriétés additionnelles au constructeur de Felix, qui contient les services partagés. Pour être exact, la passerelle principale indique des couples service OSGi (interface Java) / référence sur l'objet implantant cette interface dans la passerelle principale. Au lancement d'une passerelle virtuelle, celle-ci lit la liste de propriétés additionnelles passée et ajoute à son registre les services partagés. Ce mécanisme nécessite de modifier Felix pour implanter le passage de

propriétés additionnelles.

4.3.2 MOSGi : Managed OSGi

JMX *Java Management eXtensions* est une solution de gestion et supervision pour le monde Java. Deux documents principaux définissent cette technologie : le JSR[1] 3 (Java Management Extensions Specification) [37] et le JSR 160 (Java Management Extensions Remote API) [119]. On distingue plusieurs éléments architecturaux : le superviseur, l'agent et les connecteurs. Le superviseur peut être une console graphique de supervision ou encore un serveur d'auto-configuration (ACS) ; il concentre les données de gestion provenant des équipements supervisés. L'agent est local à l'équipement supervisé, dans notre cas la passerelle domestique ; il remonte des informations de gestion et les met à disposition du superviseur. Enfin, les connecteurs définissent le protocole de transport utilisé entre agent et superviseur ; des exemples courants sont RMI [38] et HTTP.

L'agent remonte les données de gestion en les puisant dans des *sondes*. Dans le cas de la technologie SNMP, l'agent lit ces données dans une base d'informations MIB (*Management Information Base*), laquelle est peuplée via des sondes propres au système d'exploitation et aux applications supervisés. Dans le cas de JMX, la base d'informations est représentée par un ensemble d'interfaces Java de type *MBean*, enregistrées auprès du *MBeanServer* de l'agent. Un MBean (l'interface et son implantation) répond à trois types d'appels : `getXX()` pour lire la valeur de l'attribut `XX`, `setXX()` pour la modifier, et `une_méthode()` pour invoquer un traitement quelconque. Toutes ces méthodes peuvent être implantées en faisant appel à du code Java classique, du code spécifique à une application ou à un système d'exploitation, ou encore par du code natif.

Agent JMX L'agent JMX est fourni sous la forme d'un *bundle* OSGi. En Java 1.5 classique (mono-utilisateur), cet agent est un singleton, c'est-à-dire qu'il n'en existe qu'une seule instance dans toute la machine virtuelle. Or nous avons choisi en section 4.2.2 d'exécuter un agent de supervision par passerelle logicielle, pour assurer l'indépendance de chaque acteur économique en terme de gestion distante, mais aussi en terme de choix de technologie de supervision. La solution retenue [120] est que lorsque le *bundle* agent JMX est utilisé dans la passerelle principale, il réutilise l'agent JMX singleton de la machine virtuelle, si celui-ci est disponible (à savoir dans Java versions 1.5 et plus). Dans les autres cas, ce *bundle* contient une version allégée de l'agent MX4J [121]. Les passerelles virtuelles sont libres d'utiliser ce *bundle* ou non ; par exemple, des *bundles* SNMP sont disponibles depuis le dépôt OBR de l'Alliance OSGi[2].

---

[1] *Java Specifications Request*

[2] `http://www2.osgi.org/Repository/HomePage`

Connecteurs RMI et HTTP Nous utilisons les connecteurs fournis par MX4J, après avoir séparé chacun dans un *bundle* particulier. Le connecteur HTTP utilise l'adaptateur HTTP et le processeur XSLT de MX4J.

Sondes Les sondes font remonter des informations de gestion en provenance de plusieurs niveaux : matériel, système d'exploitation, JVM, OSGi, VOSGi et applications. Nous avons notamment développé des sondes pour :

- Linux : remonte la version du noyau et les ressources disponibles en terme de disque, de temps processeur, de mémoire et de swap. Ces informations sont extraites du répertoire `/proc` tenu à jour par le noyau Linux ;
- OSGi : remonte le nom et la version de l'implantation, ainsi que le profil Felix en cours d'utilisation ;
- VOSGi : liste les passerelles virtuelles lancées. Permet d'en lancer de nouvelles et d'en arrêter ;
- Bundles : donne la liste des *bundles* installés sur la passerelle logicielle courante. Permet également d'installer, désinstaller, mettre à jour, démarrer et arrêter un *bundle* via l'interface de gestion distante ;
- OBR : liste les *bundles* disponibles sur un dépôt distant, et permet de les déployer.

Superviseur Nous avons vu l'architecture de gestion locale à la passerelle domestique, et sa composition en agent, sondes et connecteurs. Il manque à présent les outils qui interagissent à distance avec l'agent, via les connecteurs. En effet, le fournisseur d'accès doit gérer la passerelle principale, les fournisseurs de services doivent gérer leur passerelle virtuelle, et l'usager au domicile doit, le cas échéant, interagir avec les services installés pour renseigner des préférences. Nous utilisons pour cela une console de gestion.

Plutôt que d'utiliser une console de gestion classique comme la JConsole [122] de Sun, nous proposons un outil qui tire parti des aspects dynamiques des passerelles multi-services. En particulier, un fournisseur de services gère une liste de passerelles virtuelles, chacune ayant sa propre liste de MBeans. Durant le fonctionnement normal des passerelles, des MBeans sont susceptibles d'apparaître et de disparaître, en fonction des abonnements de l'usager domestique auprès de fournisseurs de services.

Notre outil, nommé JMXconsole, est une console de gestion JMX pour connecteurs RMI, qui tourne sur environnement OSGi. JMXconsole présente un écran (figure 4.5) en trois parties :

- Á gauche, la liste des passerelles logicielles supervisées ;
- En bas à droite, des informations communes à toutes ces passerelles, en particulier la consolidation des *logs* ;
- En haut à droite, des onglets correspondant aux MBeans présents sur la passerelle logicielle sélectionnée.

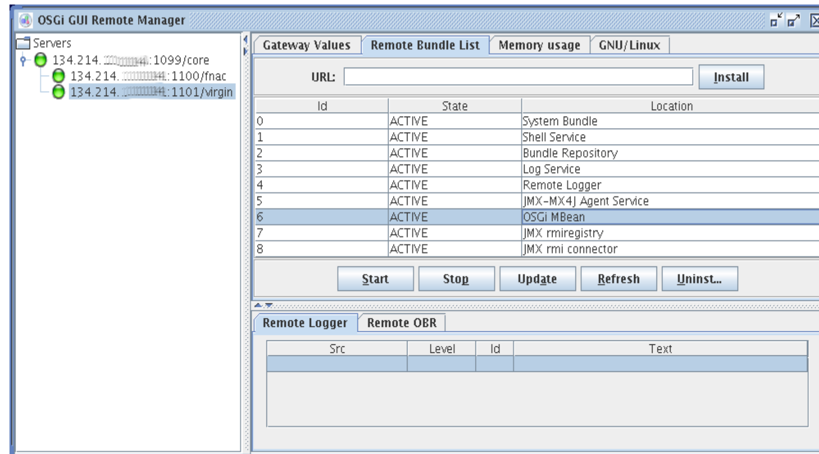


FIG. 4.5: JMXconsole : console de gestion distante. À gauche, la liste des passerelles logicielles gérées depuis la console. En haut à droite, les onglets graphiques spécifiques à la passerelle sélectionnée, son environnement d'exécution et ses services lancés. En bas à droite, les onglets graphiques communs à toutes les passerelles, en particulier la consolidation des journaux d'événements.

Lorsqu'un *bundle* est installé sur une passerelle logicielle, il est libre d'ajouter des sondes permettant de le gérer. Ainsi, un *bundle* lecteur de fichiers MP3 peut ajouter un service, à la fois OSGi et MBean, répondant aux méthodes `start()`, `pause()` et `next()`. L'agent MOSGi écoute l'enregistrement de services OSGi ; il détecte un service qui est également un MBean et le publie automatiquement dans son serveur de MBeans.

Mais l'aspect dynamique de la gestion peut être poussé plus loin. Admettons que le *bundle* lecteur de MP3 ait besoin d'une interface de gestion particulière, par exemple affichant la pochette de l'album en cours de lecture. En plus de déclarer ses propres MBeans, ce *bundle* donne l'interface graphique pour interagir à distance avec ses MBeans. Plus exactement, il donne l'URL où se trouve un onglet graphique pour JMXconsole.

Lorsque le gestionnaire de services sélectionne une passerelle logicielle (à gauche sur la console), JMXconsole récupère la liste des MBeans disponibles sur celle-ci, ainsi que les onglets graphiques à afficher dans la partie en haut à droite. Ce mode de fonctionnement correspond au patron de conception *visiteur* [123].

Le premier avantage est d'utiliser la même console pour tous les acteurs gestionnaires, et de la spécialiser en fonction des MBeans qu'ils déploient via leurs *bundles*. Le second est que la liste des onglets visibles sur la console reflète le contenu de la passerelle logicielle supervisée à un moment donné.

Disponibilité Le code de MOSGi est intégré au projet Apache Felix, et est disponible depuis son dépôt Subversion sous licence ASL. Le code de VOSGi est disponible sous licence CeCILL

auprès de l'INRIA.

4.4 Cas d'utilisation : services e-frigo et UPnP/AV

Pour présenter concrètement le fonctionnement de VOSGi et MOSGi, nous utilisons un démonstrateur basé sur deux services : un réfrigérateur communicant et un service UPnP/Audio-Vidéo.

Scénario Le scénario global pour l'exemple du réfrigérateur communicant est le suivant.

1. L'usager au domicile achète un équipement électroménager doté de fonctionnalités de communication ;
2. L'usager visite le site Web du fabricant pour valider la souscription au service de gestion de l'équipement électroménager ;
3. Le fabricant, alors fournisseur de service, notifie le fournisseur d'accès de l'usager de cette nouvelle souscription ;
4. Le fournisseur d'accès, qui gère la passerelle principale de l'usager, démarre une passerelle virtuelle dédiée au nouveau fournisseur de services ;
5. Le fournisseur de services déploie ses *bundles* sur la passerelle virtuelle. Ces *bundles* servent à contrôler le réfrigérateur sur le réseau domestique, et à rendre ce contrôle accessible depuis l'extérieur.

Pour le service UPnP/AV, le scénario suit les mêmes grandes lignes.

L'application e-frigo Nous utilisons un simulateur de réfrigérateur, dont le comportement est simple : lorsque la porte est close, la température interne tend vers la température du thermostat ; lorsque la porte est ouverte ou l'appareil est éteint, la température interne tend vers la température extérieure. Le réfrigérateur communique sur le réseau domestique via UPnP.

Sur la passerelle domestique, lorsque le *bundle* de gestion du réfrigérateur est déployé et démarré, le gestionnaire de services peut, via MOSGi :

- Lire la température interne ;
- Régler la température du thermostat ;
- Donner une limite de température interne qui, si elle est dépassée, déclenche l'envoi d'une alerte par courriel ;
- Renseigner l'adresse courriel à laquelle faire suivre l'alerte.

L'interface de gestion du réfrigérateur est principalement dédiée au fournisseur de services.

Le service UPnP/AV L'architecture UPnP/AV est constituée de trois entités. Le *media server* héberge du contenu multimédia, et le fournit sous forme de flux. Les *media renderers* sont des écrans, des télévisions ou des haut-parleurs qui jouent le contenu multimédia envoyé par un *media server*. Enfin, le *media controller* gère la connexion entre *server* et *renderers*.

Notre *media server* et *media controller* sont des *bundles* OSGi ; nous utilisons l'application GNU/Linux `gstreamer` comme *media renderer*. Tous trois tournent sur des équipements de type iPaq ou NSLU2.

L'interface de gestion UPnP/AV est dédiée à l'usager au domicile. Ainsi, le *media controller* est divisé en deux parties. Le point de contrôle UPnP/AV est hébergé sur la passerelle logicielle, mais l'interface graphique JMXconsole est déportée sur un assistant personnel (iPaq). La communication entre le point de contrôle et *media server* et *renderer* se fait par UPnP/AV ; la communication entre le point de contrôle et son interface de gestion sur iPaq se fait par événements JMX.

4.5 Expérimentations

4.5.1 Environnement de test et méthodologie d'évaluation

Une passerelle domestique moyenne est équipée d'un processeur autour de 266 MHz, de 64 Mo de mémoire et d'un disque de 16 Mo. Ceci correspond aux modèles de Thomson en pré-production pour le projet MUSE. Nous avons utilisé deux environnements de test ; le premier est un Linksys NSLU2 (couramment appelé SLUG), équivalent de la génération précédente de passerelles domestiques. Le second environnement est un Epia, équivalent de la génération suivante. Leurs caractéristiques techniques sont données dans le tableau 4.1. En plus de l'implantation Felix d'OSGi, certains tests pour la partie gestion utilisent Concierge [124], une autre implantation libre.

	Passerelles actuelles	SLUG	EPIA
Processeur	266 MHz	ARM 133 MHz	VIA Nehemiah (x86) 1 GHz
Mémoire	64 Mo	32 Mo [1]	64 Mo
Disque	16 Mo	8 Mo flash [1]	512 Mo
Java	—	JamVM	JamVM, Sun 1.5
OSGi	—	Felix, Concierge	Felix

Tab. 4.1: Caractéristiques techniques des environnements de test

Pour mesurer l'impact des couches de virtualisation et gestion d'OSGi sur les ressources locales à la passerelle domestique, nous avons réalisé les tests suivants.

[1] Le SLUG dispose de ports USB ; nous avons utilisé une clé pour ajouter 32 Mo de swap et de l'espace disque pour le cache des *bundles*.

Les fournisseurs de services exécutent des *bundles* OSGi qui allouent une certaine taille mémoire. Dans les deux premiers tests (figure 4.6), cette taille est de 1 Mo. Ceci est suffisant pour héberger par exemple plusieurs petits jeux, des applications embarquées et un mini-serveur Web. A titre indicatif, le point de contrôle UPnP/AV que nous avons réalisé utilise 600 Ko de mémoire. Pour le troisième test (à droite sur la figure 4.6), la mémoire allouée est de 2,6 Mo, valeur empiriquement déterminée comme étant réclamée par un lecteur MP3 lisant un fichier encodé en 128 Kb/s.

Dans le premier jeu de tests, tous les fournisseurs de services exécutent leur service dans la même passerelle logicielle. Nous mesurons donc le comportement mono-utilisateur standard d'OSGi. Dans le second jeu de tests, chaque fournisseur de service dispose de sa propre passerelle virtuelle, mais sans disposer d'agent de gestion. Nous mesurons le coût de la virtualisation d'OSGi seule (VOSGi). Dans le dernier jeu de tests, chaque passerelle virtuelle dispose de son propre agent de gestion JMX. Nous mesurons donc le coût conjoint de VOSGi et MOSGi, dans un mode de fonctionnement qui correspond au modèle économique multi-services.

La métrique relevée est la consommation mémoire globale du système, à savoir la consommation conjointe de GNU/Linux, de la JVM, d'OSGi et des surcouches VOSGi et MOSGi, ainsi que des services exécutés.

4.5.2 Coût mémoire

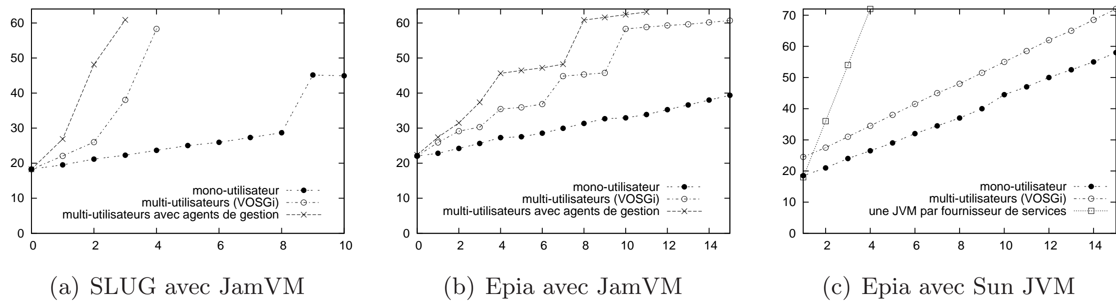
La figure 4.6 montre la consommation mémoire cumulée par toutes les couches logicielles (axe des y) par rapport au nombre de *bundles* de test exécutés, qui correspond au nombre de fournisseurs de services sur la passerelle domestique (axe des x).

