Managing resource sharing conflicts in an open embedded software environment

Koutheir Attouchi

▶ **To cite this version:**

Koutheir Attouchi. Managing resource sharing conflicts in an open embedded software environment. Embedded Systems. Université Pierre et Marie Curie - Paris VI, 2014. English. NNT : 2014PA066619 . tel-01088028v2

HAL Id: tel-01088028
https://theses.hal.science/tel-01088028v2

Submitted on 30 Apr 2015

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.

Université Pierre et Marie Curie
École Doctorale Informatique, Télécommunications et Électronique (Paris)
Laboratoire d'Informatique de Paris 6 / REGAL
Orange Labs, Grenoble

Ph.D. Thesis in Computer Science

Managing Resource Sharing Conflicts in an Open Embedded Software Environment

Author
Koutheir Attouchi

Ph.D. Director
Gilles Muller

Ph.D. Director
Gaël Thomas

Industrial Supervisor
André Bottaro

Presented and defended publicly on Friday 11th of July 2014,
in front of a thesis committee composed of :

Reviewers :
Didier Donsez — *Professor (HDR), University of Grenoble 1.*
Laurence Duchien — *Professor (HDR), University of Lille 1.*

Jury Members :
Béatrice Berard — *Professor (HDR), UPMC Paris 6.*
Jean-Philippe Fassino — *Cyber-Security Architect, Schneider Electric.*
Johann Bourcier — *Lecturer, University of Rennes 1.*

Ph.D. Directors :
Gilles Muller — *Senior Research Scientist (HDR), Inria.*
Gaël Thomas — *Associate Professor (HDR), UPMC Paris 6.*

Industrial Supervisor :
André Bottaro — *Research Program Director, Orange Labs.*

Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-nd/4.0/

Abstract

Our homes are becoming smart thanks to the numerous devices, sensors and actuators available in it, providing services, e.g., entertainment, home security, energy efficiency and health care. Various service providers want to take advantage of the smart home opportunity by rapidly developing services to be hosted by an embedded smart home gateway. The gateway is open to applications developed by untrusted service providers, controlling numerous devices, and possibly containing bugs or malicious code. Thus, the gateway should be highly-available and robust enough to handle software problems without restarting abruptly. Sharing the constrained resources of the gateway between service providers allows them to provide richer services. However, resource sharing conflicts happen when an application uses resources "unreasonably" or abusively. This thesis addresses the problem of resource sharing conflicts in the smart home gateway, investigating prevention approaches when possible, and considering detection and resolving approaches when prevention is out of reach.

Our first contribution, called Jasmin, aims at preventing resource sharing conflicts by isolating applications. Jasmin is a middleware for development, deployment and isolation of native, component-based and service-oriented applications targeted at embedded systems. Jasmin enables fast and easy cross-application communication, and uses Linux containers for lightweight isolation. Our second contribution, called Incinerator, is a subsystem in the Java Virtual Machine (JVM) aiming to resolve the problem of Java stale references, i.e., references to objects that should no more be used. Stale references can cause significant memory leaks in an OSGi-based smart home gateway, hence decreasing the amount of available memory, which increases the risks of memory sharing conflicts. With less than 4% overhead, Incinerator not only detects stale references, making them visible to developers, but also eliminates them, hence lowering the risks of resource sharing conflicts. Even in Java, memory sharing conflicts happen. Thus, in order to detect them, we propose our third contribution: a memory monitoring subsystem integrated into the JVM. Our subsystem is mostly transparent to application developers and also aware of the component model composing smart home applications. The system accurately accounts for resources consumed during cross-application interactions, and provides on-demand snapshots of memory usage statistics for the different service providers sharing the gateway.

Keywords: resource; sharing conflict; embedded software; component; service; smart home; home automation; Fractal; MIND; Jasmin; OSGi; JVM; Incinerator; stale reference; virtual machine; isolation; monitoring.

Résumé

Nos maisons deviennent de plus en plus intelligentes grâce aux nombreux appareils, capteurs, et actionneurs disponibles, et fournissant des services, tels que le divertissement, la sécurité, l'efficacité énergétique et le maintien à domicile. Divers fournisseurs de services veulent profiter de l'opportunité de la maison intelligente en développant rapidement des services à héberger dans une passerelle domotique embarquée. La passerelle est ouverte à des applications développées par des fournisseurs de services non fiables, contrôlant de nombreux appareils, et pouvant contenir des bugs ou des codes malicieux. Par conséquent, la passerelle doit maintenir une haute disponibilité et suffisamment de robustesse pour tolérer les problèmes logiciels sans avoir à redémarrer brutalement. Partager les ressources, même restreintes, de la passerelle entre les fournisseurs de services leur permet de fournir des services plus riches. Cependant, des conflits de partage des ressources se produisent quand une application utilise les ressources «déraisonnablement» ou abusivement. Cette thèse aborde le problème des conflits de partage des ressources dans la passerelle domotique, investiguant des approches de prévention autant que possible, et envisageant des approches de détection et de résolution quand la prévention est hors de portée.

Notre première contribution «Jasmin» vise à prévenir les conflits de partage des ressources en isolant les applications. Jasmin est un intergiciel pour le développement, le déploiement et l'isolation des applications natives, à base de composants et orientées services prévues pour des systèmes embarqués. Jasmin permet une communication rapide et facile entre applications, et utilise les conteneurs Linux pour une isolation à faible coût. Notre seconde contribution «Incinerator» est un système dans la machine virtuelle Java (JVM) qui résout le problème des références obsolètes en Java, c.-à-d., des références à des objets à ne plus utiliser. Les références obsolètes peuvent causer des fuites mémoire importantes dans une passerelle domotique basée sur OSGi, diminuant ainsi la quantité de mémoire disponible, ce qui augmente les risques de conflits de partage de mémoire. Avec un coût inférieur à 4%, Incinerator non seulement détecte les références obsolètes, les rendant visibles aux développeurs, mais aussi les élimine, diminuant ainsi les risques de conflits de partage de ressources. Même en Java, les conflits de partage de mémoire se produisent. Afin de les détecter, nous présentons notre troisième contribution : un système de surveillance de mémoire intégré à la JVM. Notre système est pratiquement transparent aux développeurs d'applications et conscient du modèle à composants formant les applications domotiques. Le système compte précisément les ressources consommées pendant les interactions entre applications, et fournit, à la demande, des statistiques instantanées d'utilisation de mémoire pour les différents fournisseurs de services partageant la passerelle.

Mots clés : ressource ; conflit de partage ; logiciel embarqué ; composant ; service ; maison intelligente ; domotique ; Fractal ; MIND ; Jasmin ; OSGi ; JVM ; Incinerator ; référence obsolète ; machine virtuelle ; isolation ; surveillance.

مُلَخّصٌ

بُيُوتُنَا تَزْدَادُ ذَكَاءً نَظرًا لتَعَدُّد المُعَدّات وَأَجْهِزَة الإِسْتِشْعَارِ وَالمُشَغّلات دَاخلَها، وَالتِي تُوَفّرُ خِدْمَات كَالتّرْفيه، أَمْن المَنْزِل، فَعَالِيّة إِسْتِهْلاك الطّاقَة وَالرعَايَة الصّحّيّة. عَديدُ مُقَدّمِي الخِدْمَات يَرْغَبُون فِي الإِسْتِفَادَة من فُرْصَة المَنْزِل الذّكِي عَبْرَ تَطْوِيرٌ سَرِيعٌ لخِدْمَات يَقَع تَشْغِيلُهَا فِي بَوّابَةٌ مُدْمَجَة بِالمَنْزِل. البَوّابَةُ مَفْتُوحَةٌ لِتَطْبِيقَات تَتَحَكّم بعِدّة أَجْهِزَة، يُطَوّرُهَا مُقَدّمُوا خِدْمَات غَيْرُ مَوْثُوق بِهِم، وَقَدْ تَحْتَوِي عَلَى أَخْطَاء أَوْ عَلَى بَرْمَجِيّات خَبِيثَة. لِذَلِك، وَجَب أَن تَكُون البَوّابَةُ مُتَاحَةً بِنِسْبَة عَالِيَة وَمَتِينَةً بِمَا يَكْفِي لِلْتَعَامُل مَع مَشَاكِل البَرْمَجِيّات دُون الحَاجَة إِلى إِعَادَة تَشْغِيل فُجْئِيّة. تَقَاسُم المَوَارِد المَحْدُودَة لِلْبَوّابَة بَيْن مُقَدّمِي الخِدْمَات يُتِيح لَهُم تَقْدِيم خِدْمَاتٌ أَثْرَى. لَكِن صِرَاعَات تَقَاسُم مَوَارِد تَقَع حِين تَسْتَخْدِم أَحَدُ التّطْبِيقَات المَوَارِدَ بِطَرِيقَة غَيْرِ مَعْقُولَة أَوْ مُتَعَسّفَة. هَذِه الأُطْرُوحَةُ تُعَالِج مُشْكِلَةَ صِرَاعَات تَقَاسُم المَوَارِد فِي بَوّابَة المَنْزِل الذّكِي، بِالتّحْقِيق فِي أَسَالِيب الوِقَايَة قَدْرَ الإِمْكَان، وَدِرَاسَة مَنَاهِج الكَشْف وَالعِلاَج حِين تَكُون الوِقَايَةُ بَعِيدَةَ المَنَال.