Sur l'Epia exécutant JamVM (figure du milieu), nous observons que la passerelle domestique supporte de l'ordre de la dizaine de fournisseurs de services simultanés. Ceci reste confortable pour le modèle économique multi-services. Par régression linéaire, nous obtenons un surcoût en mémoire de 3 Mo par fournisseur de services lié à VOSGi et MOSGi conjointement, dont 1,6 Mo dus à VOSGi.

Le processeur et l'espace disque ne subissent pas de surcharge significative ; les réels consommateurs sont les *bundles* que les fournisseurs de services déploient.

On peut remarquer plusieurs irrégularités sur les courbes en 4.6. Premièrement, les courbes pour SLUG-JamVM et Epia-JamVM ont une importante différence de pente. Elle est due à un problème d'implantation de JamVM sur processeurs ARM. L'impact dans notre cas est que le passage à l'échelle est réduit par trois sur processeurs ARM. Cette architecture est souvent utilisée sur les passerelles domestiques d'aujourd'hui ; pour avoir les meilleures performances possibles, il est donc préférable soit de corriger le problème dans JamVM, soit d'utiliser une autre machine virtuelle, comme la J9 d'IBM.

La deuxième remarque est que sur Epia-SunJVM, le coût de VOSGi est de 1 Mo par



(a) SLUG avec JamVM (b) Epia avec JamVM (c) Epia avec Sun JVM

FIG. 4.6: VOSGi/MOSGi : Consommation mémoire sur 3 environnements de test. Les courbes montrent la mémoire consommée en méga-octets par l'environnement d'exécution (système d'exploitation, JVM, OSGi) en fonction du nombre d'utilisateurs système. Les points noirs indiquent que tous les services s'exécutent dans la même instance d'OSGi (mode mono-utilisateur). Pour les points blancs, chaque service s'exécute dans un coin utilisateur dédié, sans agent de gestion. Les courbes avec des croix ajoutent un agent de gestion par coin utilisateur. Enfin, sur Epia avec Sun JVM nous avons lancé une JVM par service (carrés blancs) pour montrer la pénalité sur le passage à l'échelle.

fournisseur de services, contre 1,6 Mo pour Epia-JamVM. De plus, la courbe est nettement plus linéaire ; ceci s'explique par une différence de politique d'allocation mémoire entre JamVM et Sun JVM. Nous voyons donc encore une fois l'importance du choix de la JVM sur les performances obtenues.

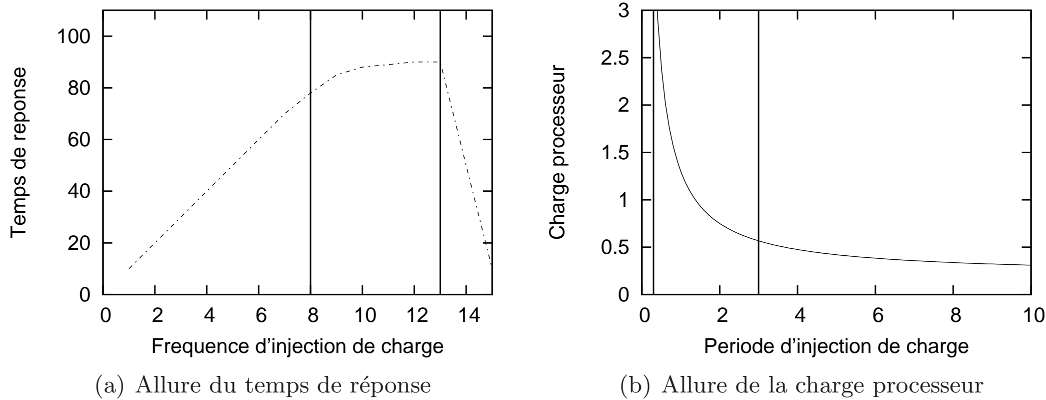
4.5.3 Comportement de la couche JMX

4.5.3.1 Méthodologie d'évaluation

Nous avons évalué nos extensions à OSGi en terme de consommation de ressources locales à la passerelle domestique, c'est-à-dire en adoptant un point de vue système d'exploitation. Prenons à présent un point de vue gestion, et observons le comportement d'un OSGi multi-services en fonction de stimuli provenant du réseau.

La plupart des tests de performance dans la littérature [125, 126] observent le temps de réponse d'un logiciel en le soumettant à différents jeux de requêtes. Ces requêtes peuvent varier en type, en taille et surtout en fréquence. En variant la fréquence des requêtes, on obtient communément une courbe de la forme représentée en figure 4.7(a). Trois zones sont identifiables. Dans une première partie, le temps de réponse évolue de manière pseudo-linéaire avec la charge. Ensuite, le système n'arrive plus à maintenir un taux de réponse maximal et perd des requêtes en file d'attente : il « plie le genou ». Enfin, arrive le point où le système sous test s'écroule.

Pour évaluer le comportement de l'agent JMX, une machine de type PC émet des requêtes



(a) Allure du temps de réponse

(b) Allure de la charge processeur

FIG. 4.7: Comportement typique d'un système sous stress. Sur la figure 4.7(a), le temps que met le système sous stress à répondre à une requête augmente linéairement avec la fréquence des requêtes injectées. Dans une seconde phase, il plie le genou et commence à rejeter des requêtes, pour enfin s'écrouler sous la charge. Sur la figure 4.7(b), les axes sont inversés : les abscisses montrent la période de temps entre deux injections de requêtes, et les ordonnées montrent la charge de calcul imposée sur le système sous test.

`getCpu()` vers une passerelle virtuelle. La machine de test est le SLUG avec JamVM ; l'agent JMX évalué est l'implantation de MX4J. Le PC superviseur (processeur double coeur à 2 GHz, 1 Go de mémoire) est plus puissant que la passerelle domestique car il ne doit pas être un facteur limitant sur les mesures effectuées. De même, le PC superviseur et la passerelle domestique sont sur un réseau local isolé, car nous voulons faire abstraction des délais dus au réseau. Enfin, la requête `getCpu()` lit la valeur d'un attribut simple et a une durée de traitement que nous considérons négligeable. Un appel du superviseur à `getCpu()` a sensiblement le même temps de réponse qu'un appel à une méthode `getValeur()` qui retourne une constante, sans faire d'autres calculs.

4.5.3.2 Résultats

Les mesures effectuées sont représentées en figure 4.8. La courbe lisse est une courbe pseudo-théorique de la forme $y = \alpha/x + \beta$. α équivaut à $y = 100\%$ CPU ; c'est le temps de réponse à `getCpu()` avec un délai dû au réseau nul. β est la charge résiduelle lorsque la passerelle domestique n'exécute aucun service particulier ; on l'appelle aussi « bruit système ». Cette courbe n'est que pseudo-théorique car elle est obtenue à partir de deux points empiriques, par exemple à des périodes de requêtes de 100 et 400 ms (axe des x).

La courbe verte épaisse est le temps de réponse effectivement mesuré en invoquant `getCpu()` à différentes périodes, allant de une fois toutes les 1000 ms à une fois toutes les 10 ms. Les pics que l'on observe correspondent à des *garbage collections* dans le système sous test.

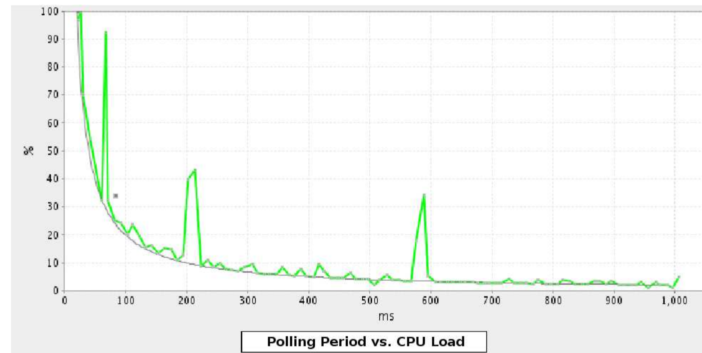


Fig. 4.8: MOSGi : temps de réponse de l'agent JMX. En abscisses, la période à laquelle la sonde `getCpu()` est appelée. En ordonnées, la charge processeur induite sur la machine sous stress (un SLUG avec JamVM). L'asymptote vers $x = 0$ donne le temps de réponse α à une requête lorsque le délai réseau est nul. L'asymptote quand x tend vers ∞ donne le « bruit système », soit la charge à vide.

La courbe se lit ainsi : si l'on autorise la couche de gestion à n'utiliser que 5% du temps processeur au maximum, alors la sonde `getCpu()` peut être interrogée toutes les 500 ms. Avec environ 10 fournisseurs de services sur la passerelle domestique, chacun peut appeler une sonde simple toutes les 5 secondes sans surcharger le processeur.

4.6 Discussion

Au début de ce chapitre, nous avons présenté le décalage entre OSGi et le modèle d'environnement d'exécution multi-services donné au chapitre 3. Nous avons ensuite proposé et testé une implantation, composée de VOSGi pour la partie isolation et de MOSGi pour la partie gestion. Nous discutons ici les fonctionnalités à parfaire, certains choix d'implantation et leur impact sur les performances.

4.6.1 Phase de login

L'isolation entre utilisateurs du système fait partie des mécanismes qui contribuent à la sécurité de l'environnement d'exécution. Cependant, en amont de l'isolation locale fournie par VOSGi, chaque utilisateur du système doit disposer d'un accès distant à son interface de gestion. Par manque de temps, l'implantation de cet accès distant n'est pas encore sécurisé. Il manque un mécanisme de *login* pour identifier et authentifier les utilisateurs. Puisque chacun dispose de son propre agent de gestion, nous pouvons modifier légèrement l'agent pour utiliser un connecteur SSL ou TLS et pour accepter un mot de passe utilisateur.

4.6.2 Performances

Pour évaluer l'implantation de VOSGi et de MOSGi, il faut mesurer deux impacts. Le premier, sur l'environnement d'exécution, se mesure en consommation disque et mémoire principalement. Le second, sur les performances de gestion, se mesure en temps de réponse.

Nous discutons ici de l'impact sur l'environnement d'exécution, mesuré précédemment. Nous proposons également une méthode d'évaluation de l'impact sur les performances de gestion, et concluons sur l'utilisabilité de VOSGi et MOSGi sur des passerelles domestiques.

Performance mémoire Suivant l'environnement de test considéré, un *coin* utilisateur occupe de 2 à 3 Mo de mémoire, indépendamment des services déployés. Ce coût, bien qu'il soit raisonnable et permette de se conformer au modèle multi-services, n'est pas négligeable. Pour améliorer ce point, nous pouvons travailler sur l'infrastructure de virtualisation et sur celle de de gestion.

Le coût de VOSGi est balancé par la possibilité de partager des services OSGi entre passerelles virtuelles, et ce, sans modifier le système d'exploitation ou la machine virtuelle sous-jacents. Pour garder cet avantage tout en réduisant le coût de VOSGi, nous pouvons passer de l'implantation Felix à une autre plus légère, comme la récente Concierge. Les autres alternatives à VOSGi souffrent par contre soit d'un mauvais passage à l'échelle, soit d'une dégradation du modèle de programmation, soit d'un problème plus terre à terre de disponibilité. Pour implanter des *coins* utilisateur, nous pouvons lancer une machine virtuelle Java par utilisateur. Nous avons vu en section 2.5.2.8 page 36 que ceci passe mal le facteur d'échelle, et que résoudre ce problème est l'idée motrice derrière la machine virtuelle multi-tâches (MVM).

La seconde alternative est donc d'utiliser une MVM, en dédiant un *isolate* (ou tâche) à chaque acteur économique. Dans le monde Java, il s'agit du meilleur compromis disponible entre passage à l'échelle et isolation de ressources. Cependant, la MVM tourne sur architecture SPARC et système d'exploitation Solaris, qui visent les gros serveurs ; cette implantation n'est donc pas utilisable sur une passerelle domestique. La seconde et dernière implantation que nous connaissons, phoneME, cible en revanche les environnements Java ME. En particulier, une version en cours de développement utilise la configuration CDC, censée supporter OSGi. Nous n'avons pas testé le bon fonctionnement de phoneME sur passerelle domestique, ni son comportement avec OSGi ; il s'agit d'une piste de travaux futurs. Notons tout de même que dans une machine virtuelle multi-tâches, chaque *isolate* dispose d'un espace d'adressage particulier. Ainsi, si phoneME permet de mettre en place une architecture orientée service pour faire coopérer divers *isolates*, elle ne permet pas de le faire par programmation orientée service.

Enfin, une troisième alternative à VOSGi est d'utiliser une isolation de plus bas niveau, comme par exemple Xen ou VServer. On obtient alors le meilleur degré d'isolation disponible

aujourd'hui. Cependant, tout comme pour la MVM et phoneME, la programmation orientée service entre machines virtuelles Xen reste à démontrer comme étant possible. De plus, avec les implantations actuelles, chaque machine virtuelle Xen doit héberger sa propre machine virtuelle Java, avec les inconvénients cités plus haut.

En conclusion, si le coût de VOSGi reste non négligeable et peut être réduit par des efforts d'implantation, cette solution reste le meilleur compromis entre degré d'isolation, modèle de programmation et performance pour les environnements multi-services basés sur Java.

Agents de gestion Un agent MOSGi utilise autour de 1,5 Mo de mémoire. Ce coût est dupliqué pour chaque *coin* utilisateur, en l'occurrence pour chaque passerelle virtuelle VOSGi. Pour réduire ce coût, là encore, nous pouvons faire des efforts d'implantation pour obtenir des agents plus petits. Une alternative est de n'utiliser qu'un seul agent de gestion pour tous les acteurs économiques. Dans ce cas, la MIB de chaque acteur est identifiée par un espace de nommage, que l'agent doit mettre en œuvre et prendre en compte lorsqu'il enregistre des MBeans et transmet des requêtes. L'intérêt est de réduire le coût d'un *coin* utilisateur au coût de VOSGi uniquement. Il faut alors modifier l'agent pour assurer la sécurisation des données dans le serveur de MBeans. Toutefois, le plus gros problème technique est que l'interface de gestion d'un fournisseur de services se retrouve en dehors de son espace de nommage VOSGi. Faire le lien entre les deux est contre-intuitif et demande alors de casser le système d'isolation.

En plus de cet obstacle technique, choisir entre un agent unique et un agent par *coin* utilisateur est aussi un choix économique. En effet, dans le second cas, chacun est libre d'utiliser la technologie de gestion de son choix, ce qui nous semble un avantage important.

Amélioration du comportement de JMX En plus de la consommation mémoire de l'agent JMX, nous pouvons tenter d'améliorer le temps de réponse aux requêtes. Le comportement de JMX en fonction de la charge doit en effet être plus finement étudié [127], en particulier pour les environnements à ressources de calcul modestes.

Nous avons mesuré qu'un appel à une sonde vide, c'est-à-dire qui retourne immédiatement, prend environ 25 ms. Ces 25 ms sont dues à la traversée des couches RMI (pour l'appel distant) et JMX (pour trouver le MBean demandé dans le registre JMX). Le temps dû à la latence du réseau est ici négligé. Notons que cette mesure diffère selon l'environnement de test : elle est bien de 25 ms avec un noyau Linux 2.6, la librairie C `glibc` et l'implantation Felix d'OSGi, mais passe à 80 ms avec un noyau Linux 2.4, `uclibc` et Concierge OSGi.

Nous observons sur la courbe 4.8 qu'un appel à la sonde `getCpu()` prend environ 30 ms. Cela signifie que pour une sonde simple, plus de 80% du temps de réponse (à α et β près) est dû au fonctionnement de JMX. De plus, ces 30 ms sont répétées à chaque fois que le superviseur invoque une sonde.

Une optimisation indispensable est donc de créer des « super MBeans », qui regroupent plusieurs requêtes en une seule. Ainsi, lorsque le superviseur veut appeler par exemple `getCpu()`, `getMem()` et `getDisk()`, il utilisera une seule super requête `getConcatAttributes()`. L'appel arrive sur l'agent JMX de la passerelle domestique et est démultiplexé par le super MBean. Les 3 sondes processeur, mémoire et disque sont consultées, les valeurs de retour sont regroupées et renvoyées au superviseur. Enfin, le superviseur démultiplexe la réponse pour dissocier les 3 valeurs demandées. Ainsi, le coût de traversée de JMX n'est payé qu'une seule fois au lieu de trois. La méthode standard `getAttributes()` permet de réaliser ceci en regoupant des appels à plusieurs MBeans en une seule requête ; cependant, elle subit quand même le coût de recherche de chacun des MBeans impliqués auprès du registre de MBeans. Avec l'exemple ci-dessus, nous créons un unique MBean à interroger, qui regroupe dans sa méthode `getConcatAttributes()` tous les appels aux sondes demandées. Nous obtenons ainsi un temps de réponse d'environ 42 ms avec une requête multiplexée, contre 90 ms avec 3 requêtes distinctes.

4.6.3 Partage de services

L'implantation actuelle du partage de services dans VOSGi est statique et unidirectionnel : le *coin* administrateur fournit, à la création des *coins* utilisateur, des références sur les services partagés. Un *coin* utilisateur qui démarre récupère ainsi une propriété `vosgi.export-services`. Ses *bundles* peuvent alors interroger son registre, via la méthode `getServiceReferences`, et récupérer des références sur les services partagés.

Une première amélioration serait de rendre ce partage dynamique, en supportant l'ajout et la suppression de services partagés durant toute la vie des *coins* utilisateur. Il s'agit en effet d'une limitation de l'implantation de VOSGi, et non de sa conception. Pour cela, le registre OSGi du *coin* administrateur doit pouvoir communiquer par notification avec le registre de chaque *coin* utilisateur. Le point dur est probablement la suppression de services partagés : il faut alors proposer une politique pour les *garbage collections* du gestionnaire mémoire de la machine virtuelle. Deux choix sont possibles pour supprimer un service partagé : attendre que tous les consommateurs du service libèrent leur référence sur le service, ou forcer son arrêt et laisser les consommateurs gérer le cas d'erreur. Il s'agit d'une décision plus économique que technique.

La seconde amélioration possible est de partager multi-directionnellement les services, c'est-à-dire qu'un utilisateur peut proposer des services aux autres utilisateurs de son choix. Le mécanisme de communication entre registres OSGi en devient alors plus complexe. Si cette amélioration présente un intérêt technique, nous n'avons cependant pas trouvé de motivation économique pour la soutenir.

5 Mise en œuvre pour GNU/Linux

5.1 GNU/Linux et le modèle multi-services 76
5.1.1 Fonctionnalités existantes 76
5.1.2 Outils existants et leurs limites 77
5.1.2.1 Gestionnaires de paquetages 77
5.1.2.2 init et rc 78
5.1.2.3 Mécanismes d'isolation entre acteurs économiques 79
5.2 Proposition : HGL 80
5.2.1 Architecture 80
5.2.2 Aspects déploiement 81
5.2.3 Aspects multi-acteurs 82
5.3 Implantation 83
5.3.1 Aspects déploiement 83
5.3.1.1 Gestionnaire de paquetages 83
5.3.1.2 Dépôts de bundles 84
5.3.2 Cycle de vie 84
5.3.3 Aspects multi-acteurs 85
5.3.3.1 Coins utilisateur 85
5.3.3.2 Partage de services 86
5.3.4 Interface de gestion 86
5.4 Expérimentations 87
5.4.1 Intégration avec TR-069 87
5.4.2 Mesures de performance 87
5.4.2.1 Mémoire 87
5.4.2.2 Disque 88
5.4.2.3 Analyse des résultats 88
5.5 Discussion 89
5.5.1 Fonctionnalités 89
5.5.2 Choix d'implantation 90
5.5.3 Partage de services dans HGL 91
5.5.4 Modèle de programmation : services et méta-données 91

5.1 GNU/Linux et le modèle multi-services

L'intérêt d'utiliser GNU/Linux sur des passerelles multi-services est avant tout d'avoir un système relativement stable et éprouvé, qui dispose d'une multitude d'outils déjà existants.

Le terme GNU/Linux désigne l'écosystème où l'administrateur système installe un noyau Linux, dans un numéro de version et un *patchset* de son choix, ainsi qu'un ensemble d'applicatifs (programmes dans l'espace utilisateur). Ces applicatifs vont des librairies usuelles, comme la `libc`, `libm` ou `libutil`, aux outils standard Unix : `ls`, `cat`, `ps`, `init`, GNU binutils ou encore coreutils. Au minimum, les applicatifs installés doivent permettre de démarrer et utiliser le système.

Dans le but d'implanter une passerelle domestique multi-services, nous allons donc nous baser sur une collection d'applicatifs existants, et les modifier ou compléter la collection le cas échéant. Nous avons vu au chapitre 2.3 page 19 que GNU/Linux traite, au moins partiellement, les éléments requis par le modèle économique multi-services qui sont représentés en figure 5.1.

5.1.1 Fonctionnalités existantes

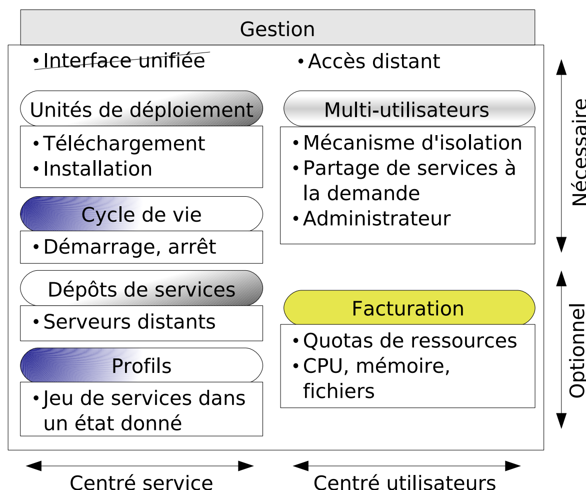


FIG. 5.1: GNU/Linux : fonctionnalités multi-services existantes. La plupart des briques existent (bord arrondi), mais ne sont pas uniformes, p.ex. unités de déploiement et dépôts concernent des paquetages, tandis que cycle de vie et profils concernent des scripts `rc`. Il manque une interface unifiée de gestion pour rendre le tout cohérent.

Pour la plupart des « boîtes » de fonctionnalités demandées, il existe des outils répondant au moins partiellement aux besoins des environnements multi-services. Le principal point manquant est une interface unifiée de gestion, permettant d'accéder à toutes les activités de gestion sur la passerelle domestique. En effet, les aspects déploiement et multi-acteurs sont

chacun gérés par plusieurs applicatifs, qui ne forment pas un tout cohérent et uniforme. Pour la partie déploiement, on distingue :

- Le rapatriement et l'installation. Sur les distributions GNU/Linux usuelles, ces activités sont commandées par le *gestionnaire de paquetages*.
- La gestion du cycle de vie. Le démarrage et l'arrêt des applications sont généralement commandés par des scripts `rc`.

Ces deux familles d'outils disposent de leurs propres méta-données, en particulier l'expression des dépendances. Ainsi, les dépendances à l'exécution pour une application donnée sont exprimées à la fois dans son script `rc` de démarrage et dans son paquetage d'installation (les dépendances doivent être installées avant de pouvoir être démarrées), avec des syntaxes différentes. Il y a donc duplication d'efforts et risque de contradiction.

De même, pour la partie multi-acteurs, plusieurs outils disponibles proposent de l'isolation de nommage ou de l'isolation de ressources. On trouve en particulier des outils de virtualisation et de contrôle de ressources.

Il apparaît que les multiples activités de gestion sur une passerelle domestique utilisant un système GNU/Linux sont implantées par de nombreuses couches logicielles. Chacune a sa propre syntaxe en ligne de commande, utilise son propre formalisme de journalisation et/ou d'événements, et communique de façon non universelle par des signaux POSIX, des *sockets* Unix ou des *pipes*. Par conséquent, et pour simplifier la tâche de gestionnaires locaux et distants, il devient nécessaire de consolider tous ces applicatifs derrière une interface de gestion unifiée, comme le fait par exemple le shell Felix pour Java/OSGi.

5.1.2 Outils existants et leurs limites

Parmi les outils existants, nous retiendrons trois catégories. Premièrement, les gestionnaires de paquetages correspondent aux boîtes « déploiement » et « dépôts de services ». Deuxièmement, `init`, `rc` et leurs dérivés correspondent aux boîtes « cycle de vie » et « profils ». Troisièmement, les outils d'isolation, de virtualisation ou de contrôle de ressources implantent les boîtes « multi-utilisateurs » et « facturation ».

Nous avons présenté ces outils en section 2.3.1 page 19 ; nous étudions ici leurs limites par rapport au modèle d'environnement d'exécution multi-services.

5.1.2.1 Gestionnaires de paquetages

Les gestionnaires de paquetages GNU/Linux vont du plus simple (Slackware) au plus complet (Debian). Sur Slackware, il permet d'installer, supprimer et remplacer (mettre à jour) un paquetage. C'est à l'administrateur de gérer manuellement les dépendances et la sauvegarde des fichiers de configuration. Sur Debian, `dpkg` effectue les mêmes tâches, mais, en plus, vérifie que les dépendances sont installées et rapatrie le fichier archive du paquetage voulu depuis

un dépôt distant. Des surcouches à `dpkg`, comme `apt`, se chargent en plus de rapatrier et installer les dépendances manquantes. A l'installation d'un paquetage Debian, un script `rc` peut également être installé et démarré.

Un point commun à tous ces gestionnaires de paquetages est d'exécuter des scripts de configuration avant ou après les phases d'installation ou de désinstallation. Cependant, ils diffèrent dans le langage de ces scripts. Pour Debian, ils peuvent par exemple être écrits en Bash ou en Perl, ce qui crée des dépendances additionnelles selon le paquetage. Pour Slackware, tous ces scripts sont écrits en Ash, le shell contenu dans Busybox. Quel que soit le paquetage, le gestionnaire de paquetages Slackware ne dépend ainsi que de Busybox [128], un exécutable combinant une version légère de la plupart des outils GNU usuels (*coreutils*).

Tous ces gestionnaires de paquetages souffrent également d'une limitation commune vis-à-vis du modèle multi-services : ils sont mono-utilisateur. Plus exactement, ils permettent à un administrateur unique d'installer des paquetages globalement sur le système. Un environnement multi-services doit permettre à plusieurs acteurs économiques d'installer des paquetages sans interférer les uns avec les autres.

5.1.2.2 init et rc

Lorsque l'on démarre un PC, plusieurs actions s'enchaînent, que nous allons ici simplifier.

- Tout d'abord, le BIOS est exécuté depuis un endroit connu sur la carte mère. Le BIOS détermine quel périphérique de démarrage utiliser : un disque dur, un support externe, ou encore PXE (*Preboot Execution Environment*) depuis le réseau ;
- Un périphérique de démarrage de type disque dur a un secteur connu (le premier) appelé *Master Boot Record*. Le MBR contient le chargeur de démarrage dit *stage 1*, qui est alors chargé en mémoire et qui pointe vers le chargeur de démarrage *stage 2* ;
- Ce chargeur de démarrage stage 2 (LILO ou GRUB pour GNU/Linux) peut donc occuper une taille quelconque sur le disque, à un emplacement variable, non connu à l'avance par le BIOS. Sa tâche principale est de décompresser et charger le noyau en mémoire ;
- Une fois chargé, le noyau lance une application.

Cette application est usuellement `/sbin/init`, et peut être changée via un paramètre à LILO ou GRUB. `init` est un démon dont l'arrêt signifie l'arrêt de la machine. Il a l'identifiant PID 1. Lors de son lancement, il charge des modules du noyau et exécute des processus : `getty` pour les terminaux virtuels, des démons pour `cron`, `dhcp` ou encore `acpi`, un serveur `X`, etc. Chacun de ces processus dispose d'un script `rc`, qui permet de le démarrer et de l'arrêter avec des options et des variables d'environnement choisies. `rc` est le nom du programme utilisé par `init` pour démarrer ou arrêter ces scripts.

L'ensemble des scripts `rc` à lancer au démarrage est regroupé sous la forme d'un *runlevel*, ou profil. Un *runlevel* associe un nom ou un numéro à une liste de scripts `rc`, et a un but

fonctionnel précis : fonctionnement avec un seul utilisateur, en multi-utilisateurs, en mode texte, en mode graphique. Il est possible de choisir un *runlevel* au démarrage du système, et d'en changer en cours d'utilisation via `rc`. On parle alors de *configuration logicielle* du système.

Évolution de init et rc Il existe de nombreuses implantations de `init` et de `rc` [129]. La version la plus utilisée est celle de System V. Certaines distributions la modifient pour obtenir un grain plus fin dans la gestion du cycle de vie des applications : gestion des dépendances (p.ex. le service DHCP a besoin du service réseau), mise en pause, redémarrage automatique (mode *respawn* ou résurrection). D'autres ajoutent des outils pour plus facilement obtenir l'état d'un script `rc` ou changer de *runlevel*. Enfin, des acteurs importants du monde Unix repensent `init` d'un point de vue conceptuel. Dans MacOS X 10.4, Apple propose que `launchd` remplace tous les outils lançant des applications, comme `init`, `inet`, `cron` et `anacron`. Avec `upstart`, Ubuntu propose en plus de s'interfacer avec les gestionnaires d'énergie (ACPI) et de périphériques (udev). Ainsi, le passage d'un portable en mode batterie, le branchement d'un périphérique ou le lancement d'une application au démarrage deviennent des événements, auxquels il est possible de greffer des actions, comme lancer une autre application. Les dépendances entre scripts `rc` sont donc traités comme l'attente d'un événement. Finalement, Sun propose SMF (*Service Management Facility*) sur Solaris 10. L'objectif est d'améliorer la détection de fautes et leur diagnostic. En cas de besoin, les services peuvent fonctionner en mode dégradé ou en mode maintenance, dans le but de maximiser le taux de fonctionnement (*uptime*) du système.

En conclusion, `init` et `rc` entrent en jeu non seulement au démarrage de la machine, mais aussi pendant toute sa durée de fonctionnement. Ils gèrent le cycle de vie d'applications, ainsi que leur organisation en profils. Leurs limitations par rapport au modèle multi-services sont doubles. D'une part ils sont décorrélés des gestionnaires de paquetages, ce qui implique une dispersion voire une duplication des méta-données exprimant les dépendances. D'autre part, tout comme les gestionnaires de paquetages, ils fonctionnent avec une logique mono-administrateur : un utilisateur système doit avoir les droits `root` pour commander les scripts `rc`.

5.1.2.3 Mécanismes d'isolation entre acteurs économiques

Des mécanismes d'isolation présentés en section 2.5 page 32, nous retenons deux familles de solutions pour GNU/Linux : les quotas et les machines virtuelles. On distingue les quotas par processus des quotas par utilisateur ou groupe d'utilisateurs. Les quotas par processus, exprimés par une structure POSIX `rlimit`, fixent entre autres :

- La taille maximale de la mémoire virtuelle allouée au processus ;

- Une limite de temps processeur en secondes ;
- La taille maximale des fichiers que le processus peut créer ;
- Une priorité maximale (ou *niceness* minimale) dans l'ordonnanceur ;
- La taille maximale de la pile mémoire du processus.