تَهْدف أَوّل مُسَاهَمَة لَنَا، وَتُدْعَى Jasmin، إِلى الوِقَايَة من صِرَاعَات تَقَاسُم المَوَارِد عَبْرَ عَزْل التّطْبِيقَات. Jasmin هِي بَرمَجِيّة وَسِيطَة لِتَطْوِيرِ وَنَشْرِ وَعَزْل التّطْبِيقَات الأَصْلِيّة القَائِمَة عَلَى العَنَاصِرِ وَذَات المَنْحَى الخِدْمِي وَالتِي تَسْتَهْدف النُّظم المُدْمَجَة. Jasmin تُتِيح تَوَاصُلا سَرِيعا وَسَهْلا بَيْن التّطْبِيقَات، وَتَسْتَخْدِم حَاوِيَات Linux لِلْعَزْل بِتَكْلُفَة مُنْخَفضَة. مُسَاهَمَتُنَا الثّانِيَةُ، Incinerator، هُوَ نِظَامٌ فِي الآلَة الإِفْتِرَاضِيّة لـ Java يَهْدف إِلَى حَل مُشْكِلَة الإِشَارَات البَاطلَة فِي Java، أي الإِشَارَات إِلى الكَائِنَات الّتِي لَم يَعُدْ يَنْبَغِي إِسْتِخْدَامُهَا. الإِشَارَاتُ البَاطلَةُ يُمْكن أَنْ تُسَبّب تَسرُّبَات كَبِيرَة لِلْذّاكِرَة فِي بَوّابَة مَنْزِل ذَكِي مُرْتَكِزَة عَلَى OSGi، وَلِذَا فَهِي تُقَلّل من مسَاحَة الذّاكِرَة المُتَاحَة، ممّا يَزِيدُ من مَخَاطِر صِرَاعَات تَقَاسُم الذّاكِرَة. بِتَكْلُفَة أَقَل من 4%، Incinerator لا يَكْتَفِي بِكَشْف الإِشَارَات البَاطلَة فَحَسْب، لِيَجْعَلَهَا مَرْئِيّةً لِلْمُطَوّرِين، بَل يَقْضِي عَلَيْهَا كَذَلِك، ممّا يُخَفّض من مَخَاطِر صِرَاعَات تَقَاسُم المَوَارِد. حَتّى فِي Java تَحْدُث صِرَاعَات تَقَاسُم الذّاكِرَة. لِكَشْفِهَا، نُقَدّم ثَالث مُسَاهَمَاتنَا: نِظَامٌ لِرَصْد الذّاكِرَة مُدْمَجٌ فِي الآلَة الإِفْتِرَاضِيّة لـ Java. نِظَامُنَا شَفّافٌ عُمُوما بِالنّسْبَة لمُطَوّرِي التّطْبِيقَات كَمَا أَنّه وَاع بِنَمُوذَج المُكَوّنَات المُؤَلّف لِتَطْبِيقَات المَنْزِل الذّكِي. يَحْسب النّظَام بِدقّة المَوَارِدَ المُسْتَهْلَكَة خِلال التّفَاعُلات بَيْن التّطْبِيقَات، وَيُقَدّم عِنْدَ الطّلَب لَقَطَات مِن إِحْصَائِيّات إِسْتِخْدَام الذّاكِرَة بَيْن مُقَدّمِي الخِدْمَات الذِين يَتَقَاسَمُون البَوّابَة.

كَلِمَاتٌ مَفَاتِيح: مَوَارِدٌ، صِرَاع تّقَاسُم، بَرْمَجِيّاتٌ مُدْمَجَةٌ، مُكَوّنٌ، خِدْمَةٌ، بَيْتٌ ذَكِي، تّشْغِيلٌ آلِي لِلْمَنْزِل، Fractal، MIND، Jasmin، OSGi، JVM، Incinerator، إِشَارَةٌ بَاطِلَةٌ، آلَةٌ إِفْتِرَاضِيّةٌ، عَزْلٌ، رَصْدٌ.

Acknowledgments

I can fairly rate the three years of my Ph.D. as a good experience on many levels. It allowed me to focus on a specific theme I like for a period long enough to gain expertise and to perceive the challenges of the domain. I had the freedom of exploration and enough challenges to feed my eager passion of software development. But the best thing I appreciated in my Ph.D. thesis is that I learned, way many things!

I knew many interesting people during the Ph.D. work at Orange Labs, and during my visits to LIP6. I am very grateful to those people who helped me achieve the Ph.D. work and contribute to the scientific community. I want to say a warm "*Thank you for all your efforts!*" to my Ph.D. supervisors André Bottaro, Gilles Muller and Gaël Thomas in addition to my Master supervisor Matthieu Anne. I would like to thank the members of my Ph.D. defense Jury: Béatrice Berard, Didier Donsez, Jean-Philippe Fassino, Johann Bourcier and Laurence Duchien. I am also thankful to the people who proof read my documents and helped me with their constructive remarks: Christophe Demottié, Jacques Pulou, Julia Lawall, Lobna Jilani, Olivier Beyler, Ravi Mondi and Yoann Maurel.

I am very grateful to my friends who helped boost my moral state when I was near the edges... *Thank you* Achraf Bichiou, Ahmed Ouertani, Ali Abdelkafi, Ayoub Maatallaoui, Chayma Souihli, Clara Murcia Castillo, Faouzia Daoud, Hamdi Bouafsoun, Imen Zaabar, Issam Baghdadi, Khalil Negrichi, Lassaad Najjar, Lobna Jilani, Mahdi Kettani, Mohamed Ibn El Azzouzi, Mohamed Salmaoui, Tarek Toumi and Yoann Maurel.

I probably would not have achieved all of this without the continuous efforts and presence of my family, even from a distance. I am deeply grateful to my grandfather Ahmed, my wife Zeyneb, my parents Mohamed and Azza, my parents-in-law Habib and Souad, and the rest of my family, mainly Habiba, Maher, Khadija, Monji, Ayatoullah, Malek and Jihed.

I hope I made you all proud of me and of the work I performed during this Ph.D.

Thank you again for your help and support
Koutheir Attouchi

Contents

Abstract i

Résumé iii

مُلَخَّصٌ v

Acknowledgments vii

Introduction xiii

1 State of the Art **1**
1 Introducing the Smart Home . . . 1
2 The Dynamicity of the Smart Home . . . 5
2.1 Hot-swapping Native Applications . . . 5
2.2 Hot-swapping Java Applications . . . 6
3 The Distributed Aspect of the Smart Home . . . 8
4 Rapid Application Development in the Smart Home . . . 9
4.1 The Component Model . . . 10
4.2 The Fractal Component Model . . . 11
4.3 The OSGi Component Model . . . 13
4.4 The OSGi Declarative Services Component Model . . . 15
5 The Heterogeneity of the Smart Home . . . 16
5.1 The Service-Oriented Architecture . . . 16
5.2 OSGi as a Service-Oriented Architecture . . . 18
6 The Open and Embedded Aspects of the Smart Home . . . 19
6.1 Isolation based on Hardware Protection and Virtualization . . . 20
6.2 Isolation based on Language Type Safety . . . 25
6.3 Discovering Resource Sharing Conflicts via Monitoring . . . 27
6.4 Stale References – A Typical Source of Memory Conflicts . . . 31
7 Conclusion . . . 36

2 Jasmin – Isolating Native Component-based Applications **39**
1 Jasmin Component-Based and Service-Oriented Applications . . . 40
2 The Standalone Jasmin Architecture . . . 41
3 The Distributed Jasmin Architecture . . . 42
3.1 Architectural Separation of Jasmin Applications . . . 42
3.2 Jasmin Distributed Service-Oriented Architecture . . . 43
4 Multi-Level Isolation of Jasmin Applications . . . 43
4.1 Zero Isolation . . . 44

4.2 Process-based Isolation . . . 44
4.3 Container-based Isolation . . . 44
5 Transparent and Fast Communication between Jasmin Applications . . . 45
6 Evaluation . . . 46
6.1 Communication Speed Benchmarks . . . 46
6.2 Disk Footprint . . . 47
6.3 Memory Footprint . . . 48
6.4 Performance and Porting Efforts of a Legacy Multimedia Application . . . 48
6.5 Summary of Results . . . 51
7 Conclusion . . . 51

3 Incinerator – Stale references detection and elimination 53
1 Memory Leaks and State Inconsistencies . . . 54
2 Introducing Incinerator . . . 54
3 Detecting and Eliminating Stale References . . . 55
3.1 Stale Class Loaders . . . 56
3.2 Synchronization Handling in Incinerator . . . 56
3.3 Finalization Handling in Incinerator . . . 57
4 Implementation Extents . . . 59
4.1 Changes to the Java Virtual Machine . . . 59
4.2 Monitoring Bundles Updates and Uninstallations . . . 61
4.3 Modifications to the Just-In-Time Compiler . . . 61
5 Functional Validation and Performance Evaluation . . . 62
5.1 Stale Reference Micro-Benchmarks . . . 62
5.2 Bundle Conflicts . . . 65
5.3 Memory Leaks . . . 66
5.4 Stale References in Knopflerfish . . . 67
5.5 Performance Benchmarks . . . 67
6 Conclusion . . . 68

4 OSGi-Aware Memory Monitoring 71
1 Memory Monitoring Issues . . . 71
2 Our Solution to Memory Monitoring in OSGi . . . 73
3 Accurate and Relevant Memory Monitoring . . . 73
3.1 Assumptions . . . 74
3.2 Goals . . . 74
3.3 Domain-Specific Language for Resource Accounting Rules . . . 75
3.4 Resource Accounting Algorithm . . . 77
4 Implementation Details . . . 78
4.1 OSGi State Tracker . . . 79
4.2 Accounting Configuration Manager . . . 79
4.3 Monitoring Manager . . . 81
5 Functional Validation and Performance Evaluation . . . 81
5.1 Functional Tests . . . 81
5.2 Performance Micro-Benchmarks . . . 85
5.3 DaCapo Benchmarks . . . 87
6 Conclusion . . . 88

Conclusion 89

Appendices 93

A Résumé de Thèse 95
1 Introduction . . . 95
1.1 Problématique . . . 96
1.2 Contributions . . . 96
1.3 Structure de Document de Thèse . . . 98
2 État de l'Art . . . 99
3 Jasmin . . . 101
4 Incinerator . . . 102
5 Surveillance de Mémoire en OSGi . . . 105
6 Conclusion . . . 106

Bibliography 111

List of Figures 123

List of Tables 125

Introduction

Our homes are full of electronic devices that assist our life in various ways. An increasing number of these devices is becoming smarter, i.e., they are being equipped with more processing power, more sensors and actuators and better connectivity with the surrounding devices and the internet. These smart devices are becoming more and more popular at home due to the advanced services they provide at an affordable cost. The services include entertainment, home security, energy efficiency, health care, well-being, comfort, and content sharing. Thanks to these devices, we are becoming the inhabitants of *smart homes*.

The smart home offers new opportunities to various service providers, who long to rapidly develop and deploy services that take advantage of, not only the devices present at home, but also services offered by other providers. To allow cohabitation of different service providers, smart home operators are designing a *gateway* that hosts applications provided by different actors [59, 112]. The gateway provides basic services to hosted applications, including deployment, connectivity, data storage, and service discovery. In order to allow the sharing of devices and services between hosted applications, smart home operators are standardizing an interface that is used to provide and access the services provided by smart devices and providers. For example, one service provider could use the standard interface to control a device manufactured by another service provider that conforms to the same interface.

The applications running on the smart home gateway control actuators and devices inside the home, and some of those devices handle security and health of the inhabitants, such as gas sensors, air conditioners, door alarms, etc. Therefore, abruptly restarting the gateway can be dangerous. In fact, the smart home gateway is a *long-running platform* that needs to be *highly available* and robust enough to handle software errors without the need to restart abruptly. This long-running nature is the reason the smart home gateway supports application *hot-swapping*, i.e., starting and stopping and updating of applications on the fly, without the need to restart the whole environment. To implement hot-swapping, hosted applications are made of *components* that can be individually started and stopped and replaced.

Addressed issues

Sharing of the smart home resources between service providers is necessary to provide rich and integrated services to the end-user, while keeping the choice to the user to freely mix smart devices and services from different hardware manufacturers and software editors. However, resource sharing naturally raises the risk of resource conflicts between service providers, such as resource starvation, illegal access to resources, and abusive use of resources. These conflicts threaten confidentiality, and slow down the gateway performance, and deprive correct sharing of available resources.

In this thesis, we study the problem of resource sharing conflicts in the smart home gateway from different perspectives. We confront the issue of *preventing* resource conflicts from happening in the first place. But since we cannot prevent all resource sharing conflicts, we also focus on *resolving* the conflicts when they happen or when they are discovered. Discovering resource conflicts needs resource *monitoring* approaches, which is our third concern.

The applications deployed by service providers can contain bugs. Those bugs may lead to injection of malicious behavior into the applications. These risks are the reasons *we do not trust* applications developed by service providers [79], which is why the solutions we present tend to avoid, as far as possible, to delegate additional responsibilities to application developers.

To summarize, we argue that the smart home gateway needs an open software environment that provides the following features:

1. Prevents resource sharing conflicts as far as possible, using different forms of isolation and protection.

2. Monitors resources conflicts that still can happen, and quantifies resource usage as accurately as necessary.

3. Resolves resource sharing conflicts when they happen, and makes them visible to application developers or system administrators.

Contributions

In this thesis, we present three contributions.

We began by investigating ways to *prevent* resource sharing conflicts between *native* applications running in the smart home gateway. The most obvious way to prevent conflicts is isolation. So we wanted to achieve the highest isolation possible, but without hindering performance. Among the different isolation mechanisms available, the *containers* technologies were the best bet fulfilling our requirements. The Linux containers isolate applications' memory, network communications, files, users, etc. We created *Jasmin*: an execution environment that isolates native component-based and service-oriented applications inside containers provided by the Linux kernel. Jasmin [4] configures and deploys containers and runs applications inside them, while controlling applications life cycles. Jasmin also resolves services between containers and provides transparent service invocation mechanisms.

Some resource sharing conflicts cannot be prevented due to the highly dynamic software environment envisioned for the smart home, and due to the diverse devices present at home. Therefore, we also explore approaches to *resolve* conflicts between applications running in the smart home gateway. One of the issues we addressed is the problem of *stale references*, which is common in platforms that support hot-swapping. Hot-swapping enables updating or uninstalling applications without restarting the platform. In normal situations, when an

application is uninstalled, all other applications remove their references to it, in order to allow the gateway to remove the uninstalled application from memory. However, if a buggy application keeps holding a reference to the uninstalled application, then that reference is known as stale reference. The stale reference forces the gateway to keep the uninstalled application in memory, thus causing a significant *memory leak*. If the buggy application tries to use the uninstalled application via its stale reference, then the results are *undefined*, and the states of running applications can become *inconsistent*, because the uninstalled application does not expect to be invoked after it has executed its termination routines during its uninstallation event. If the uninstalled application was controlling an actuator, then the state inconsistency can damage the controlled hardware, therefore threatening the safety of smart home inhabitants. To solve this problem, we created *Incinerator*, a system that pinpoints stale references and removes them. After hot-swapping an application, Incinerator investigates all references in the gateway, looking for stale references. When a stale reference is found, Incinerator removes it, and disallows the buggy application from using that reference in the future, and performs cleanup that should have been done by the buggy application. By finding stale references, Incinerator helps developers debug this problem which is hard to perceive. By removing stale references, Incinerator not only lowers the risk of state inconsistency, but also avoids the memory leak caused by stale references, thus allowing the gateway to continue working without running out of memory. The Incinerator prototype was tested using Knopflerfish [81], one of the main open-source OSGi [127] implementations. Thanks to Incinerator, we discovered and fixed a stale reference bug [8] in Knopflerfish.

In order to maintain a long-running system stable and fully functional, resource usage needs to be monitored in order to discover situations of resource sharing conflicts, e.g., abusive resource usage, resource starvation, etc. Being a long-running system, the smart home gateway needs a monitoring system that supports its openness to multiple service providers, and that provides accurate memory usage reports. In particular, during an interaction between two applications, the monitoring system needs to be able to distinguish which application should be accounted for the memory consumed during the interaction. That is the reason we developed a monitoring system to report the memory consumed by applications hosted by different service providers on the gateway. Our monitoring system allows different applications to communicate to render services, so it does not require special isolation mechanisms. In most cases of interaction between service providers, the accounting approach used by the system produces accurate reports of memory usage by application. Furthermore, we defined a domain-specific language that is used to declare explicit accounting rules that dictate which application should be accounted for the memory consumed during the specific interaction. The language is used to declare rules in a simple and readable way. It is also flexible, as it allows using wildcards to define an interaction in an accounting rule that would match several interactions at runtime, which helps reduce the number of rules to write and maintain. Our monitoring system prototype was tested using the Knopflerfish OSGi implementation.

Document overview

This thesis is structured in four main chapters. The first chapter begins by exploring the state of the art in order to describe the existing ecosystem of the smart home, focusing on

the particular properties of this environment, and the challenges they induce. We detail those challenges in order to illustrate the conceptual and technical locks that still need to be resolved or whose existing solutions should be enhanced. We illustrate why the dynamicity of the smart home requires hot-swapping of applications. Then we discuss the distributed nature of the smart home and its effect on cross-application communication. We argue that service providers need to rapidly develop smart home applications, and we illustrate how the component model can help achieve that. Furthermore, we show how the service-oriented architecture helps simplify the heterogeneity of the smart home. Finally we illustrate the concepts and efforts needed to support the open and embedded nature of the smart home gateway.

The second chapter describes our first contribution, i.e., Jasmin. It first describes two architectures of Jasmin that could be used in different kinds of smart home gateway. Then we describe the multi-level isolation support provided by Jasmin based on processes and on Linux containers. Then we discuss the transparent and fast cross-application communication mechanism provided by Jasmin, and how it helps provide a distributed service-oriented platform. We finally evaluate Jasmin's performance and features using micro-benchmarks, and we discuss the effort needed to port legacy applications to Jasmin using a typical smart home application.

In the third chapter, we present Incinerator, the second contribution. We recall the bug of Java stale references introduced in the first chapter, then we explain how Incinerator detects and resolves this issue. Next, we show the areas that needed modifications in order to implement Incinerator. We then evaluate the features of Incinerator using ten scenarios where references become stale. We also demonstrate that stale references can endanger the smart home inhabitants if a buggy application controls critical home devices. Finally we evaluate the performance of Incinerator using the DaCapo benchmarks.

Our third contribution is presented in the fourth chapter, where we begin by describing the problem of accurate and relevant memory monitoring in an OSGi-based platform open to multiple untrusted service providers. We discuss our design goals and we describe the algorithm we defined to provide an accurate and relevant memory monitoring system for the smart home gateway. We also describe the implementation extents of our system, before evaluating the features and performance of the system using micro-benchmarks and using the DaCapo benchmarks.

We conclude our thesis by resuming the issues we address and the contributions we propose, and by suggesting further actions to take and paths that still need exploration.

Chapter 1.