Les limites par processus sont héritées lors d'appels à `fork` et `execve`. Le support de ces limites est intégré au noyau du système d'exploitation. Des outils additionnels, usuellement dans le paquetage `pam` ou `shadow`, permettent de fixer des quotas par utilisateur, ou par groupe d'utilisateur. Parmi ces quotas, on trouve :

- Une limite maximale d'espace disque sur un système de fichiers ;
- Un nombre maximal de processus ;
- Un nombre maximal de fichiers ouverts.

L'intérêt des quotas pour le modèle multi-services est de s'intégrer avec la notion d'utilisateur Unix et les permissions associées. On peut ainsi obtenir une isolation de nommage entre utilisateurs système, donc entre acteurs économiques. L'inconvénient est que configurer correctement ces quotas est relativement complexe et sujet à erreur de la part de l'administrateur.

Certaines techniques de virtualisation en revanche, comme Xen ou chroot, disposent d'outils facilitant la création de machines virtuelles. Il est alors aisé d'ajouter et supprimer des utilisateurs sur le système. Cependant, le désavantage de la virtualisation par rapport aux outils de quotas est qu'un ensemble de données, sur le disque et en mémoire, sont répliqués pour chaque machine virtuelle. Le passage à l'échelle est alors un critère à surveiller.

5.2 Proposition : HGL

Notre proposition se nomme HGL, pour *Home Gateway Linux*. Dans HGL, les unités de déploiement sont appelées des *bundles*, pour rappeler la terminologie OSGi. Un *bundle* HGL est une archive qui contient des exécutables, des librairies ou tout autre type de fichiers, ainsi qu'un activateur facultatif. Une application « bundelisée » a son script `rc` ou son exécutable principal pour activateur ; une librairie bundelisée n'en a pas. L'objectif est de centraliser toutes les activités de rapatriement, installation et cycle de vie au sein d'une seule unité de déploiement. Pour résumer, un *bundle* HGL est un *package* GNU/Linux augmenté d'un activateur et de méta-données, en particulier les dépendances au démarrage.

5.2.1 Architecture

La conception de HGL est représentée en figure 5.2. Une unique interface de gestion donne accès à la fois aux activités de gestion du déploiement et des utilisateurs système. Ces deux familles d'activités disposent chacune d'un registre, qui reflète l'état du système.

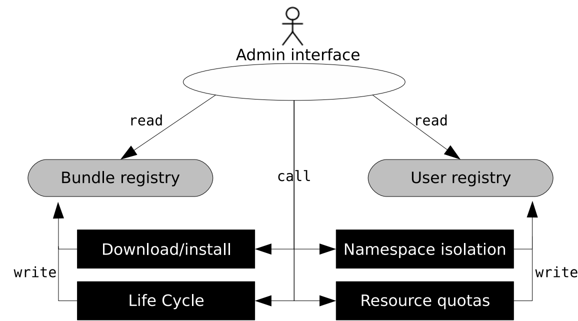


Fig. 5.2: HGL : architecture multi-services pour GNU/Linux

Chaque *coin* utilisateur dispose d'un registre des *bundles* décrit la liste des *bundles* présents et leur état à l'instant courant : installé, démarré, en pause, arrêté. De même, pour l'utilisateur `root` uniquement, un registre des utilisateurs décrit les utilisateurs existant sur la passerelle domestique, ainsi que ceux dont le *coin* est lancé à l'instant courant.

Les boîtes de couleur noire sur la figure 5.2 peuvent être implantées de multiples façons, à partir des outils présentés précédemment. Ce choix se fait en fonction des besoins en performance, simplicité ou exhaustivité des fonctionnalités ; nous le faisons dans la section suivante présentant notre implantation.

5.2.2 Aspects déploiement

Les boîtes en noir sur le côté gauche de la figure 5.2 concernent les aspects déploiement de la gestion. Elles sont donc centrées autour des *bundles*. La figure 5.3 montre le diagramme d'état que suivent les *bundles* localement sur la passerelle domestique.

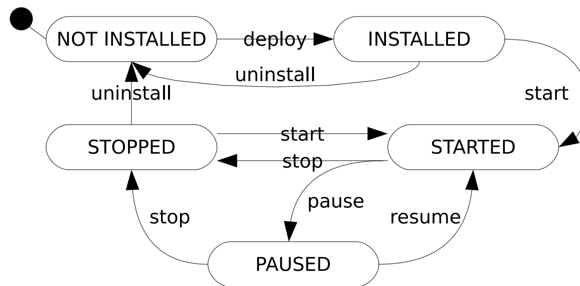


Fig. 5.3: HGL : diagramme d'état pour un bundle

La gestion du cycle de vie de *bundles* dans GNU/Linux peut se faire soit par le biais de scripts `rc`, soit avec des signaux POSIX. Dans la solution à base de scripts `rc`, chaque script prend en paramètre des commandes comme `start` ou `stop` ; l'activateur du *bundle* est le script lui-même. Dans la solution à base de signaux, l'activateur est un exécutable à lancer, qui écoute et traite les signaux comme SIGSTOP et SIGTERM.

Les scripts rc sont les plus souples d'utilisation : ils séparent la gestion du cycle de vie et l'application elle-même. Un langage de type shell autorise la configuration de variables d'environnement ou encore le lancement de vérifications avant de démarrer l'application. Á l'inverse, une solution à base de signaux POSIX impose à chaque *bundle* d'être à l'écoute de ces signaux pour terminer proprement, ce qui est intrusif dans les applications elles-mêmes. En revanche, les signaux POSIX ont l'avantage d'être immédiats, ce qui fait que le registre de *bundles* est constamment à jour. Dans le cas des scripts rc, la commande start lance généralement un exécutable en arrière-plan et sauvegarde son identifiant (PID) dans un fichier /var/run/le_nom.pid. Pour savoir si le *bundle* est toujours en cours d'exécution ou non, il faut lire ce fichier, récupérer le PID et demander au système d'exploitation si ce PID est utilisé (/proc/le_PID/). Le registre de *bundles* doit donc faire cette vérification périodiquement pour se maintenir à jour, avec pour imprécision la période choisie.

5.2.3 Aspects multi-acteurs

Les boîtes en noir sur le côté droit de la figure 5.2 concernent les aspects multi-acteurs de la gestion. Elles sont uniquement accessibles à l'opérateur de la passerelle domestique, qui est similaire à l'utilisateur root. Celui-ci est chargé d'autoriser ou non de nouveaux fournisseurs de services sur la passerelle domestique. Chaque fournisseur de service accède à son propre *coin*, où il a les droits suffisants pour déployer ses *bundles* et gérer leur cycle de vie.

La vue en couches correspondante est donnée en figure 5.4. Notons qu'elle est conforme à celle, indépendante de toute implantation, présentée au chapitre 3 (figure 3.3 page 53).

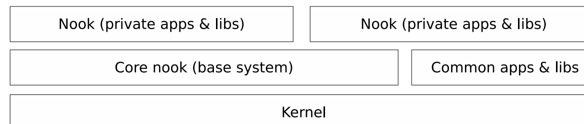


Fig. 5.4: HGL : vue en couches logicielles

La figure 5.5 représente opérateur et fournisseurs sur la passerelle domestique, avec chacun leur interface de gestion, leur registre et leur jeu de fonctionnalités de gestion.

Chaque fournisseur de services est un utilisateur Unix, qui accède via son interface de gestion personnelle aux activités consolidées de gestion du déploiement. Sur le même principe que l'implantation en Java/OSGi présentée au chapitre 4, chaque fournisseur de services est libre de choisir sa technologie de gestion, sous réserve d'implanter l'interface de gestion conformément à ce modèle.

Les différentes implantations possibles de ces vues conditionnent sur quelle partie du modèle économique multi-services l'on veut mettre l'accent. En particulier, nous devons choisir quel type d'isolation adopter : une isolation forte, telle que VMware, fournit à la fois une isolation

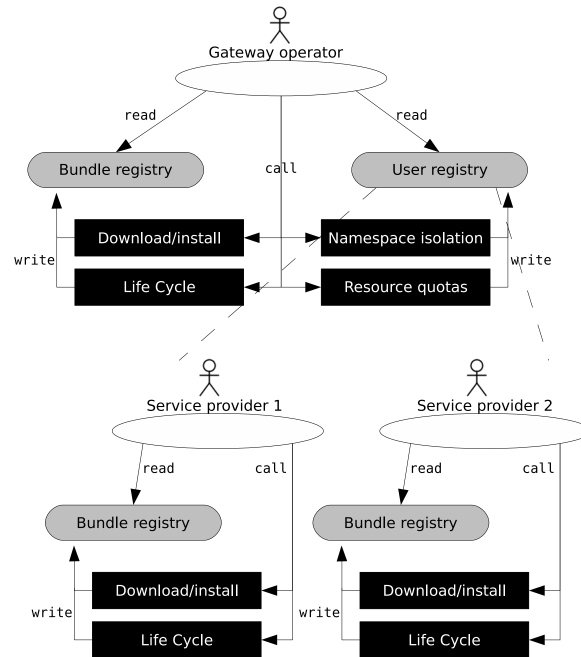


Fig. 5.5: HGL : vue gestion des acteurs en présence

de nommage et de ressources entre utilisateurs, mais ne leur permet pas de partager des services directement. Á l'inverse, une isolation faible telle que `chroot` n'offre pas d'isolation de ressources, mais permet ce partage de services plus facilement. Ces choix d'implantation sont l'objet de la section suivante.

5.3 Implantation

Nous avons implanté HGL en langage C, en réutilisant des applicatifs GNU/Linux aux endroits opportuns. Nous présentons ici nos choix d'implantation pour chacune des familles de fonctionnalités de gestion.

5.3.1 Aspects déploiement

5.3.1.1 Gestionnaire de paquetages

L'implantation actuelle de HGL utilise le gestionnaire de paquetages de la distribution Slackware. La raison principale est qu'il a très peu de dépendances (`tar`, `gzip`) et que celles-ci font partie de `busybox`, une implantation des outils GNU les plus communs pour environnements à ressources réduites. Le gestionnaire de paquetages Slackware lui-même est un ensemble de scripts shell, plus exactement du shell `ash`, lui aussi partie intégrante de `busybox`.

Le gestionnaire de paquetages Slackware est chargé d'installer, désinstaller et mettre à

jour les *bundles* ; le rapatriement et la gestion des dépendances sont effectués par HGL. Par conséquent, l'implantation des *bundles* dans HGL est très proche de celle des paquetages Slackware : il s'agit d'archives `tar.gz` contenant les fichiers à installer, un fichier de description, et éventuellement des scripts pré- et post-installation et désinstallation. Les méta-données HGL supplémentaires, comme les dépendances et l'activateur, sont actuellement inscrites sur le dépôt de *bundles*.

5.3.1.2 Dépôts de bundles

Un dépôt de *bundles* est représenté par un fichier texte. Dans OSGi, un dépôt OBR est un fichier XML qui contient, pour chaque *bundle* répertorié, son nom, sa version, son URI, sa taille, ses dépendances, et d'autres méta-données comme une description textuelle et un copyright. Dans Slackware, un dépôt est un fichier PACKAGES.TXT en texte clair, qui contient pour chaque paquetage son nom, son URL, sa taille et sa description. Pour ces deux exemples, le fichier texte représentant le dépôt est généré automatiquement à partir des méta-données contenues dans les *bundles* ou des paquetages. Seule la taille est obtenue à partir des fichiers `.jar` ou `.tgz` eux-mêmes.

HGL réutilise les paquetages Slackware et décrit un dépôt par un fichier BUNDLES.TXT en texte clair. Par rapport à PACKAGES.TXT, ce fichier contient en plus les dépendances et l'activateur de chaque *bundle*. Dans l'implantation actuelle, ces méta-données supplémentaires existent sur le dépôt mais pas dans les paquetages Slackware, pour faciliter la réutilisation de paquetages existants.

5.3.2 Cycle de vie

Nous avons choisi de gérer le cycle de vie des *bundles* par signaux POSIX. Les avantages par rapport aux scripts `rc`, outre le temps de réaction immédiat, sont la simplicité de mise en œuvre et l'uniformité de la solution pour chaque *bundle*. L'inconvénient est que les options et pré-traitements que l'on peut appliquer au lancement d'une application sont moins flexibles, et doivent être gérés par l'activateur du *bundle* plutôt que par un script séparé. L'activateur de chaque *bundle* est alors un processus, qu'il faut lancer puis terminer pour respectivement démarrer et arrêter le *bundle*.

Un processus `hgl_init` est chargé de lancer les activateurs des *bundles* à démarrer. Dans la hiérarchie des processus Unix, `hgl_init` est donc le père des processus associés aux *bundles*. Il est ainsi notifié de la mort d'un processus fils, donc du passage d'un *bundle* à l'état STOPPED ; il peut également commander l'arrêt, la mise en pause et la continuation d'un *bundle* via les signaux SIGTERM, SIGSTOP et SIGCONT, respectivement. Le nom `hgl_init` est fait pour rappeler celui d'`init`, avec lequel il partage la particularité de lancer et gérer d'autres processus.

Notons que ce fonctionnement a un inconvénient : l'activateur d'un *bundle* ne doit pas se détacher de sa console (c'est-à-dire passer en mode démon) de lui-même, sans quoi HGL ne sera plus en mesure de gérer son cycle de vie. Ce défaut mineur est cependant compensé par la simplicité de la solution et le caractère immédiat des signaux POSIX.

5.3.3 Aspects multi-acteurs

5.3.3.1 Coins utilisateur

Nous avons choisi d'implémenter les *coins* utilisateur par des prisons `chroot`. L'objectif est d'obtenir une isolation de nommage sur le système de fichiers qui n'implique qu'un surcoût minime sur la consommation des ressources de la passerelle domestique, et qui en même temps autorise de partager des services de l'opérateur de la passerelle vers les fournisseurs de services.

Notons que ceci permet à chaque utilisateur d'administrer son propre *coin* sans avoir besoin des droits `root`. Par exemple, chacun est libre d'installer ou modifier des fichiers dans son propre répertoire `$CHROOTDIR/bin/` ou `$CHROOTDIR/usr/bin/`. Le seul acteur ayant les droits `root` sur la passerelle domestique est l'opérateur.

D'un point de vue processus, la hiérarchie correspondante est représentée en figure 5.6. Le processus `hgl_init` de l'utilisateur `root` crée et gère les applications de `root` et les applications partagées ; de même, il crée et gère un processus `hgl_init` par utilisateur de la passerelle domestique. Chaque coin utilisateur a donc un processus `hgl_init` dédié, qui gère les *bundles* de ce coin particulier.

```
hgl_init (root)
 |
 |- application root
 |
 |- application partagée
 |
 |- hgl_init (coin 1)
 |    |
 |    |- application utilisateur 1a
 |    `- application utilisateur 1b
 |
 `- hgl_init (coin 2)
      |
      `- application utilisateur
```

FIG. 5.6: HGL : vue processus des coins utilisateur

5.3.3.2 Partage de services

Le partage de services se fait par le biais de répertoires partagés. L'opérateur de la passerelle installe les exécutables et les librairies à partager dans des répertoires spécifiques, actuellement `/bin/` et `/lib/`. Tous appartiennent à l'utilisateur `root` dans le groupe `root` ; les exécutables ont les droits 755 et les librairies 644. Ces répertoires sont montés dans chaque environnement `chroot`, sous `$CHROOTDIR/bin/` et `$CHROOTDIR/lib/`, via un appel à `mount` avec l'option MS_BIND. Ainsi, chaque utilisateur voit son répertoire `$CHROOTDIR/` comme le répertoire racine `/`, et a tous les droits sur tous les répertoires et fichiers qu'il contient, à l'exception des répertoires partagés qu'il ne peut pas modifier. Typiquement, un fournisseur de services installe ses exécutables dans `/usr/bin/` et ses librairies dans `/usr/lib`.