State of the Art

Contents

1 Introducing the Smart Home **1**
2 The Dynamicity of the Smart Home **5**
2.1 Hot-swapping Native Applications 5
2.2 Hot-swapping Java Applications 6
3 The Distributed Aspect of the Smart Home **8**
4 Rapid Application Development in the Smart Home **9**
4.1 The Component Model 10
4.2 The Fractal Component Model 11
4.3 The OSGi Component Model 13
4.4 The OSGi Declarative Services Component Model 15
5 The Heterogeneity of the Smart Home **16**
5.1 The Service-Oriented Architecture 16
5.2 OSGi as a Service-Oriented Architecture 18
6 The Open and Embedded Aspects of the Smart Home **19**
6.1 Isolation based on Hardware Protection and Virtualization 20
6.2 Isolation based on Language Type Safety 25
6.3 Discovering Resource Sharing Conflicts via Monitoring 27
6.4 Stale References – A Typical Source of Memory Conflicts 31
7 Conclusion **36**

This chapter describes the *smart home* environment, which is the real life domain where our research and propositions are projected. We begin by defining the smart home and inspecting its particular properties. We then focus on each property to extract the challenges it implies and the existing work tackling these challenges. This focus also enlightens areas that still need to be investigated and problems waiting to be resolved, or needing better solutions.

1. Introducing the Smart Home

The smart home is a home full of communicating electronic devices that collaborate in order to provide intelligent services to the home inhabitants, as illustrated in Figure 1.1. The devices

include *sensors* for temperature, presence detection, gas detection, etc. Different *actuators* are also present, e.g., motorized window curtains, controllable gas valves, motorized doors. Smart phones, smart watches, and different kinds of *personal devices* can be present at the smart home from time to time. This wealth of devices offers a new market to many device manufacturers and software editors (denoted by ★, ♣ and ⧫ in Figure 1.1), allowing them to create applications targeted at the different inhabitants of the home (see App_1...App_6 in Figure 1.1), providing services in different domains, e.g., security enforcement, energy efficiency, comfort and health care, helping with home support, enabling content sharing, etc.

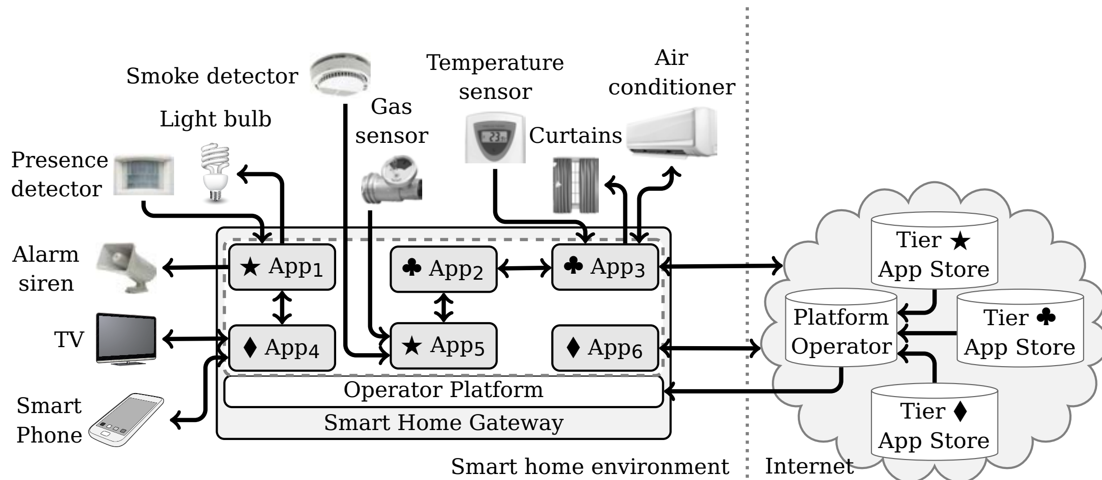


Figure 1.1: Smart home structure. Communications between the "Operator Platform" and App_1,...,App_6 are omitted to simplify the figure.

The Smart Home Gateway. The smart home, as we envision, includes one or more devices that host different applications developed by different tiers. As an example, the Home Gateway Initiative (HGI) [59, 112], which is an industrial alliance of telecommunication operators targeting the smart home, envisions a *gateway* device that is connected to the different smart home devices and to the Internet, known as the *smart home gateway*. Figure 1.1 illustrates this gateway. The "Operator Platform" layer provides a middleware and common services to applications, e.g., persistence, logging, access to smart home devices, network services, etc. The operator platform notifies applications about different events occurring either on the platform, or in the smart home environment, or on the internet, e.g., new sensors' data, notifications for application start-up, new services available, etc. Every software editor deploys a set of applications that can communicate, not only with each other, but also with applications from other vendors, and with the operator platform. Applications communicate in order to provide integrated services to the end-user.

The Smart Home Properties

The smart home is a particular hardware and software environment. This section describes the main properties of the smart home, and describes the challenges introduced by each property. Each challenge raises a set of issues that need to be tackled. We detail those issues throughout this chapter, and we present existing solutions addressing them along with their limits.

Openness. Devices in a typical home are often created by different hardware manufacturers. Along with devices, applications running in the devices or supporting them are also developed by different software editors. The smart home inhabitants do not need to be locked to a particular vendor. This *openness* of the smart home to different tiers raises the need for an open smart home gateway that is able to host applications from different untrusted tiers.

Dynamicity. The smart home is *dynamic* at the hardware level, as different devices appear and disappear all the time, and several devices are movable. We expect the smart home to be dynamic also at the software level, where applications are installed, updated and uninstalled at a fast pace compared to the "up time" of the smart home gateway, which is a long-running system. Therefore, applications need to be designed to cope with this dynamic nature. *Hot-swapping* solves this issue by starting and stopping applications as necessary to cope with, not only hardware changes happening in the smart home, but also relatively frequent software updates and the continuous end-user desire to install new applications providing innovative services.

Rapid Development. Thanks to the wealth of devices present in it, the smart home offers new opportunities to service providers, who long to develop *ubiquitous* applications that take advantage of those devices, and of the physical proximity to the end-user. In order to catch the rising smart home market, and to enable faster innovation pace, the service providers want to *rapidly* develop and deploy services at the smart home. Rapid development needs simple tools and processes to manage applications life cycle, especially development tools and languages, without forgetting deployment and maintenance tools. Applications need to be easily created using reusable building blocks that can be composed quickly. These building blocks need to be loosely coupled in order to allow easy upgrades and maintenance. The *component model* is a set of rules and processes to easily compose applications out of basic building blocks called *components*, urging for code reuse and encapsulation. A component model also defines what is a component from what is not, and describes the process of creating individual components. The logic in each component is highly "encapsulated" inside it, and interaction between components only happens through interfaces, which implies that components are relatively independent of each other, making them good candidates for hot-swapping mechanisms. For these reasons, we believe that smart home applications should be developed based on a component model.

Heterogeneity. The devices and applications of the smart home have various capabilities for use by the home inhabitants. Some devices and applications provide different variations

of the same capability, e.g., voice calling versus video calling. Other devices and applications provide different implementations of the same capability, e.g., voice calling via the *Global System for Mobile Communications* (GSM) network or voice calling via the *Voice over Internet Protocol* (VoIP). To cope with this *heterogeneity*, and in order for smart home applications to use these capabilities, the features provided by different tiers need to be standardized as contracts that are independent of their implementation details and vendors. This the reason smart home applications need to be *service-oriented*, based on standard interfaces, where a service can be defined as a contract-based controlled access mechanism to a set of capabilities. The *Service-Oriented Architecture* (SOA) [103, 78] allows applications and devices to provide and consume standard services independently of their vendors, implementations, etc.

Distributed Aspect. In order to provide fully integrated services to the end-user, often multiple devices and applications need to collaborate and communicate. The smart home devices and applications are often physically distant, and they are connected via diverse networking technologies of different scales, ranging from Personal Area Networks (PAN, e.g., Infrared, Bluetooth, ZigBee, Z-Wave) to Local Area Networks (LAN, e.g., Ethernet, WiFi) and even to Internet. This highlights the *distributed* nature of the smart home, where software running in devices, in the gateway and in the internet collaborate to offer smart services to the home inhabitants. This is why applications deployed on the smart home gateway and delivered by different smart home actors should be able to *communicate easily*.

Embedded Aspect. Constrained by its *cost*, the smart home gateway is an *embedded* system that has limited hardware resources. We expect the gateway to have a micro-controller or a micro-processor running at hundreds megahertz, using tens to hundreds megabytes of volatile and non-volatile memory. Existing affordable single-board computers such as the “Raspberry Pi” [108] and the “Arduino Yún” [6] already exhibit such performances. This reduces the choice of technologies allowing *agile* [17] software development, as some of these technologies induce high overheads that are not affordable under embedded constraints. For instance, Java [55] is one of the commonly used Object-Oriented technologies allowing agile software development. Java has several *profiles* suitable to different levels of embedded constraints, e.g., Java Mobile Information Device Profile (MIDP) [110], Java Connected Device Configuration (CDC) [36, 123], Java ME Embedded Profile [15], etc. One of the main component-based and service-oriented platforms based on Java and compatible with the embedded constraints is the OSGi [30, 127, 26, 82, 54] framework, which can be used to develop smart home applications. We also expect some smart home gateways to be further constrained, i.e., based on a micro-controller or a micro-processor running at tens megahertz, using hundreds kilobytes of volatile and non-volatile memory. For these gateways, virtual machines are often unaffordable, which raises the need for native applications. As an example, the MIND [92] framework, which is an implementation of the Fractal [29] component model, can be used to write native component-based applications.

2. The Dynamicity of the Smart Home

The dynamic nature of the smart home raises the need for hot-swapping applications, i.e., loading, unloading and updating applications on the fly without the need to restart the platform. In order to perform hot-swapping, application code should be *loadable* into a running middleware platform, and *unloadable* when the application is no longer needed. The exact hot-swapping mechanism depends on the nature of the operator platform middleware and the applications hosted on it.

In this section, we focus on hot-swapping mechanisms for native and Java applications. First, we present position-independent code and how it can be used to load a native application into a running process. Then we define Java class loaders and the structure of Java references, which justifies the class loader *unloading conditions* in the Java virtual machine. Finally, we present a concrete example where these unloading conditions represent a conceptual and a technical lock for hot-swapping of dynamic Java applications: multi-tenant Java virtual machines.

2.1. Hot-swapping Native Applications

A *native* application is an application that *directly* uses the Instruction Set Architecture (ISA) [115] of the processors running the computer. By "directly", we mean that no interpretation or binary translation is needed in order to run the application. In order to run a standalone native application, an operating system loads the application into memory in a dedicated *virtual address space*, i.e., a *process*, often at a specific *base address* dictated by the application binary meta-data. The machine code contained in the standalone application assumes that it is always loaded at the specified base address, and uses that assumption to calculate absolute addresses of functions and variables and embeds them into the application binary. Embedding of absolute addresses avoids unnecessary calculations of addresses at runtime, therefore improving application performances.

Position-Independent Code

The constant base address assumption relies on the dedicated address space given by the process abstraction. Thus, applications that share the same address space generally cannot assume a constant base address, which obliges them to use *relative addressing*. Machine code that does not rely on a constant base address is called a *Position-Independent Code* (PIC) [63, 85, 128]. Position-independent code forms the basic mechanism through which many operating systems provide the ability to load machine code into a running process. The operating system dynamic loader loads the position-independent code into memory, then it performs necessary data initialization and linking with other required libraries. Loadable machine code units are called *Shared Objects* (.so) in UNIX and Linux, and called *Dynamic-Link Libraries* (.dll) in Windows.

Native applications generated in the form of loadable code units can be loaded by the operator platform middleware, based on the operating system facilities. Once loaded, the application code can be executed as any other code in the platform, so it can be started, stopped and called back when events occur.

2.2. Hot-swapping Java Applications

A Java application is distributed as a set of *Java binary classes*. A Java binary class is a collection of intermediate code, called *Java byte code*, and *meta-data* describing that byte code. The meta-data describe Java class name, fields information, methods information, etc. The Java byte code is a sequence of virtual instructions in the form of a UNiversal Computer-Oriented Language (UNCOL) [34, 116, 117, 89, 5] that conforms to a Virtual Instruction Set Architecture (V-ISA) defined by the Java virtual machine specifications [76].

2.2.1. Java Class Loaders

The standard mechanism to load Java binary classes into a running Java virtual machine is the *Java class loader*, which is a Java object whose purpose is to return a Java class instance (i.e., a `java.lang.Class` object) given a fully qualified class name[1]. The JVM specifications assert the existence of a *primordial class loader* provided by the JVM, and allows applications to define new class loaders. New class loaders can reuse a part of the functionality provided by the primordial class loader.

The Java virtual machine identifies a loaded class uniquely by its name and its class loader object, as follows: $LoadedClass = \langle ClassLoaderObject, ClassName \rangle$. This identification implies that a class loader object defines a *name space* in which its classes are identified separately from the classes loaded by other class loaders. Therefore, two classes with the same name loaded on the same JVM instance using different class loaders are considered different types. This name space isolation enables loading multiple applications, each within a dedicated class loader, without risking naming conflicts.

2.2.2. Class Loader Unloading Conditions

In Java [55], an object reference is, by default, *strong* [119], i.e., the object reference not only gives access to the pointed-to object, but also guarantees that the object will not be collected by the garbage collector. In other words, an object will remain in memory as long as there is one or more strong references to it somewhere in the JVM. As strong references are the default reference type in Java, we refer to them simply as "references", unless otherwise specified.

A garbage collection *root* is either a Java `static` reference (e.g., a Java class field, a thread-local variable) or a Java object reference in a running method (i.e., a method local variable).

[1] In this document, we refer to "fully qualified class name" simply by "class name".

A Java object is *reachable* if it can be accessed by traversing the object graph in some sort, starting from a garbage collection root, and following references.

Figure 1.2 shows that a Java object holds a reference to its class, and a class holds a reference to its class loader. Additionally, a class loader holds references to all the classes it loaded. The garbage collector [64] can collect an object when it becomes unreachable. Collecting a Java class C_i implies reclaiming the byte code of the methods of C_i, their generated machine code, and associated information, e.g., C_i's fully qualified name, hierarchy, and fields. Therefore:

Property 1.1 (Class loader collecting condition. See Figure 1.2.)**.** A class loader L that loaded classes $C_1, \ldots, C_N$ can only be collected when the graph of objects $\{L, C_1, \ldots, C_N\}$ becomes unreachable, which implicitly requires that, for each C_i, all objects of C_i are unreachable.

O_1 is an object of class C_1.
O_2 and O_3 are objects of class C_2.
C_1 and C_2 were loaded by L.

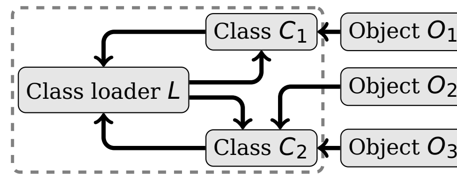


Figure 1.2: Java references graph between objects, classes, and class loaders.

Property 1.2 (Class loader lifetime)**.** In Java [55], the lifetime of a class loader encompasses the lifetimes of all the classes it loads, i.e., a class loader is created *before* it can load classes, and it is collected *after* all its loaded classes get collected. Moreover, the lifetime of a class encompasses the lifetimes of all objects of the class, i.e., a class is loaded *before* any objects of it are created, and it is collected *after* all its objects get collected. Consequently, the lifetime of a class loader encompasses the lifetimes of all the classes it loads, and all objects of those classes.

2.2.3. Hot-swapping in Multi-Tenant Java Virtual Machines

A Java virtual machine is called *multi-tenant* [62, 11] when it can run multiple Java applications in the same time. Multi-tenancy is often used when a system needs to run multiple Java applications, but due to hardware constraints, the system cannot run multiple Java virtual machines, one for each application.

In multi-tenant Java virtual machines, class loaders are often used to load entire applications. A given class loader will load every Java binary class composing the application *on-demand*, i.e., the first time the class is accessed. Due to Property 1.1, collecting classes only happens *in bulk*, i.e., all classes loaded by a class loader must be collected before the class loader itself can be collected. Practically speaking, a Java application loaded by a class loader can only be collected as a whole, with all its objects, classes, and the class loader itself, when all these become unreachable.

Consequently, if an application keeps a stale reference (i.e., an unneeded reference) to a second application that should be unloaded (for example, because it is being uninstalled),

then the second application would never be reclaimed from memory, therefore causing a significant memory leak. This is why a system based on a multi-tenant Java virtual machine is vulnerable to stale references, especially when the system is required to run for long periods of time.

3. The Distributed Aspect of the Smart Home

A smart home application often needs to communicate with devices and with other applications, in order to compose services and provide an integrated experience to the end-user. Therefore, communication mechanisms need to be easy to use, and as fast as possible.

Communication mechanisms between applications depend on whether they can share memory or not. On one hand, applications that can share memory often can communicate using local procedure calls. On the other hand, applications that can only communicate via *messages* typically call each other using *remote procedure calls*. We briefly describe these two procedure call mechanisms in this section.

Local Procedure Calls. Two applications running in the same address space can communicate via *local procedure calls*. In order for an application to call a local procedure, the application must adhere to a specific calling convention [7] agreed upon by the procedure, e.g., standard call convention (`stdcall`), fast call convention (`fastcall`), C declaration call convention (`cdecl`), Pascal calling convention (`pascal`), etc. A calling convention describes the protocol of calling the local procedure, e.g., where and how arguments are passed, what exceptions can be thrown, what execution context is expected by the procedure, etc. The calling convention also describes the return protocol, i.e., how the procedure should return control to its caller. Calling a local procedure generally requires few processor cycles, which is relatively fast. Local procedure calls require memory sharing between the calling application and the called application, which is intrinsically possible when both applications run in the same address space.

Remote Procedure Calls. When applications cannot share memory, they cannot communicate using local procedure calls, and they rather use *message-based* communication mechanisms, such as *remote* procedure calls. In order for a procedure P_1 in an application App_1 to call a remote procedure P_2 in an application App_2, the following steps are usually required:

1. P_1 calls a remote procedure proxy in App_1, passing it the remote procedure information, e.g., remote application, procedure name, input parameters.

2. App_1 *serializes* all the input parameters of P_2 into a buffer B_{in}. Serialization, a.k.a, *marshaling*, is the process of packing a list of parameters into a contiguous buffer suitable for sending in a message.

3. App_1 sends B_{in} through a *communication channel* to App_2, and waits for a reply from App_2.

4. App_2 receives B_{in}, then it *deserializes* the buffer. Deserialization, a.k.a, *unmarshaling* is the process of extracting a list of parameters from a serialized buffer.

5. App_2 calls P_2 via a local procedure call, passing the parameters extracted from B_{in}.

6. P_2 returns control to App_2.

7. App_2 serializes all the output parameters of P_2 into a buffer B_{out}, then it sends the buffer through the communication channel to App_1.

8. App_1 receives B_{out}, then it deserializes the buffer.

9. App_1 then returns control to P_1, passing it the output parameters extracted from B_{out}.

Serialization is usually a significantly slow and expensive operation, especially compared to merely passing memory addresses in local procedure calls. In fact, a remote procedure call can be more than ten times slower than a local procedure call, as illustrated in Table 2.1 page 46. Examples of remote procedure protocols include Remote Procedure Calls (RPC) [118], Java Remote Method Invocation (RMI) [133], Distributed Component Object Model (DCOM) [75], Incommunicado [102], Internet Inter-ORB Protocol (IIOP) [39] based on the Common Object Request Broker Architecture (CORBA) [91].