5.3.4 Interface de gestion

L'interface de gestion est implantée via une *socket*, qui reste à l'écoute de commandes extérieures tant que tourne le processus `hgl_init`. Les commandes sont les suivantes :

```
     **** Available commands ****
list                          List bundles in the repository.
install     <bundle-name>     Download and install a bundle.
uninstall   <bundle-name>     Uninstall a bundle.
ps                            List bundles in the registry. Extra param = verbose.
start       <bundle-name>     Start a bundle's activator.
stop        <bundle-name>     Stop a running bundle activator.
pause       <bundle-name>     Pause a started bundle activator.
resume      <bundle-name>     Resume a paused bundle activator.
<newline>                     Close connection.
help                          Display this help.

users                         List existing users.
useradd     <nook-name>       Create a nook (unix user, home directory, port, etc).
userdel     <nook-name>       Delete a nook (unix user, home directory and all).
userstart   <nook-name>       Start a nook (hgl_init process).
userstop    <nook-name>       Stop a nook  (hgl_init process).
```

Listing 5.1: HGL : commandes pour l'interface de gestion

Chaque commande appelle la fonction HGL correspondante ou un outil externe. Dans l'implantation actuelle, l'interface de gestion utilise ces outils :

- Rapatriement de *bundles* : `wget` et `cp`, selon le schéma de l'URL de téléchargement (`http://` ou `file://`) ;
- Installation et désinstallation de *bundles* : `installpkg` et `removepkg` (outils du gestionnaire de paquetages Slackware) ;
- Liste des *bundles* installés ou démarrés : consultation interne du registre des *bundles* HGL ;

- Création et suppression de coins utilisateur : `useradd` et `userdel` pour créer les utilisateurs Unix ;
- Liste des utilisateurs :consultation interne du registre des *bundles* HGL ;
- Démarrage et arrêt de *bundles* ou de coins utilisateur : appels à `fork` et `exec`, signaux POSIX.

La *socket* de l'interface de gestion peut être attaquée soit directement (via `telnet`), soit en passant par un connecteur (p.ex. TR-069 ou SNMP) vers un serveur d'auto-configuration ou autre logiciel de gestion existant. Par défaut, la *socket* écoute sur le numéro de port TCP qui correspond à l'identifiant utilisateur (UID) associé au *coin* utilisateur.

5.4 Expérimentations

5.4.1 Intégration avec TR-069

L'interface de gestion étant techno-agnostique, elle peut s'intégrer avec n'importe quel agent de gestion. En particulier, HGL est interfacé avec un agent TR-069, pour un démonstrateur dans les locaux d'Alcatel-Lucent.

5.4.2 Mesures de performance

5.4.2.1 Mémoire

Le tableau 5.1 indique l'empreinte mémoire du processus `hgl_init`. Il est dynamiquement lié à `glibc` version 2.5 ; les librairies C plus petites comme `uclibc` ou `dietlibc` restent à tester. Pour vérifier que le code est indépendant d'un compilateur particulier, HGL a été compilé avec celui de GNU (`gcc` version 4.1) et celui d'Intel (`icc` version 9).

	0 *coins*		> 0 *coins*	
	HGL	HGL + libs partagées	HGL	HGL + libs partagées
`gcc -Os`	256 Ko	1560 Ko	296 Ko	1760 Ko
`icc -Os`	284 Ko	1780 Ko	324 Ko	1980 Ko

TAB. 5.1: HGL : utilisation mémoire du processus `hgl_init`

Les librairies partagées utilisées sont `glibc-2.5` et `ld-2.5`. Cependant, le code créant des *coins* utilisateur utilise des fonctions liées à la création d'utilisateurs Unix, et fait appel à NSS (*Name Service Switch*). Lorsque le premier *coin* utilisateur est créé, c'est-à-dire lorsque l'opérateur envoie la première commande `useradd`, les librairies partagées `libnss_compat-2.5`, `libnsl-2.5`, `libnss_nis-2.5` et `libnss_files-2.5` sont chargées, d'où la distinction dans le tableau de mesures.

Lorsque l'opérateur crée un nouveau *coin* utilisateur, le registre des utilisateurs grossit de quelques octets, pour un impact négligeable sur ces mesures. Le réel impact par contre est la création d'un nouveau processus `hgl_init` en mémoire. Puisque les *coins* utilisateur non `root` n'ont pas de registre des utilisateurs, leur processus `hgl_init` a la taille mémoire indiquée dans la colonne de gauche.

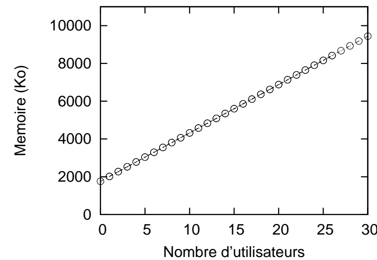


Fig. 5.7: HGL : consommation mémoire

La figure 5.7 représente la consommation mémoire conjointe de tous les processus `hgl_init` en fonction du nombre d'utilisateurs du système. Si l'on ajoute à chaque *coin* utilisateur un *bundle* qui consomme 1 Mo de mémoire, une passerelle domestique dotée de 64 Mo de mémoire supporte plus de 45 utilisateurs simultanés.

5.4.2.2 Disque

Le tableau 5.2 donne l'espace disque requis par l'exécutable `hgl_init`. Ce fichier n'est pas répliqué dans chaque *coin* utilisateur, mais est partagé ; son coût n'est donc payé que par le *coin* principal.

	Dynamiquement lié à glibc 2.5
gcc -Os	36 637 octets
icc -Os	46 449 octets

Tab. 5.2: HGL : taille disque du binaire

Chaque *coin* utilisateur nouvellement créé, c'est-à-dire sans *bundle* installé, utilise moins d'une vingtaine d'inodes. Le coût d'un *coin* utilisateur en espace disque est donc négligeable.

5.4.2.3 Analyse des résultats

Le facteur limitant le nombre maximum d'utilisateurs du système est la consommation mémoire. Les autres facteurs, à savoir l'espace disque, le nombre maximum d'utilisateurs Unix, le nombre maximum de *sockets* ouvertes pour les agents de gestion et le nombre maximum de processus simultanés restent au moins deux ordres de grandeur plus importants que le facteur

mémoire. Le nombre maximum d'utilisateurs simultanés sur une passerelle domestique dotée de 64 Mo de mémoire est donc près de 250 pour des *coins* sans aucun *bundle*, et d'environ 48 pour des *coins* hébergeant chacun un *bundle* qui consomme 1 Mo de mémoire. Notons que ces chiffres ne prennent pas en compte les agents de gestion.

5.5 Discussion

Dans ce chapitre, nous avons proposé un style d'implantation, basé sur des outils GNU/Linux, pour le modèle d'environnement d'exécution présenté au chapitre 3. Nous avons fait des choix d'implantation et évalué leur comportement en termes de performances. Nous discutons ici de ces choix, d'améliorations possibles et de fonctionnalités à compléter.

5.5.1 Fonctionnalités

Agents de gestion Pour chaque *coin* utilisateur, HGL propose une interface de gestion du déploiement des *bundles*. Pour le *coin* administrateur, cette interface inclut de plus le déploiement des *coins* utilisateur eux-mêmes. Les interfaces de gestion sont publiées via une *socket* TCP, et peuvent donc être manipulées localement ou à distance. Notre objectif était d'installer un agent TR-069 par *coin* utilisateur, pour faire le lien entre la *socket* de HGL et un serveur d'auto-configuration (ACS) TR-069, qui est en passe de devenir la technologie standard pour gérer des passerelles domestiques.

Nous avons développé HGL durant une collaboration avec Alcatel-Lucent. Nous avons pu tester que HGL s'intègre avec un agent et un ACS TR-069 propriétaires à Alcatel-Lucent. Nous avons vérifié que le déploiement d'un *bundle* HGL peut être piloté depuis l'ACS vers un routeur Wifi Asus WL-520g. Cependant, l'agent en question est développé en Java, et nécessite d'installer une machine virtuelle Java. Or l'un des objectifs de HGL est de se passer de la JVM et de n'exécuter que du code natif voire du shell. Nous n'avons donc pas relevé de mesures de performances avec l'agent TR-069. Pour le faire de manière juste, il nous faut plutôt un agent TR-069 écrit en C ; nous n'en avons trouvé aucun librement utilisable durant nos travaux.

Profils et versions HGL ne gère pas encore les numéros de version de *bundles*, ni leur organisation en profils. Supporter les numéros de version, en particulier dans l'expression et dans la résolution des dépendances, est un effort d'implantation. Par contre, intégrer la notion de profil peut avoir un impact au niveau de la conception de HGL. L'idée est d'associer un nom de profil à une liste de *bundles*, chacun dans un état de cycle de vie donné. Une possibilité est d'étendre la notion de *runlevel* Unix pour prendre en compte cette diversité des états. Une extension à approfondir est d'écouter divers événements en provenance de l'environnement

d'exécution, pour le cas échéant commander un changement de profil. Les événements peuvent provenir notamment d'ACPI (*Advanced Configuration and Power Interface*) ou de la couche d'abstraction du matériel (HAL, *Hardware Abstraction Layer*), déclenchés par exemple lors de l'apparition ou disparition de périphériques sur la passerelle domestique.

5.5.2 Choix d'implantation

Gestionnaire de paquetages Pour installer des *bundles*, HGL fait appel à un gestionnaire de paquetages GNU/Linux. L'implantation actuelle utilise celui de Slackware, parce qu'il est simple et ne dépend que de la boîte à outils `busybox`. Parmi ses concurrents, il faut relever `ipkg`, une variante de `apt` de Debian utilisé dans les distributions comme OpenWRT dédiées aux routeurs Wifi. Un des intérêts de `ipkg` pour HGL serait de fournir des diagnostics plus précis en cas d'erreur lors de l'installation ou désinstallation d'un *bundle*, au prix de dépendances supplémentaires.

Isolation Le mécanisme d'isolation actuel utilise des processus `hgl_init` et des prisons `chroot` pour fournir une isolation de nommage. Si l'on veut obtenir une isolation plus forte, il faut mettre en place une isolation de ressources, soit par une technique de virtualisation, soit en ajoutant du contrôle de ressources à l'implantation existante.

Dans de futurs travaux, nous souhaitons tester des techniques de virtualisation sur des passerelles domestiques. Leur point sensible est le passage à l'échelle, en particulier en termes de consommation mémoire par machine virtuelle, mais aussi et surtout en termes d'espace disque. En effet, la tendance actuelle dans les passerelles domestiques est que la mémoire, à base de SDRAM, revient moins chère que l'espace de stockage, à base de mémoire Flash. Ainsi, les équipements récents sont dotés de 64 Mo de mémoire contre 16 Mo de stockage seulement. Il convient donc de surveiller de près la consommation disque, et d'éviter la duplication de fichiers entre *coins* utilisateur. Dans l'implantation actuelle, le partage de répertoires via `mount -bind` sert à cela. Les techniques de virtualisation que nous voulons tester sont, selon les critères de légèreté donnés en figure 2.8 page 41, en priorité celles à base de conteneurs, comme VServer ou OpenVZ.

Une autre option est d'utiliser des outils comme `quota` ou `rlimits` pour imposer des limites maximum d'utilisation de ressources par *coin* utilisateur. Ces limites sont mises en place par le système d'exploitation soit par utilisateur Unix, soit par processus. Ces techniques sont donc compatibles avec l'implantation actuelle, qui fait correspondre chaque *coin* utilisateur à un utilisateur Unix et à un processus père `hgl_init`.

5.5.3 Partage de services dans HGL

Dans l'implantation actuelle de HGL, les *coins* utilisateur sont implantés via des prisons `chroot`. Le partage de services se fait en rendant visibles aux *coins* utilisateur certains répertoires appartenant au *coin* administrateur. Ainsi, les répertoires `/bin/` et `/lib/` de l'administrateur sont accessibles en lecture seule depuis les `chroot` utilisateur. Ce mécanisme est relativement simple, et prend en compte l'apparition et la disparition de fichiers dans `/bin/` et `/lib/` durant le fonctionnement de la passerelle domestique. Son gros désavantage est d'être lié à des fichiers, donc aux dépendances réelles des services, mais sans être explicitement lié aux *bundles* HGL correspondant à ces services, donc aux dépendances exprimées. Une solution à ce problème est d'ajouter des communications IPC entre le registre du *coin* administrateur et le registre des *coins* utilisateur, pour notifier des changements dans le cycle de déploiement des *bundles* partagés.

5.5.4 Modèle de programmation : services et méta-données

Dans GNU/Linux, le « registre » qui liste les services installés et leur état est réparti en plusieurs lieux : le gestionnaire de paquetages, de multiples entrées dans `/proc` et `/var`, ou encore des variables d'environnement. Assurer la cohérence de tous ces éléments est une tâche hasardeuse : si un shell de commande est ouvert, il est très facile de supprimer un fichier ou de tuer un processus pour rendre obsolètes les informations du registre sur un paquetage ou sur un script `rc`. Dans HGL, nous avons choisi d'utiliser un registre unique qui centralise toutes les informations sur les *bundles* HGL, particulièrement leur cycle de vie et leurs dépendances. Nous évitons ainsi l'éclatement des méta-informations et leur possible contradiction. Pour éviter les incohérences entre le registre HGL et le système de fichiers, les accès distants à un shell de commande, du type SSH, sont à proscrire. Toute action de gestion ou d'administration doit impérativement passer par l'interface de gestion du *coin* utilisateur concerné.

6 Synthèse

6.1 Bilan des implantations . . . 93
6.2 Bilan des styles d'implantation . . . 94
6.2.1 Différences de performances . . . 94
6.2.2 Différences de fonctionnalités . . . 95
6.2.3 Dépendances exprimées et dépendances réelles . . . 97
6.2.4 Tableau comparatif . . . 100
6.3 Autres styles d'implantation possibles . . . 101
6.3.1 GNU/Linux et SOA . . . 101
6.3.2 GNU/Linux et SOP . . . 102
6.4 Conclusion . . . 103

6.1 Bilan des implantations

Nous avons vu, au premier chapitre de ce document, l'évolution du modèle économique de la fourniture de services au domicile. En particulier, il faut retenir que si le marché se diversifie en terme de fournisseurs de services multimédia et données, il doit également se diversifier en gamme de services, en intégrant par exemple la domotique et l'aide aux personnes âgées. Nous appelons ce modèle économique *multi-services*. Dans le chapitre 2, nous nous sommes focalisés sur la passerelle domestique, qui relie le réseau domestique au réseau de l'opérateur. Cet équipement a le double emploi d'environnement d'exécution, pour héberger des services, et de passerelle de gestion, pour les déployer et les superviser. Ces deux aspects, appliqués au modèle économique multi-services, nous ont mené à exprimer les besoins des passerelles multi-services dans le chapitre 3. L'aspect environnement d'exécution, en particulier, est composé d'activités de déploiement et d'isolation. Le déploiement regroupe le rapatriement de services sous forme d'unités de déploiement, leur installation et la gestion de leur cycle de vie. L'isolation assure que les multiples acteurs économiques cohabitent en ne gérant que leur propre *coin* sur la passerelle domestique. Nous avons représenté ces activités sous forme de briques fonctionnelles, centrées sur les services ou sur les acteurs économiques.

A partir de l'expression des besoins en déploiement, en isolation et en gestion des passerelles multi-services, nous avons proposé deux styles d'implantation : l'une avec une technologie à base de machine virtuelle, en l'occurrence Java/OSGi, et l'autre avec un système d'exploitation du type POSIX, en l'occurrence GNU/Linux. Java/OSGi a l'avantage de déjà proposer une gestion intégrée et cohérente des briques fonctionnelles ; en revanche, seule une partie des briques demandées existe. Notre travail a donc consisté à fournir ces briques manquantes. Dans le cas de GNU/Linux, presque toutes les briques fonctionnelles existent ; le problème est qu'elles sont implantées par une multitude d'outils, qui ne sont pas intégrés. Notre travail a donc consisté à donner une cohérence aux briques existantes.