4. Rapid Application Development in the Smart Home

Given the numerous devices present in it, the smart home offers new opportunities to many service providers, who plan to develop *ubiquitous* applications that take advantage of these devices, and of the physical proximity to the home inhabitants. To innovate fast, service providers need to be able to *rapidly* develop and deploy services at the smart home. This raises the need for simple tools and processes to manage applications life cycle, including development tools and languages, and deployment and maintenance tools. Developers should be able to easily create applications from reusable building blocks that can be composed quickly. These building blocks need to be loosely coupled in order to allow easy upgrades and maintenance.

The *component model* is a set of rules and processes to easily compose applications out of basic building blocks called *components*, urging for code reuse and encapsulation. Each component encapsulates a business logic that interacts with the platform and other components only through interfaces, therefore components are relatively standalone entities, making them easily hot-swappable. These are the reasons for choosing the component-based model as a design approach for smart home applications. In this section, we characterize the component model, then we present three examples of components models that are suitable for the smart home gateway.

4.1. The Component Model

As stated by Crnković et al. [38], different definitions of the concept of a software *component* exist in literature. The commonly used definition is the following:

Definition 1.3 (Software component, defined by Szyperski et al. [122])**.** "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party."

There are two types of interfaces: (1) *operation-based* interfaces often used in software design, and (2) *port-based* interfaces, often used to design hardware systems [38]. As we focus our definitions on software design paradigms, we define an interface as a set of operations, a.k.a., methods. Each *operation* is identified by a name and a list of parameters that are input or output from the component providing the interface. Formally speaking:

Definition 1.4 (Operation-based interface formal structure)**.** An operation-based interface is a standalone contract that can be described as follows:

$$Interface = \langle Identifier, \{Operation_0, \ldots, Operation_n\}\rangle$$
$$Operation = \langle Identifier, \emptyset | \{Parameter_0, \ldots, Parameter_m\}\rangle$$

An interface is *standalone*, as it can exist independently of components, in order to formalize standard contracts that can be fulfilled by components or required by other components.

Definition 1.5 (Software component formal structure)**.** A component C can be formally characterized as follows:

$$Component = \langle Identifier, ProvidedInt, RequiredInt, Properties, Implementation\rangle$$

where $ProvidedInt$ is a set of interfaces provided by the component to its environment, $RequiredInt$ is a set of interfaces required by the component from another component, or from the platform, $Properties$ is a set of properties holding non-functional meta-data describing different component aspects, $Implementation$ is the implementation of the component provided interfaces and its internal code.

The $Implementation$ part shown in Definition 1.5 indicates that a component is *executable*, either directly on the machine, or through interpretation, or binary translation, etc. Unlike other executable units, the code of a component *encapsulates* its responsibilities and business logic, and it is limited to interacting with other components and with the platform. A component also includes *meta-data* describing itself, e.g., its identification, interfaces, properties, etc.

A *component model*, not only identifies what is a component from what is not one, but also describes the process of creating individual components and the rules to connect them to form a complete system. Figure 1.3 illustrates such a system, where two components are deployed on the platform. When a set of components also conforms to the component model, it is called a *composite*.

Relative to a component model, a component can be defined as follows:

Definition 1.6 (Software component, defined by Heineman and Councill [56])**.** "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

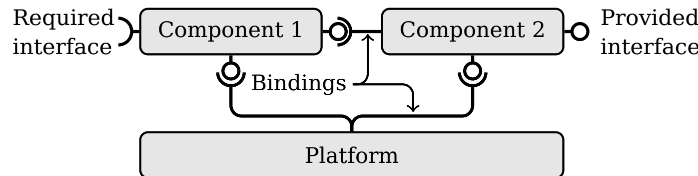


Figure 1.3: A component-based system.

A component can *provide* an interface I^P (i.e., an element of $\mathit{ProvidedInt}$ in Definition 1.5), by (1) implementing operations of I^P, and (2) exposing I^P as a *provided interface*, as represented by a –O in Figure 1.3. A component can also *require* an interface I^R (i.e., an element of $\mathit{RequiredInt}$ in Definition 1.5), by declaring I^R as a *required interface*, as represented by a –(in Figure 1.3. In order to be deployed on the platform, a component typically needs to provide a set of *standard interfaces* required by the platform.

A *binding*, a.k.a., a *wiring*, is a one-way channel of communication between a required interface and a provided interface, enabling invocation of operations defined in the provided interface. Bindings enable *composition* of components to build complete component-based systems. In order for a binding to be established from a required interface I^R to a provided interface I^P, I^R must be a subset of I^P, i.e., all operations defined in I^R must be defined in I^P. In object-oriented paradigm, this is formalized as: $I^P = I^R$ or I^P inherits from I^R. In order for a component to invoke an operation defined in an interface, it needs to (1) create a binding from the interface it requires to the interface provided by a component, and (2) invoke the operation via that binding.

4.2. The Fractal Component Model

Fractal [28, 29, 68] is a *hierarchical* and *reflexive* component model to design, implement, deploy and manage software systems. It is modular, extensible and programming-language-agnostic. This model can accurately describe, not only low-level resource access mechanisms, e.g., memory management or CPU schedulers, but also complex high-level applications, e.g., Web servers, data base applications, etc. Fractal has several implementations in various programming languages. Common implementations include Cecilia [31], THINK [3] and MIND [92] written in `C` and Julia [33] written in Java.

4.2.1. Component Model Description

In Fractal, an operation-based interface (see Definition 1.4) is represented by an *interface* that follows an *Interface Definition Language* (IDL) defined by Fractal. Operations are

represented by method signatures. A required interface is called a *client interface*, and a provided interface is called a *server interface*.

The simplest Fractal component (see Definition 1.5) is called a *primitive component*, and it is both a design-time and a runtime entity, acting as unit of encapsulation, composition and configuration. A set of components that conforms to the component model is called a *composite component*. By definition, a composite component can hold a set of primitive and composite components. Component properties are stored into *attributes*.

A component provides server interfaces as access points to the services it implements, and expresses its functional dependencies through client interfaces which describe the services it needs in order to operate. Components interact via *bindings* between client and server interfaces. Fractal also defines "controllers" which are optional *standard interfaces* that allow discovering and changing the structure and bindings of components at runtime. By implementing a subset of these standard interfaces, component developers can adjust the level of visibility of the component structure, and the level of control provided to the platform.

4.2.2. The MIND Component Model – a Fractal Embedded Implementation

MIND [92] is a native framework enabling composition of `C` components with configurable reflexive properties, targeted at embedded systems [58]. It is an open source implementation of the Fractal component model, and it is derived from THINK [3] and Cecilia [31]. Our first contribution called *Jasmin* (see Chapter 2) builds on the MIND framework, and enhances its structure, not only by making it service-oriented, but also by providing isolation support for components, and enabling transparent communication between isolated components.

The MIND framework is a development tool-chain and a set of reusable component libraries. MIND provides its own compiler `mindc` (see Figure 1.4) accepting `C` source files, architectural descriptions of components written in Fractal ADL [32] and interface description files. MIND compiler produces a set of `C` files[2] holding the glue to reify the component architecture at runtime. Finally, a `C` compiler and an object linker produce the executable binaries for the target platform. MIND provides an Operating System Abstraction Layer (OSAL), i.e., a set of components that provide generic operating system services, which eases porting of components to different operating systems.

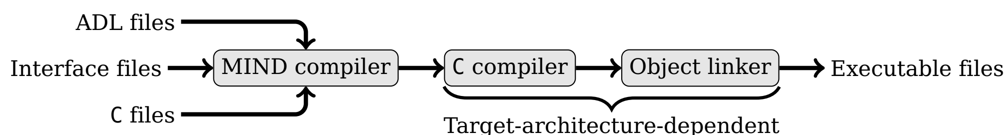


Figure 1.4: MIND compilation flow.

MIND also provides a documentation generator called `mindoc` which parses the different input files looking for special comments to create HTML files describing the parsed components. The MIND visual editor is an Eclipse IDE plug-in called `mindEd`. It enables designing components using a Graphical User Interface (GUI) that generates the components code automatically.

[2] `mindc` produces `C89` code to cover most `C` compilers.

MIND defines *annotations* to control several aspects of the compilation flow and to define non-functional properties of the components. Some annotations are used to add specified Fractal controllers to components in order to enable discovering and/or changing their structure at runtime. Other annotations provide support for reuse and integration of legacy code, including `C` libraries, through the automatic generation of interface and glue code exposing the legacy code to other components. Another set of annotations are used to specify command line options to the `C` compiler and the object linker for a specific component, for example, to link the component to a particular library. Finally, the MIND compiler provides *extension points* that enable configurable actions to happen at specific times during compilation. This enables several actions such as controlling the component architecture at compilation time, and applying specific architectural patterns, and adding new annotations to `mindc`.

4.3. The OSGi Component Model

OSGi [30, 127, 26, 82, 54] is a component model that enables building component-based Java applications, and running them side-by-side on the same Java Virtual Machine. The OSGi framework is a platform that provides various services to running applications, and enables installing, updating and uninstalling individual applications without restarting the platform.

4.3.1. Component Model Description

In the OSGi component model, an operation-based interface (see Definition 1.4) is represented by a Java package, where operations are Java methods defined by the classes belonging to the package. A required interface is called an *imported package*, and a provided interface is called an *exported package*.