Dans ce dernier chapitre, nous faisons tout d'abord le point sur les performances obtenues par les deux implantations. Nous comparons ensuite les deux styles d'implantation par rapport au modèle exprimé au chapitre 3, en explicitant leurs différences dans la réalisation de fonctionnalités similaires. Nous étudions également d'autres styles d'implantation possibles, représentant d'autres choix que ceux que nous avons fait, et analysons leurs avantages potentiels et leur coût. Enfin, nous nous abstrayons des styles d'implantation, et concluons ce document en faisant un bilan des aspects importants pour les environnements multi-services.

6.2 Bilan des styles d'implantation

Nous proposons deux implantations d'un environnement d'exécution multi-services, qui diffèrent en termes de performance et de fonctionnalités. Mais, de manière plus importante, elles correspondent à deux styles d'implantation différents : l'un est basé sur une machine virtuelle et un modèle de programmation à composants et services, et l'autre est basé sur un système d'exploitation de type POSIX. Outre leurs performances globales, ces deux styles diffèrent en termes de cycle de vie des unités de déploiement, d'expression et de résolution des dépendances, de partage de services entre *coins* utilisateurs, de modèle de programmation, ou encore de capacités de gestion. Nous explorons ici ces différences, tant dues au développement de VOSGi/MOSGi et HGL qu'à leur conception.

6.2.1 Différences de performances

Pour qualifier la performance d'un environnement d'exécution multi-services, nous observons le nombre d'acteurs économiques pouvant simultanément l'utiliser, c'est-à-dire le nombre maximal de *coins* utilisateur simultanés. Sur une architecture x86, avec 64 Mo de mémoire, et si chaque *coin* utilisateur héberge un service qui consomme 1 Mo de mémoire, ce nombre se porte à 10 pour VOSGi/MOSGi, ou encore 15 pour VOSGi seulement, sans l'agent de gestion JMX. Pour HGL, ce nombre monte à 48, sans agent de gestion équivalent à MOSGi.

Si HGL supporte 3 fois plus d'utilisateurs, VOSGi/MOSGi garde en revanche les avantages

que Java offre aux développeurs. En effet, le langage Java est fortement typé, ce qui facilite la détection d'erreurs à la compilation ; aussi, la JVM propose un ramasse-miette qui permet de s'abstraire des problèmes d'allocations mémoire. Ces avantages conduisent le développeur Java tester et déboguer plus vite, tout en écrivant moins de code non fonctionnel.

6.2.2 Différences de fonctionnalités

Cycle de vie Les diagramme d'état des *bundles* OSGi (figure 2.4 page 22) et des *bundles* HGL (figure 5.3 page 81) sont très similaires. On leur trouve cependant deux différences, qui sont l'ajout dans HGL d'un état EN PAUSE et la suppression de l'état RÉSOLU d'OSGi. L'état EN PAUSE est utile pour les *bundles* ayant un état interne, c'est-à-dire le jeu de valeurs de ses attributs : le mettre en pause conserve cet état interne, tandis que l'arrêter remet son état interne à zéro. L'absence de l'état EN PAUSE dans OSGi est un choix de l'Alliance OSGi ; c'est alors à chaque *bundle* de sauvegarder son état interne dans sa méthode `stop` et de le recharger dans sa méthode `start`. L'exemple typique est d'écrire et lire l'état interne dans un fichier, situé dans le répertoire cache privé du *bundle*, en utilisant l'appel `bundleContext.getDataFile(fichier)`. Pour résumer, dans HGL, l'environnement d'exécution gère la mise en pause, alors que dans OSGi, le développeur doit coder lui-même la mise en pause dans l'application.

De la même manière, un service OSGi est sans état. Par choix dans les spécifications, tout service est un singleton : chaque appel à `registerService` enregistre un objet unique dans la plate-forme OSGi, qui implante une interface de service particulière dans une version donnée. Si l'on veut un équivalent des EJB Session à état de J2EE, où chaque client d'un service dispose de son propre objet à état implantant de le service, il faut développer un service OSGi spécifique réalisant ceci ; la plate-forme OSGi ne s'en charge pas. Là encore, c'est au développeur de coder lui-même cet aspect non fonctionnel dans l'application, par exemple en utilisant des usines [123]. Ce choix s'explique par la volonté d'avoir une plate-forme OSGi simple, qui reste pertinente pour différents cas d'utilisation, allant du serveur d'applications d'entreprise aux passerelles domestiques, en passant par l'informatique pour automobiles. L'intégration de logique non métier additionnelle dans l'environnement d'exécution OSGi se fait donc par des surcouches, comme Spring pour les applications d'entreprise ou VOSGi pour les aspects multi-utilisateurs.

L'état RÉSOLU dans OSGi, en revanche, existe à cause de contraintes techniques, plus précisément à cause du mécanisme de chargement de classes par *bundles*. Si l'on réutilise le formalisme que nous avons posé en section 3.2.3 page 48, si un *bundle* A dépend d'un *package* Java présent dans le *bundle* B, on a : $A_{STARTED} \rightarrow B_{INSTALLED}$. En d'autres termes, pour que le *bundle* A puisse démarrer, le *bundle* B doit être correctement installé. OSGi vérifie que

cette dépendance est résolue non pas lorsque l'on commande de démarrer A, mais au préalable, lorsque OSGi crée le chargeur de classes dédié à A. Cette dépendance est également réévaluée à chaque apparition, disparition ou changement du cycle de vie d'un *bundle*. Ainsi, un *bundle* « bien installé » est un *bundle* dont le chargeur de classes est correctement lancé (ou peut correctement se lancer), ce que OSGi appelle résolu. La dépendance de A n'est donc pas vers $B_{INSTALLED}$ mais vers $B_{RESOLVED}$. Aussi, cette dépendance n'est pas vérifiée quand A démarre, mais quand A est résolu. L'expression correcte de cette dépendance est donc : $A_{RESOLVED} \rightarrow B_{RESOLVED}$.

Dans HGL, ce mécanisme de chargeurs de classes par *bundle* n'existe pas. Les dépendances au niveau *bundle* sont donc généralement du type $A_{INSTALLED} \rightarrow B_{INSTALLED}$, tandis que celles liées à l'activateur sont le plus couramment du type $A_{STARTED} \rightarrow B_{STARTED}$. L'état RÉSOLU est donc dispensable dans HGL.

Partage de services Dans OSGi comme dans GNU/Linux, pour faire appel à du code provenant d'une autre unité d'exécution, cette dernière doit être « visible ». Dans OSGi, un *bundle* s'adresse à son chargeur de classes pour trouver une classe Java, et au registre OSGi pour trouver un service OSGi. Dans GNU/Linux, un processus cherche à lancer un exécutable ou à charger une librairie partagée. Il les recherche, sous forme de fichier, dans les répertoires compris dans ses variables d'environnement PATH et LD_LIBRARY_PATH, respectivement.

Ainsi, partager un *bundle* OSGi entre plusieurs utilisateurs revient à créer un lien entre leurs chargeurs de classes respectifs. Partager un *bundle* HGL entre plusieurs utilisateurs revient à donner les droits de lecture et d'exécution sur des répertoires contenant des exécutables et des librairies dynamiques.

L'avantage du premier sur le second est d'être fortement typé, c'est-à-dire que le compilateur et l'environnement d'exécution donnent des garanties sur la résolution de la dépendance. L'avantage du second sur le premier relève de l'implantation, et non de la conception : si un *bundle* partagé apparaît ou disparaît pendant la durée de vie d'un *coin* utilisateur, HGL le prend en compte, VOSGi non.

Modèle de programmation GNU/Linux ne propose pas par défaut de programmation orientée service. Ce qui s'en rapproche le plus est la notion de noms virtuels, que l'on trouve par exemple dans la distribution Gentoo, pour les paquetages comme pour les scripts `rc`. Un paquetage virtuel indique que plusieurs paquetages « concrets » sont susceptibles de fournir les mêmes fonctionnalités, et qu'ils sont donc interchangeables. Par exemple, le paquetage Gentoo `virtual/jdk` est concrétisé par `dev-java/sun-jdk`, mais aussi par `dev-java/blackdown-jdk`. Les paquetages nécessitant un JDK dépendent donc de `virtual/jdk` plutôt que d'une implantation particulière. De même, les scripts `rc` fournissent un service ; `dcron` et `vixie-cron` sont deux implantations du même service. Ils déclarent donc `provide cron`, et les scripts `rc`

nécessitant un serveur `cron` déclarent `need cron`. Ces noms virtuels sont donc une abstraction d'implantations par une description plus générique, tout comme une interface Java (ou un service OSGi) abstrait l'implantation qui en est faite. La différence, tout de même fondamentale, est que la notion d'interface Java intervient dans le code source est est connue du compilateur, alors que les noms virtuels dans GNU/Linux ne sont qu'une convention d'écriture des méta-données, et ne sont pas connus du compilateur.

Architecture de gestion Dans les fonctionnalités de gestion offertes, MOSGi est plus avancé que HGL. Si HGL gère le déploiement des *bundles* dans un *coin* utilisateur, et gère les *coins* utilisateur eux-mêmes, MOSGi propose en plus d'intégrer la gestion des aspects métier des services contenus dans le *coin* utilisateur. Pour cela, localement à la passerelle, la MIB de MOSGi est peuplée à la volée par les services démarrés qui proposent une interface MBean. De plus, pour le superviseur de services, nous fournissons une console de gestion dont l'interface graphique est spécialisable à la volée en fonction de la MIB du *coin* utilisateur distant à un instant donné. HGL ne permet pas de faire cela, et nécessite donc l'ajout d'un agent de gestion supplémentaire du type TR-069 ou SNMP si l'on veut obtenir des fonctionnalités similaires à MOSGi. Notons que les services GNU/Linux existants doivent alors être modifiés pour tirer parti de cet agent.

6.2.3 Dépendances exprimées et dépendances réelles

Un environnement d'exécution multi-services résout des dépendances entre unités d'exécution, tout au long du cycle de vie de ces dernières. En premier lieu, les méta-données des unités de déploiement contiennent les dépendances exprimées par le programmeur ou le *packager*. En second lieu, leur code a des dépendances que nous appelons réelles : une classe Java à charger pour instancier un objet, une librairie native à charger pour résoudre un symbole. Ces dépendances réelles sont nécessaires pour l'exécution de l'unité de déploiement, sans quoi l'environnement d'exécution retourne une erreur. Idéalement, les premières dépendances doivent refléter les secondes. Si ces deux jeux de dépendances ne se recouvrent pas parfaitement, on a alors soit des dépendances exprimées superflues, qui provoquent d'inutiles installations ou démarrages d'unités de déploiement, soit des dépendances exprimées insuffisantes, qui provoquent de potentielles erreurs ou *crash* à l'exécution. Nous observons ici les différences dans les dépendances exprimées et réelles entre OSGi et GNU/Linux.

Dépendances exprimées dans OSGi Pour OSGi, les dépendances s'expriment à deux niveaux. Premièrement, un *bundle* OSGi requiert un *package* Java particulier (Import-Package), ce qui est le cas le plus courant, ou il requiert un autre *bundle* (Require-Bundle), ce qui en pratique reste un cas particulier. Ces dépendances s'expriment par des méta-données contenues

dans le fichier *manifeste*. Elles doivent donc être connues à l'avance, à la création du fichier archive `.jar` du *bundle*. Des outils comme `bnd` savent générer automatiquement le fichier *manifeste* à partir du code, mais cette phase se fait toujours au préalable. Les dépendances de *packages* ou de *bundles* sont résolues après l'installation du *bundle*, et avant de passer à l'état RESOLVED.

Les dépendances de *packages* peuvent aussi être résolues plus tard, en utilisant la clause DynamicImport-Package. Dans ce cas, le *bundle* démarre, cherche à charger une classe qui n'a pas encore été résolue, et déclenche le mécanisme de résolution à ce moment là. Il ne s'agit cependant, d'après les termes de la spécification OSGi, que d'une solution de dernier recours, lorsque l'on ne connaît pas à l'avance la liste des *packages* Java que le *bundle* est susceptible d'utiliser.

Le deuxième niveau de dépendances dans OSGi correspond aux services. Un *bundle* OSGi démarré consulte le registre de services et demande quels sont les services enregistrés correspondant à une interface particulière (`getServiceReferences`). Le nom de cette interface est paramétrable, et n'est donc potentiellement pas connue au moment de la compilation et de la création de l'archive `.jar` ; son *package* en revanche doit être connu et inclus dans la directive Import-Package, ou à défaut via DynamicImport-Package. Le *bundle* applique un filtre LDAP sur la liste de services renvoyée par le registre, en choisit un, et récupère une référence sur l'objet qui l'implante (`getService`).

Dépendances réelles dans OSGi La plate-forme OSGi vérifie par programmation que les dépendances exprimées au niveau *package* ou *bundle* sont résolues, et matérialise les imports/exports de *packages* par des objets `wire`. En d'autres termes, le mécanisme de chargeurs de classes, via les objets `wire`, assure que le partage de *packages* entre *bundles* ne produit pas d'erreurs lorsqu'un *bundle* démarre ou cherche à utiliser une classe d'un *package* importé.

Au niveau service, le mécanisme de chargeurs de classes intervient également. Si un *bundle* appelle `getServiceReferences(org.foo.MonService)`, alors le *package* `org.foo` doit être visible par son chargeur de classes. Mais ce n'est pas tout : le service WireAdmin crée également des objets `wire` pour représenter la relation entre le fournisseur et le consommateur d'un service OSGi. Le WireAdmin est utilisé par exemple pour propager des changements de propriétés Java sur un service OSGi.

Au final, les dépendances réelles dans OSGi sont des classes particulières à charger et des références sur des objets à obtenir. Elles sont résolues par le biais de chargeurs de classes et d'objets facilitateurs `wire`. On retombe systématiquement sur la notion de classe Java et du *package* auquel elle appartient. Les dépendances exprimées dans les méta-données OSGi étant principalement au niveau *package*, la correspondance entre dépendances exprimées et dépendances réelles est quasiment immédiate. Il s'agit d'une propriété intéressante, car elle

facilite la génération automatique des dépendances exprimées dans les méta-données à partir d'une analyse du code. Il y a ainsi peu de risques que les méta-données d'un *bundle* soient incohérentes par rapport à son contenu, car il y a peu de place pour une erreur humaine du développeur ou du *packager*.

Dépendances exprimées dans GNU/Linux Dans les systèmes GNU/Linux, on trouve trois types de dépendances : entre paquetages, entre scripts `rc` et entre modules du noyau. Les dépendances entre paquetages sont exprimées de la même manière que celles d'OSGi au niveau *package* Java, c'est-à-dire par des méta-données. Le gestionnaire de paquetages maintient une représentation du système, sous la forme d'une liste de paquetages installés. Les dépendances ne sont pas vérifiées par programmation, mais par correspondance entre méta-données : si un paquetage est noté comme installé, on le considère installé et fonctionnel, même s'il a été supprimé manuellement du système de fichiers.

Les dépendances entre scripts `rc` fonctionnent de la même manière : elles sont exprimées par des clauses du type `need XX` et `provide YY` dans les scripts eux-mêmes, et sont vérifiées par correspondance de méta-données. Si le script `XX` est considéré comme lancé, mais que son processus démon associé a été tué, alors la résolution de dépendances `need XX` va tout de même, à tort, réussir.

Enfin, les dépendances entre modules du noyau sont exprimées dans des fichiers Kconfig : `config XXX depends on YYY && ZZZ`. Pendant l'exécution du noyau, les dépendances sont vérifiées et écrites via le fichier `/proc/modules`. Dans GNU/Linux, le système de fichiers joue donc en quelque sorte le rôle du registre, en particulier les répertoires `/proc` et `/var`.