An OSGi component is a Java object, called a *bundle*, and identified with a positive integer bundle ID that is unique within the running platform instance. A bundle is loaded from an *archive* using a dedicated Java class loader (see Section 2.2.1) that confines the bundle implementation into a separate Java name space. The archive holds meta-data describing the bundle, e.g., exported packages, imported packages, name, vendor, version, implementation entry point, etc. The implementation of a bundle is commonly provided as Java classes (see Section 2.2) included in the archive, but it can also be platform-specific machine code loaded by the Java Virtual Machine, i.e., Java native libraries. An OSGi *application* is merely a set of bundles, and there is no notion of composite in this component model.

In order for a bundle B_1 to invoke a method M_2 defined in a class C_2 belonging to a package P_2 provided by a bundle B_2, three conditions are necessary: (1) B_2 exports P_2, and (2) B_1 imports P_2, and (3) if M_2 is a `static` Java method, then B_1 calls M_2 via a Java reference to C_2, otherwise, B_1 calls M_2 via a Java reference to an object of C_2. If C_2 is rather an interface, then B_1 can only call M_2 via a Java reference to an object implementing the interface C_2. In fact, OSGi bindings are represented by Java object references.

4.3.2. Common Details of OSGi Implementations

The OSGi framework is based on Java and it is built to run multiple OSGi bundles on one Java virtual machine. This gives OSGi the ability to provide fast cross-bundle communication, and Java code isolation, and hot-swapping support.

Fast Cross-Bundle Communication. All OSGi bundles run in the same JVM inside the same memory address space. Consequently, communication between bundles happens via local Java method invocation (local procedure calls), making cross-bundle communication as fast as internal bundle communication. OSGi avoids the expensive cost of call serialization performed by common remote procedure call technologies, e.g., RPC [118], RMI [133], Incommunicado [102].

Code Isolation. At any given time, each bundle is associated to one dedicated Java class loader that loads the classes contained inside the archive of the bundle. Based on its class loader, each bundle can define its own security policy for its classes, which defines their access to system methods and to classes of other bundles. OSGi controls the accessibility of classes loaded by the class loader of each bundle. A bundle can choose to *export* some of its classes, i.e., to make them accessible to other bundles. Classes that are not exported cannot be accessed from outside the bundle. This control of accessibility is possible due to the process needed to access a class in Java, which goes as follows: In order for a class C_1 loaded by a class loader L_1 to access a class C_2 loaded by a class loader L_2, C_1 must access L_2 first, asking it to resolve C_2. L_2 can forbid access to C_2 in order to isolate C_2 from C_1.

In Java, two classes with the same name, loaded from the same class file using different class loaders are considered different types, as described in Section 2.2.1. Based on this fact, OSGi can load classes inside bundles with no risks of name conflicts, as each class loader defines a *private name space*, which enable each bundle to define its own version of the classes. This is particularly useful in OSGi, because it allows loading two versions of the same bundle simultaneously This allows, for example, to upgrade a bundle and while keeping previous versions loaded in order to preserve backward compatibility.

Hot-swapping. OSGi implements hot-swapping to load, unload and update archives on the fly. Archive loading and unloading is based on Java class loaders, which implement lazy loading, i.e., classes are loaded when they are accessed the first time. In order to update a bundle B_i, the following procedure is performed (see Figure 1.5):

1. OSGi creates a new class loader which loads the new version of the archive.

2. OSGi associates B_i with the new class loader, and it removes the association with the previous class loader.

3. OSGi broadcasts a message to all bundles indicating that B_i was updated. Bundles receiving this message should remove their references to B_i. Once all references to

the previous version of the bundle are removed, the previous class loader becomes unreachable (see Section 2.2.2).

4. The garbage collector should remove the previous version of the bundle from memory.

The updated bundle keeps using the same bundle ID as the previous version.

Each archive A_v is loaded using a class loader L_v. Initially, the bundle B_i is associated with the class loader $L_{1.0}$. Updating B_i from version 1.0 to version 2.0 implies creating $L_{2.0}$, then loading $A_{2.0}$ using $L_{2.0}$, then associating B_i with $L_{2.0}$ instead of $L_{1.0}$.

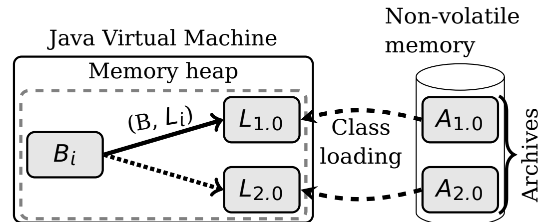


Figure 1.5: OSGi bundle update.

4.4. The OSGi Declarative Services Component Model

Declarative Services [126] is a component model that is based on the OSGi framework, for creating components that publish or use services [103]. Being declarative, publishing and consuming services do not require explicit Java code. Instead, it is specified as component meta-data and it is automatically handled by the Declarative Services subsystem in the OSGi framework.

In the Declarative Services component model, an operation-based interface (see Definition 1.4) is represented by a Java interface, where operations are Java methods defined by Java interface. A required interface is called a *used service*, and a provided interface is called a *provided service*.

A Declarative Services component is a Java object, called a *component*, and identified with a name. A component is defined by an OSGi bundle, and it is loaded from an OSGi archive by the Declarative Services subsystem in the OSGi framework. Therefore, the component lifetime is bounded by the lifetime of the bundle defining it. The archive holds meta-data describing the component, e.g., name, provided services, used services, implementation class, etc. The implementation of a component is provided as Java classes (see Section 2.2) included in the archive. There is no notion of composite, as a non-trivial set of components rarely conforms to the Declarative Services component model.

In order for a component C_1 to invoke a method M_2 defined in an interface I_2 provided by a component C_2, three conditions are necessary: (1) C_2 provides I_2, and (2) C_1 uses I_2, and (3) C_1 calls M_2 via a Java reference to an object O_2 created by C_2, and implementing I_2. In fact, bindings are represented by Java object references in the Declarative Services component model.

5. The Heterogeneity of the Smart Home

The smart home devices and applications provide services that are very different, not only functionally, i.e., features provided, but also non-functionally, i.e., quality of service, performances, etc. In order for applications developed by different service providers to inter-operate, standardized service contracts need to be defined. The rapid evolution of applications implies that they need to be upgradable individually without breaking dependent applications, which requires a low coupling between applications. The Service-Oriented Architecture (SOA) [103] fulfills these needs by enabling communication between applications via service contracts, called interfaces. This architecture keeps coupling between applications very low by dynamically binding applications providing services to applications depending on the services, allowing service consumers to "roam" between different service implementations provided by registered service providers. Furthermore, middlewares implementing the service-oriented architecture are often compatible with various component models, and with hot-swapping mechanisms.

This section describes the service-oriented architecture, and presents two examples of middlewares implementing it.

5.1. The Service-Oriented Architecture

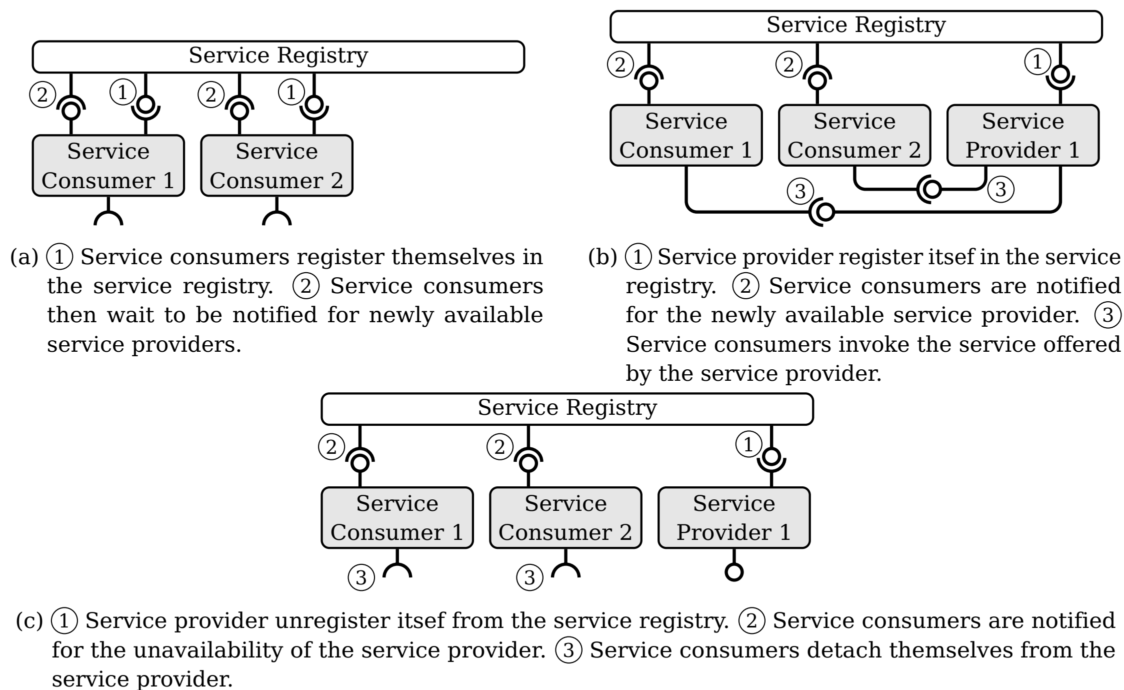
To begin, a *service* can be defined as follows:

Definition 1.7 (Service, defined by MacKenzie et al. [78])**.** "A **service** is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description." [78]

The service-oriented architecture is based on three parts: (1) a set of tiers providing a set of services, a.k.a., service *providers*, and (2) a set of tiers asking for a set of services, a.k.a., service *consumers*, and (3) a *service registry*. Services are formalized as *contracts* (a.k.a., *interfaces*) between *service consumers* and *service providers*. This architecture decouples the consumer of a service from its provider by introducing a *service registry* in between, as shown in Figure 1.6.