Dépendances réelles dans GNU/Linux Les exécutables GNU/Linux sont souvent liés à des librairies partagées, lors de la phase d'édition des liens. Pour s'exécuter, ils ont alors besoin de la présence de ces librairies partagées. Le format binaire ELF (*Executable Linking Format*) de Linux contient une table de références externes, qui donne la liste des librairies partagées et les symboles qu'elles implantent. L'outil `ld` charge les librairies partagées soit au lancement de l'exécutable, soit plus tard, à l'utilisation des symboles (*lazy resolution*, résolution fainéante). Les librairies partagées se trouvent dans des répertoires indiqués par la variable d'environnement LD_LIBRARY_PATH, indiqués par le fichier `/etc/ld.so.conf`, ou dans `/lib` et `/usr/lib`. Elles ont pour nom de fichier `libNomDeLibrairie.so`. Si une librairie partagée n'est pas trouvée, l'environnement d'exécution produit une erreur.

On peut aussi considérer que, lors d'appels à `exec` ou `system`, l'exécutable passé en paramètre est une dépendance. Le fichier exécutable doit se trouver dans un répertoire indiqué par la variable d'environnement PATH, avec les droits d'exécution correctement positionnés.

Il apparaît donc que, si les dépendances exprimées dans les méta-données portent sur des noms de paquetages GNU/Linux, les dépendances réelles contenues dans les exécutables

portent sur des fichiers (librairies et exécutables). La correspondance entre les deux n'est pas explicite, et dépend d'interventions humaines de la part des développeurs et des mainteneurs de paquetages. Il y a donc un risque d'erreur ou d'imprécision, et cela est une fragilité des distributions GNU/Linux.

Corréler dépendances exprimées et dépendances réelles Deux types de remèdes existent pour gommer cet écart. Le premier est d'utiliser des outils qui génèrent les méta-données à partir des dépendances réelles dans le code. C'est ce que propose par exemple `bnd` pour OSGi. L'avantage est de limiter les risques d'erreur dans les méta-données. L'inconvénient est que les dépendances existent toujours en deux endroits, et donc que le risque d'incohérence n'est pas nul. De plus, appliquer cette technique à GNU/Linux n'est pas qu'un problème d'analyse de code. Il faut en effet combler la séparation entre les dépendances exprimées au niveau unité de déploiement (paquetage, *bundle* HGL) et les dépendances réelles au niveau fichier (librairie partagée, exécutable). Dans la distribution Debian par exemple, les personnes qui maintiennent les paquetages tentent autant que possible de distribuer une librairie particulière dans un paquetage particulier. Il s'agit cependant d'une mise en œuvre organisationnelle, et non technique, sur laquelle l'environnement d'exécution ne peut avoir de garanties de bon fonctionnement.

La seconde solution pour faire converger dépendances exprimées et dépendances réelles est alors de ne pas utiliser de méta-données, ou plutôt de les noyer dans le code. L'environnement d'exécution se charge alors de lire le code et de recréer à la volée les dépendances exprimées, garantissant ainsi leur cohérence. Un avantage annexe est de s'affranchir du langage de description des méta-données, du type fichier *manifeste*. Le mécanisme d'annotations introduit dans Java 1.5 en est un exemple. L'inconvénient de cette technique est qu'elle nécessite un support dans le langage de programmation. Les plates-formes du type OSGi peuvent tirer profit des annotations Java au prix d'un impact sur le modèle de programmation. Pour GNU/Linux en revanche, les langages de programmation sont si nombreux et les habitudes de programmation si profondément ancrées chez les développeurs que modifier langages et modèles de programmation aurait un coût prohibitif.

6.2.4 Tableau comparatif

Le tableau 6.1 résume les points forts et points faibles des deux implantations réalisées.

Les principaux avantages de VOSGi/MOSGi sont l'expressivité et le typage fort du langage Java, le modèle de programmation orientée service d'OSGi, et une architecture de gestion multi-services de bout en bout. Ses inconvénients sont le coût en mémoire, acceptable mais perfectible, et l'isolation de nommage seulement : il est difficile d'y ajouter une isolation de ressources sans être intrusif dans le système d'exploitation ou la machine virtuelle.

Les principaux avantages de HGL sont la faible utilisation mémoire et disque, la possibilité d'inclure des outils d'isolation de ressources, ainsi que la gamme d'applications existantes réutilisables. Ses inconvénients sont l'écriture manuelle des dépendances dans les métadonnées, et la notion de service qui passe par des noms virtuels plutôt que par un typage fort.

	VOSGi / MOSGi	**HGL**
Cycle de vie	+++	+++
Gestion des dépendances	+++	+++
Dépendances exprimées et dépendances réelles	++	-
Partage de services	+ partage statique de *bundles*	+ partage dynamique de fichiers
Modèle de programmation	+++ programmation orientée service	+ services = noms virtuels
Isolation	+	++
Gestion	+++	++ agent natif TR-069 non implanté
Empreinte mémoire	+	+++
Applications existantes	+	+++

Tab. 6.1: Comparaison entre VOSGi/MOSGi et HGL

6.3 Autres styles d'implantation possibles

Nous avons choisi deux styles d'implantation d'environnement d'exécution multi-services, mais d'autres possibilités existent. En particulier, pour HGL, nous avons choisi de nous baser sur un système d'exploitation GNU/Linux sans en changer le modèle de programmation. Notre raisonnement est que nous voulons réutiliser le vaste choix d'applications existantes, sans dépayser les développeurs et les mainteneurs de paquetages. Si toutefois ces raisons ne sont pas une priorité, il est envisageable de changer le modèle de programmation des langages usuels dans GNU/Linux, notamment pour adopter une architecture orientée service ou un modèle de programmation proche de la programmation orientée service.

6.3.1 GNU/Linux et SOA

Une possibilité est donc d'adopter une architecture orientée service pour GNU/Linux. Les *bundles* HGL coopéreraient alors par un bus logiciel. C'est la voie qu'a choisi l'environnement graphique GNOME [130] : les composantes comme la barre des tâches ou le gestionnaire de fichiers communiquent par CORBA. Ainsi, les composantes de GNOME peuvent interagir, rechercher un service particulier, et faire démarrer un service en cas de besoin. Ceci a nécessité la réécriture ou l'adaptation des sous-projets de GNOME pour utiliser CORBA, et n'a pas été sans heurts. Ce cas illustre bien l'importance de l'adoption par les développeurs : beaucoup

ont critiqué le choix de CORBA, pour la lourdeur de la mise en œuvre, qui est avérée, et pour la perte de performance, qui n'a pas été prouvée.

D-Bus [131] est une proposition plus récente qui vise à remplacer CORBA dans GNOME, et son équivalent DCOP dans l'environnement graphique concurrent KDE. Il s'agit d'un mécanisme de communications inter-processus par messages et par notifications. Son format d'échange est plus concis que celui de CORBA, tout en restant indépendant du langage de programmation utilisé dans les composants D-Bus : il est donc utilisable par toute application GNU/Linux, qu'elle soit écrite en C, en shell ou autre, pourvu qu'une librairie D-Bus existe pour ce langage. D-Bus permet en théorie d'améliorer l'intégration du gestionnaire de paquetages, des activateurs ou des scripts `rc` ou encore du chargeur de modules du noyau. Il s'agit donc d'une alternative à l'implantation actuelle de HGL qu'il faut retenir.

6.3.2 GNU/Linux et SOP

Dans la programmation orientée service, un service est une interface, qui existe concrètement à l'exécution ; en Java, une interface dispose de son propre *bytecode*. Ceci permet à l'environnement d'exécution de s'assurer qu'un consommateur et différents fournisseurs d'un même service parlent bien du même service. Nous avons également exprimé, en section 2.4.2, une seconde caractéristique de la SOP : si, dans les architectures orientées service, chaque service dispose de son propre espace d'adressage, les services dans la programmation orientée service s'exécutent dans un espace d'adressage unique. Ainsi, résoudre une dépendance en SOP revient à trouver une référence dans l'espace d'adressage local. Dans OSGi, les *bundles* ont un espace d'adressage privé (leur chargeur de classes), mais ils publient leurs *packages* et services exportés dans un espace de nommage commun (délégation du chargement de classes, registre).

La programmation orientée service, contrairement aux SOA, est fortement dépendante du langage utilisé. Si l'on veut proposer un modèle de programmation orientée service pour un système d'exploitation GNU/Linux, il faut tout d'abord se restreindre à un seul langage. Nous choisirons pour ce qui suit le langage le plus utilisé : le C. Ensuite, il convient d'appliquer les deux caractéristiques sus-citées au langage choisi.

La première est quelque peu inhabituelle en C. Les fichiers en-tête `.h`, qui peuvent jouer le rôle d'interface lors de la compilation du code source, n'existent plus après cette phase : ils sont transformés en symboles dans les fichiers objet `.o` générés, avant la phase d'édition des liens. On ne peut retrouver le nom du fichier `.h` à partir de ces symboles. Une interface Java, elle, existe à la fois à la compilation et à l'exécution, sous le même nom. Un environnement d'exécution pour programmes C ne peut donc vérifier la conformité de plusieurs fichiers objet `.o` à une même interface. On peut par contre utiliser des conventions de nommage, comme on le fait déjà avec les noms virtuels pour exprimer les dépendances. Une meilleure solution

méritérait plus de réflexion que nous n'avons accordé à ce problème, qui reste en dehors de nos préoccupations pour ces travaux.

En revanche, la deuxième caractéristique, liée aux espaces d'adressage, est adaptable au C. Si l'on compile chaque *bundle* sous forme de librairie partagée `.so`, un programme principal du type `hgl_init` peut utiliser les primitives du chargeur de liens dynamique : `dlopen` pour charger un *bundle*, `dlsym` pour résoudre ses dépendances de symboles, et `dlclose` pour le décharger.

Cette proposition évite ainsi les IPC lorsque deux *bundles* veulent coopérer. Elle ne remédie cependant pas aux différences entre dépendances exprimées et dépendances réelles, ni au typage faible du C qui limite ses capacités dans la résolution des dépendances.

6.4 Conclusion

Nous avons présenté dans ce chapitre les différences techniques entre plusieurs styles d'implantation pour des environnements d'exécution multi-services. Toutefois, ces styles d'implantation restent dans l'ensemble conformes à un même modèle, qui contient des points fondamentaux bien définis, répartis en fonctions de déploiement et d'isolation.

La notion d'unité de déploiement, proche de celle de composant logiciel, encapsule du code et des données dans une unité quasi-autonome. Les unités de déploiement ont la granularité d'une application, d'un module d'une application ou d'une librairie. Elles sont dites quasi-autonomes car leurs interactions avec le monde extérieur, c'est-à-dire avec du code ou des données provenant d'autres unités de déploiement, sont clairement identifiés par des dépendances.

L'environnement d'exécution contrôle le cycle de vie des unités de déploiement : il les rapatrie sur la passerelle domestique, les installe, les démarre via un point d'entrée spécial appelé activateur, les arrête, les désinstalle. Á chaque changement dans ce cycle de vie, il résout les dépendances.

Parallèlement à ces activités de déploiement, un environnement d'exécution multi-services supporte plusieurs utilisateurs système. Chaque utilisateur administre son propre *coin* du système, où il est le seul à pouvoir déployer ses propres unités de déploiement. Un utilisateur privilégié, ou administrateur, contrôle la création des *coins* utilisateur, et propose des services communs qu'il partage vers tous les utilisateurs.

Chaque *coin* utilisateur propose une interface de gestion unique qui donne accès à toutes les activités de déploiement dans ce *coin*. Le *coin* administrateur en fait de même, mais propose en même temps de gérer les activités d'isolation.

Toutes ces fonctionnalités regroupées constituent un modèle informel pour les environnements d'exécution multi-services. Nous en avons proposé deux implantations. La première est

basée sur Java/OSGi, et est constituée de VOSGi pour la partie isolation et de MOSGi pour la partie gestion. La seconde, baptisée HGL, est basée sur GNU/Linux. Toutes sont disponibles sous licence *open source*, et MOSGi fait partie de Felix, l'implantation d'OSGi par la fondation Apache.

Les conclusions que nous tirons de VOSGi/MOSGi et de HGL sont qu'il faut faire un compromis entre l'exhaustivité des fonctionnalités, les performances et le modèle de programmation. Avec VOSGi, nous obtenons un mécanisme d'isolation de nommage, tout en tirant parti de la programmation orientée service. Obtenir une isolation de ressources est en revanche fortement dépendant de la machine virtuelle et du système d'exploitation sous-jacents, ce qui ruine l'avantage de la portabilité de Java. Avec HGL, nous obtenons également une isolation de nommage, mais il s'agit là d'un choix d'implantation et non d'une contrainte de GNU/Linux. En revanche, nous conservons le modèle de programmation usuel des langages comme le C ou le shell. Celui-ci, par rapport à la programmation orientée service, est plus sujet à des erreurs dans l'expression des dépendances. L'avantage de HGL sur VOSGi/MOSGi est que les *coins* utilisateur consomment moins de ressources, en particulier de mémoire, indépendamment des unités de déploiement qu'ils hébergent.

L'avantage de HGL en termes de performance a du sens sur des équipements réduits comme des passerelles domestiques, même si VOSGi/MOSGi a des résultats raisonnables sur les équipements actuels. Cependant, nous pouvons envisager d'utiliser un environnement d'exécution multi-services dans d'autres cas. En particulier, les gros serveurs d'entreprise sont des environnements multi-utilisateurs avec de forts besoins d'administration. Intégrer une gestion fine du déploiement et une gestion de *coins* utilisateur semble donc une voie intéressante pour ces systèmes. Dans ce cas, tirer profit de la programmation orientée service, de son expressivité et de sa dynamicité peut primer sur des différences de performances qui deviennent négligeables.

Bibliographie

[1] Hubert Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4) :425 – 432, April 1980.

[2] MUSE Project. IST-026442 Framework Programme 6. http://www.ist-muse.org/, 2004–2007.

[3] Amigo Project. IST-004182 Framework Programme 6. http://www.hitech-projects.com/euprojects/amigo/, 2004-2007.

[4] ITU-T. Focus group on next generation networks (FGNGN). http://www.itu.int/ITU-T/ngn/fgngn/, 2004–2005.

[5] IETF. The Common Management Information Services and Protocols for the Internet (CMOT and CMIP). http://tools.ietf.org/html/rfc1189, October 1990.

[6] ETSI. Telecoms and Internet converged Services and Protocols for Advanced Network. http://www.etsi.org/tispan/, since 2005.

[7] ETSI. Parlay 6.0 specifications : APIs for Open Service Access, ETSI ES 204 915. http://portal.etsi.org/docbox/TISPAN/Open/OSA/Parlay60.html, 2007.

[8] Next Generation Networks Initiative. IST project, Framework Programme 6. http://www.ngni.org/, 2001–2004.

[9] MediaNet. IST project, Framework Programme 6. http://www.ist-ipmedianet.org/, 2001–2004.

[10] CENELEC. TC 205 - Home and Building Electronic Systems (HBES). http://www.cenelec.org/Cenelec/CENELEC+in+action/Horizontal+areas/ICT/TC205.htm.

[11] Consumer Electronics Association. Home Automation System (CEBus), standard CEA-600 CEBus set. Consumer Electronics Association, Arlington, VA, 1996.

[12] KNX Association. KNX Standard, ISO/IEC 14543-3. http://www.knx.org/, December 2006.

[13] International Telecommunication Union-T. Recommendation J.190 - architecture of mediahomenet that supports cable based services. ITU-T J.190, 2002.

[14] X10. X10 Powerline Carrier (PLC) Technology. http://www.x10.com/, since 1997.

[15] The HAVi Specification. Specification of the home audio/video interoperability (havi) architecture. `http://www.havi.org/`, 2004.

[16] N.P. Mahalik and P.R. Moore. Fieldbus technology based, distributed control in process industries : a case study with LonWorks technology. *Integrated Manufacturing Systems*, 8(4) :231–243, 1997.

[17] Electronic Industries Alliance. Control network protocol specification based on the LonTalk protocol. ANSI/CEA 709.1, 1999.

[18] IEEE Computer Society. Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs). IEEE Std 802.15.4, 2003.

[19] Digital Living Network Alliance. Overview and vision whitepaper. `http://www.dlna.org/en/industry/pressroom/DLNA_white_paper.pdf`, 2007.

[20] UPnP Forum. UPnP standards and specifications. `http://www.upnp.org/standardizeddcps/`, since 1999.

[21] Sun Microsystems. Jini specifications v1.2. `http://www.sun.com/software/jini/specs/`, December 2001.

[22] J. Piesing. The DVB multimedia home platform - "MHP". In *IEE Colloquium on Interactive Television, London (Ref. No. 1999/200)*, pages 2/1–2/6, 1999.

[23] Steven Morris and Anthony Smith-Chaigneau. *Interactive TV Standards, A Guide to MHP, OCAP and JavaTV*. Number 0-240-80666-2. Elsevier, Amsterdam, 2005.

[24] ETSI SMG4. Mobile station application execution environment (MExE). Specification No. GSM 02.57.

[25] M. Bath. Agile buildings at last - via obix. *Building Engineer - London - Association of Building Engineers*, 79(7) :18–19, 2004.

[26] IETF Megaco. Media gateway control working group. `http://tools.ietf.org/wg/megaco/`, since 1999.

[27] DSL Forum. Tr-069 : CPE WAN Management Protocol. `http://www.dslforum.org/techwork/tr/TR-069.pdf`, May 2004.

[28] Home Gateway Initiative. HGI Vision and White Paper, 2007. `http://www.homegatewayinitiative.org/public/docs/HGI_white_paper.pdf`.

[29] The OSGi Alliance. *OSGi Service Platform*. IOS Press, Amsterdam, 3rd edition, 2003.

[30] Guy Bieber and Jeff Carpenter. Openwings, a Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems. `http://www.openwings.org`, 2001.

[31] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7) :76–82, July 1999.

[32] Yvan Royon and Stéphane Frénot. Multiservice home gateways : Business model, execution environment, management infrastructure. *IEEE Communications Magazine*, October 2007.

[33] IETF. A simple network management protocol (SNMP). `http://www.ietf.org/rfc/rfc1157.txt`, 1990.

[34] IETF. NETCONF Configuration Protocol. `http://www.ietf.org/rfc/rfc4741.txt`, december 2006.

[35] OASIS. Web services distributed management : Management using web services (wsdm-muws) 1.1. `http://www.oasis-open.org/`, August 2006.

[36] OASIS. Web services distributed management : Management of web services (wsdm-mows) 1.1. `http://www.oasis-open.org/`, August 2006.

[37] Java Community Process. JSR 3 - Java Management eXtensions Specification. `http://www.jcp.org/en/jsr/detail?id=3`.

[38] Troy Bryan Downing. *Java RMI : Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998.

[39] Dong-Sung Kim, Jae-Min Lee, Wook Hyun Kwon, and In Kwan Yuh. Design and implementation of home network systems using upnp middleware for networked appliances. *IEEE Transactions on Consumer Electronics*, 48(4) :963–972, 2002.

[40] A. Häber, F. Reichert, and A. Fasbender. UPnP control point for mobile phones in residential networks. In *15th IST Mobile and Wireless Communication Summit*, Myconos, Greece, 2006.

[41] OpenWrt. Wireless freedom. `http://openwrt.org/`, since January 2004.

[42] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly, Sebastopol (Calif.), first edition, February 2001.

[43] OSGi Alliance. OSGi Bundle Repository RFC. `http://bundles.osgi.org/rfc-0112_BundleRepository.pdf`, april 2006.

[44] Stephane Frenot and Yvan Royon. Component deployment using a peer-to-peer overlay. In *Working Conference on Component Deployment*, Grenoble, France, november 2005.

[45] Apache Software Foundation. Harmony : Open Source Java SE. `http://harmony.apache.org/`.

[46] Stéphane Frénot and Dan Stefan. M-OSGi : Une plate-forme répartie de services. In *Notere*, Saidia, Maroc, 26-29 juin 2004.

[47] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi : Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference (Middleware 2007)*, November 2007.

[48] Java Community Process. JSR 232 - Mobile Operational Management. `http://www.jcp.org/en/jsr/detail?id=232`.

[49] IEEE and the Open Group. Single UNIX Specification, IEEE std 1003.1. `http://www.unix.org/single_unix_specification/`.

[50] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the CORBA component model. In *Component-based software engineering : putting the pieces together*, pages 557–571, Boston, MA, USA, 2001. Addison-Wesley Longman Publishing Co., Inc.

[51] Sara Williams and Charlie Kindel. The Component Object Model : A technical overview. In *Dr. Dobb's Journal*, December 1994.

[52] Ed Roman and Rickard Oberg. The technical benefits of EJB and J2EE technologies over COM+ and Windows DNA. Technical report, The MIDDLEWARE Company, December 1999.

[53] E. Bruneton, T. Coupaye, and J-B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 2002.

[54] J-P. Fassino, J-B. Stefani, J. Lawall, and G. Muller. THINK : A Software Framework for Component-based Operating System Kernels. In *USENIX*, Monterey, California, 2002.

[55] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal component model and its support in java. *Software Practice and Experience Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36 :11–12, 2006.

[56] OSGi Alliance. Listener Pattern Considered Harmful : The Whiteboard Pattern. OSGi Alliance, San Ramon, CA, USA, 2nd revision, 2004.

[57] Guy Bieber and Jeff Carpenter. Introduction to Service Oriented Programming. http ://www.openwings.org, 2001.

[58] Humberto Cervantes and Richard Hall. Automating service dependency management in a service-oriented component model. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering : Automated Reasoning and Prediction*. Portland, United-Stated, May 2003.

[59] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO : an extensible service-oriented component framework. In *SCC 2007. IEEE International Conference on Services Computing*, volume 9, pages 474–481, July 2007.

[60] John Arthur and Shiva Azadegan. Spring framework for rapid open source J2EE web application development : A case study. In *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Networks (SNPD/-SAWN'05)*, pages 90–95, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[61] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004.

[62] S. Vinoski. CORBA : integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2) :46–55, February 1997.

[63] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5) :483, 1981.

[64] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield. Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2) :99, 1972.

[65] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, New York, NY, USA, 1973. ACM Press.

[66] Joachim J. Wlodarz. Virtualization : A double-edged sword. arXiv.0705.2786 [cs.OS], May 2007.

[67] Jim Smith and Ravi Nair. *Virtual Machines : Versatile Platforms for Systems and Processes*. Morgan Kaufmann, San Francisco, CA, June 2005.

[68] Nicola Salmoria and the MAME team. The multiple arcade machine emulator. `http://mamedev.org/`, since 1997.

[69] Kevin P. Lawton. Bochs : A portable pc emulator for unix/x. *Linux Journal*, 29 :7, September 1996.

[70] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, Anaheim, CA, 2005.

[71] Sebastian Biallas. PearPC - PowerPC architecture emulator. `http://pearpc.sourceforge.net/`, 2003–2006.

[72] Parallels, Inc. Parallels desktop virtualization technical overview. `http://www.parallels.com/files/upload/Parallels_Technical_White_Paper.pdf`, April 2007.

[73] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02)*, volume 36, pages 181–194, New York, NY, USA, 2002. ACM Press.

[74] innotek GmbH. Virtualbox. `http://www.virtualbox.org/`, 2007.

[75] Microsoft. Virtual server technology. http://www.microsoft.com/windowsserversystem/virtualserver/, since 2005.

[76] Paul Barham, Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP*, New York, USA, October 2003.

[77] A. Whitaker, M. Shaw, and S. Gribble. Denali : Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference, Monterey, CA*, June 2002.

[78] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *5th symposium on System Design and Implementation (OSDI)*, Boston, Massachusetts, USA, 2002.

[79] Jonathan Corbet. An introduction to lguest. http://lwn.net/Articles/218766/, January 2007.

[80] IBM. The z/VM hypervisor. http://www.vm.ibm.com/, since 2005.

[81] Sun Microsystems. Logical Domains (LDoms). http://www.sun.com/servers/coolthreads/ldoms/, 2007.

[82] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta*, 2000.

[83] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization : A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the EuroSys'07 conference*, Lisbon, Portugal, 2007.

[84] SWsoft. Openvz, an operating system-level server virtualization solution built on linux. http://openvz.org/, since 2006.

[85] Andew Tucker and David Comay. Solaris Zones : Operating system support for server consolidation. In *3rd Virtual Machine Research and Technology Symposium Works-in-Progress*, San Jose, CA, May 2004.

[86] Paco Hope. Using Jails in FreeBSD for fun and profit. In *;login, the Magazine of USENIX & SAGE*, volume 27, pages 48–55. June 2002.

[87] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5) :48–56, May 2005.

[88] Advanced Micro Devices. Amd64 virtualization codenamed 'pacifica' technology. Secure Virtual Machine Architecture Reference Manual, Advanced Micro Devices, Sunnyvale, CA, May 2005.

[89] Qumranet. Kernel based virtual machine. http://www.qumranet.com/wp/kvm_wp.pdf, 2006.

[90] Sun Microsystems. UltraSPARC virtual machine specification v1.0, the sun4v architecture and hypervisor API specification. http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf, January 2006.

[91] W. J. Armstrong, R. L. Arndt, D. C. Boutcher, R. G. Kovacs, D. Larson, K. A. Lucke, N. Nayar, , and R. C. Swanberg. Advanced virtualization capabilities of POWER5 systems. *IBM Journal of Research and Development*, 49(4/5) :00, 2005.

[92] The WINE team. An open source implementation of the Windows API. http://www.winehq.org/, since 1993.

[93] Sun Microsystems. Java Virtual Machine specifications, second edition. http://java.sun.com/docs/books/jvms/, 1999.

[94] Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code Sharing among Virtual Machines. In *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 2002.

[95] Grzegorz Czajkowski and Laurent Daynès. Multitasking without comprimise : a virtual machine evolution. In *OOPSLA*, pages 125–138, New York, NY, USA, 2001. ACM Press.

[96] Grzegorz Czajkowski. Application Isolation in the Java Virtual Machine. In *ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, MN, 2000.

[97] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciaran Bryce. A Resource Management Interface for the Java Platform. Sun Technical Report TR-2003-124, May 2003.

[98] Sunil Soman, Laurent Daynès, and Chandra Krintz. Task-aware garbage collection in a multi-tasking virtual machine. In *ISMM '06 : Proceedings of the 2006 international symposium on Memory management*, pages 64–73, New York, NY, USA, 2006. ACM Press.

[99] Laurent Daynès and Grzegorz Czajkowski. Sharing the Runtime Representation of Classes across Class Loaders. In *European Conference on Object-Oriented Programming (ECOOP)*, Glasgow, UK, 2005.

[100] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A Multi-User Virtual Machine. In *Usenix*, pages 85–98, San Antonio, TX, USA, 2003.

[101] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *ACM SIGPLAN Conferences on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 36–44, Vancouver, Canada, 1998.

[102] Godmar Back and Wilson C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4) :583–630, July 2005.

[103] E. Prangsma. JNode. `http://jnode.sourceforge.net`, since 2003.

[104] M. Golm, M. Felser, C. Wawersich, and J. Kleinoeder. The JX Operating System. In *USENIX*, pages 45–58, Monterey, California, June 2002.

[105] Nicolas Le Sommer. *Contractualisation des ressources pour les composants logiciels : une approche réflexive*. PhD thesis, Université de Bretagne Sud, Décembre 2003.

[106] Frédéric Ogel, Gaël Thomas, Antoine Galland, and Bertil Folliot. MVV : une plateforme à composants dynamiquement reconfigurables — la machine virtuelle virtuelle. In *Numéro Spécial Technique et Science Informatiques (TSI)*. Hermes Science, 2005.

[107] Gaël Thomas, Frédéric Ogel, and Bertil Folliot. Jnjvm : une plateforme java adaptable pour application active. In *3ème Conférence Française sur les Systèmes d'Exploitation, CFSE'3, Chapitre français de l'ACM/SIGOPS*, La Colle sur Loup, France, Octobre 2003.

[108] Ian Piumarta, Frédéric Ogel, and Bertil Folliot. Ynvm : dynamic compilation in support of software evolution. In *Ingeneering Complex Object Oriented System for Evolution (OOPSLA)*, Tampa Bay, Florida, October 2001.

[109] Walter Binder, Jane G. Hulaas, and Alex Villazon. Portable resource control in Java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 139–155, Tampa, Florida, USA, 2001. ACM Press.

[110] Grzegorz Czajkowski and Thorsten von Eicken. JRes : a resource accounting interface for Java. In *OOPSLA '98 : Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–35, New York, NY, USA, 1998. ACM Press.

[111] Vladimir Calderón. J-RAF - The Java Resource Accounting Facility. Master's thesis, University of Geneva, June 2002.

[112] Sun Microsystems. Java platform debugger architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`, since 2002.

[113] Sun Microsystems. The phoneME project. `https://phoneme.dev.java.net/`, since 2006.

[114] Apache Software Foundation. Apache virtual host documentation. `http://httpd.apache.org/docs/2.2/vhosts/`, 2007.

[115] Home Gateway Initiative. Public Deliverable v1.0. `http://www.homegatewayinitiative.org/publis/HGI_V1.0.pdf`, July 2006.

[116] Apache Software Foundation. Felix OSGi R4 Service Platform implementation. `http://felix.apache.org/`, since 2005.

[117] Yvan Royon, Stéphane Frénot, and Frédéric Le Mouël. Virtualization of Service Gateways in Multi-provider Environments. In *Component-Based Software Engineering*, Sweden, 2006.

[118] Yvan Royon and Stéphane Frénot. Un environnement multi-utilisateurs orienté service. In *Conférence Française en Systèmes d'Exploitation (CFSE'5)*, Canet en Roussillon, octobre 2006.

[119] Java Community Process. JSR 160 - Java Management eXtensions Remote API. `http://www.jcp.org/en/jsr/detail?id=160`, October 2003.

[120] Eric Fleury and Stéphane Frénot. Building a JMX management interface inside OSGi. Technical report, Inria RR-5025, INRIA Rhône-Alpes, 2003.

[121] MX4J. Open source jmx for entreprise computing, release 3.0.2. `http://mx4j.sourceforge.net/`, October 2006.

[122] Mandy Chung. Using JConsole to Monitor Applications. Sun Developer Network, December 2004.

[123] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston (Mass.), 1995.

[124] Jan S. Rellermeyer and Gustavo Alonso. Concierge : A service platform for resource-constrained devices. In *Proceedings of the 2007 ACM EuroSys Conference*, Lisbon, Portugal, 2007.

[125] Bruno Dillenseger and Emmanuel Cecchet. CLIF is a Load Injection Framework. In *Proceedings of the OOPSLA 2003 Workshop on Middleware Benchmarking*, volume 26, Anaheim, CA, USA, October 2003.

[126] Daniel F. García, Javier García, Manuel García, Ivan Peteira, Rodriguo García, and Pablo Valledor. Benchmarking of web services platforms, an evaluation with the TPC-App benchmark. In *International Conference on Web Information Systems and Technologies (WEBIST 2006)*, Setubal, Portugal, April 2006.

[127] Abdelkader Lahmadi, Laurent Andrey, and Olivier Festor. Performances et resistance au facteur d'échelle d'un agent de supervision basé sur JMX : Méthodologie et premiers résultats. In *Colloque francophone de Gestion de RÃ©seaux et de Services (GRES)*, Luchon, 2005.

[128] Nicholas Wells. BusyBox : a swiss army knife for linux. *Linux Journal*, 2000(78es) :10, 2000.

[129] Yvan Royon and Stéphane Frénot. A Survey of Unix Init Schemes. Technical Report 0338, INRIA Rhône-Alpes, June 2007.

[130] George Lebl, Elliot Lee, and Miguel de Icaza. Gnome, its state and future. *Linux Journal*, 2000(70es) :3, February 2000.

[131] Robert Love. Get on the D-BUS. *Linux Journal*, 2005(130), February 2005.