In a service-oriented architecture, every service consumer registers its dependency on a set of services it needs to invoke, as illustrated in Figure 1.6a. When a service provider starts providing a service it implements, it registers the service in the service registry, as shown in Figure 1.6b. The service registry then looks for service consumers that depend on the provided service, and when such service consumers are found, they are notified of the availability of the service, and they are provided with bindings that enable invoking the service.

When a service provider stops providing a service S_i, it notifies the service registry, which starts by looking for other service providers that implement the same service S_i. If such service providers are found, then the service registry notifies the consumers of the service S_i



(a) ① Service consumers register themselves in the service registry. ② Service consumers then wait to be notified for newly available service providers.

(b) ① Service provider register itsef in the service registry. ② Service consumers are notified for the newly available service provider. ③ Service consumers invoke the service offered by the service provider.

(c) ① Service provider unregister itsef from the service registry. ② Service consumers are notified for the unavailability of the service provider. ③ Service consumers detach themselves from the service provider.

Figure 1.6: Scenario of service discovery and service extinction in the Service-Oriented Architecture.

that they need to update their references to another service provider implementing S_i. If no other service providers provide S_i, then the service registry notifies the consumers of S_i that they need to reset their bindings to S_i and wait for another service provider. This procedure is briefly illustrated in Figure 1.6c.

5.1.1. The Service Registry

The service registry is basically an interactive service directory. Each service provider registers itself in the service registry to expose its provided services (see Figure 1.6b). Each service consumer asks the registry for services fulfilling certain functional and non-functional criteria. If the service registry finds a corresponding service, then it returns a service binding to allow the service consumer to communicate with the service provider and invoke the operations defined by the service.

When configured, the service registry notifies service consumers when new services are registered and available for use. Services can appear and disappear at any time in the service registry (see Figure 1.6c) unless otherwise specified via, for example, Quality of Service contracts.

5.1.2. The Whiteboard Design Pattern

Service providers often need to notify service consumer about *events* happening, in which case they behave as *event sources*. Event sources often implement the “Observer” [114, 48] design pattern in order to notify about events, by allowing listeners to *register* their *event handlers* which will be called when the events occur. For this, an event source would typically implement a private registry to store references to event handlers that will be called later. But, as the service-oriented platform already provides a service registry, that registry can be *reused* by event sources in order to simplify event notification. Kriens and Hargrave [72] call this reuse the “Whiteboard” pattern, in which event listeners do not need to track event sources and register themselves within the event sources. Instead, each event listener registers its event handling interface as a service in the service registry. When an event source needs to send an event notification, it invokes the event handling interfaces registered by all event listeners in the service registry.

5.1.3. The Publish-Subscribe Design Pattern

The service registry abstracts and simplifies service discovery. The whiteboard pattern also simplifies the implementation of event sources and event handlers by reusing the service registry. However, in order for an event source to invoke an event handler, the event handler must implement a specific interface provided, or agreed upon, by the event source. This coupling is what the publish-subscribe pattern aims to decouple, by defining a messaging protocol that is more flexible than coupled interfaces. This implies that an event source, called a *publisher*, does not need to be aware of the presence of even handlers, called *subscribers*, and does not impose when and how these subscribers receive event notifications.

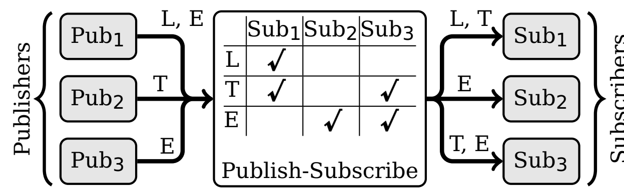
In the publish-subscribe pattern, messages are characterized by *topics*, a.k.a., *classes*[3]. The publish-subscribe pattern requires a message-oriented middleware that receives event notifications from publishers and notifies subscribers. A subscriber registers itself with a set of topics. When a publisher wants to notify about an event, it sends a message associated with a specific topic into the message-oriented middleware, which, in turn, notifies subscribers for that particular topic about the event. Notifications can be delivered to subscribers in various ways, and they can be delivered immediately or deferred.

The publish-subscribe design pattern in illustrated in Figure 1.7, where, for instance, Pub_1 sends events tagged with “Lighting” and “Energy” topics to the message-oriented middleware, which, in turn, routes lighting-tagged events to Sub_1 and energy-tagged events to both Sub_2 and Sub_3.

5.2. OSGi as a Service-Oriented Architecture

In addition to the component model defined by OSGi (see Section 4.3) and by OSGi Declarative Services (see Section 4.4), OSGi [30, 127, 26, 82, 54] can also be used as a service-oriented

[3] Not to be confused with object-oriented classes, e.g., Java classes.



Legend: L: Lighting, T: Temperature, E: Energy.

Figure 1.7: Publish-Subscribe Design Pattern.

platform to build and run component-based and service-oriented Java applications. OSGi not only provides a service registry that is compatible with the whiteboard design pattern, but also provides a message-oriented subsystem, called `EventAdmin`, that enables using the publish-subscribe design pattern.

OSGi. The OSGi framework implements a public service registry for use by all service-oriented OSGi applications running on the platform, via a service registry API. An OSGi bundle can register new services at runtime, or unregister services it previously registered. A bundle that consumes a service can track the registration of that service in order to be notified when the service is registered by some bundle. The service registry can be queried for services matching a specified filter involving its name, properties, etc.

OSGi Declarative Services. The OSGi Declarative Services describes a component model where services are first-class citizens that can be published and consumed merely using meta-data, and that have a defined life cycle, and that can be automatically configured. The lifetime of a service is still bounded by the lifetime of the bundle containing it. The service registry API is no more necessary for service registration or tracking, as that is performed automatically by the Declarative Services subsystem guided by the meta-data describing the services, stored in archives.

6. The Open and Embedded Aspects of the Smart Home

The openness of the smart home gateway implies hosting multiple applications delivered by multiple untrusted tiers and running simultaneously. As tiers are untrusted, some *isolation* needs to be achieved between applications developed by different tiers. The mechanisms needed to achieve this isolation depend on the nature of the applications to isolate. Furthermore, even when applications are isolated, they still need to be able to communicate easily and as fast as possible, in order to provide integrated services to the end-user. Moreover, the limited hardware resources shared between applications running on the embedded smart home gateway raise the risk of resource conflicts. Resource *monitoring* is a necessary mechanism to detect when these resource conflicts happen, as a prior step to resolving them.

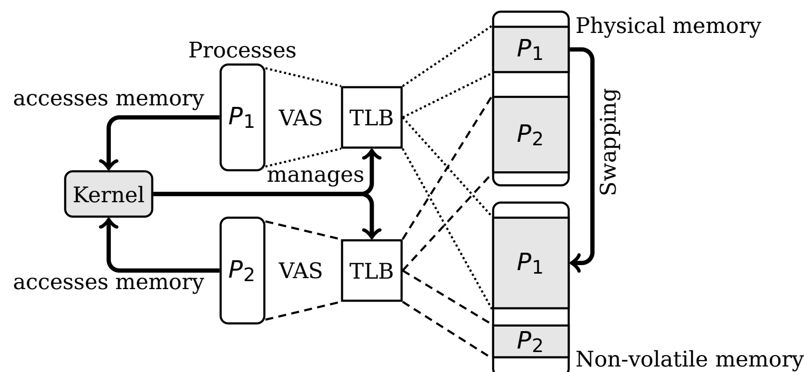
This section discusses the implications of the openness of the smart home environment on the design of the gateway, taking into consideration its hardware constraints. We discuss different isolation mechanisms based on hardware protection, virtualization, exokernels, and programming language safety. We also discuss mechanisms to monitor resource conflicts in different software environments. Then we illustrate a concrete resource conflict example: stale references.

6.1. Isolation based on Hardware Protection and Virtualization

An embedded gateway that needs to run native applications, for example due to hardware resource constraints, will need a mechanism to isolate the native applications. The mechanism needs to provide a good level of isolation, while being as lightweight as possible, and allowing easy and fast communication between isolated applications. This section presents common mechanisms used to isolate applications based on hardware protection and virtualization mechanisms.

6.1.1. Process Protection

A *process* [37, 94] is an operating system abstraction that runs an application in a *virtual memory* [42, 61, 49, 69] address space. A program running in a process accesses memory using virtual addresses which are only meaningful in the context of the process. A memory region can only be accessed if it was *allocated*. Memory allocation happens by (1) address reservation then (2) memory mapping. Because only the operating system kernel can map memory using Translation Look-aside Buffers (TLB), the kernel ensures that the physical memory blocks accessed by the processes do not overlap, which effectively isolates the memory of each process. Figure 1.8 illustrates this memory mapping.



Legend: TLB: Translation Look-aside Buffer. VAS: Virtual Address Space.

Figure 1.8: Virtual address translation in processes.

Processes also isolate Input/Output (I/O) to various kind of files, as every process has a separate file descriptor table. This enables, for example, to dedicate each unnamed file to the process that created it, e.g., unnamed pipes (FIFO), unnamed sockets.

6.1.2. Sandboxing

A sandbox is a virtual environment that severely limits the features provided to the sandboxed program by the operating system and the running programs and the hardware. The limited features given to the sandboxed program often translate to (1) a limited Application Programming Interface (API) the program can call, and (2) a limited set of machine instructions the program is allowed to execute. This effectively reduces the attack surface available to the program, and consequently increases the overall security of the system running the sandbox. Generally, a sandbox implements static and runtime security checks, and proxies the calls to its API toward the API provided by the underlying operating system, and runs the program code directly on the physical processors. This is illustrated in Figure 1.9.

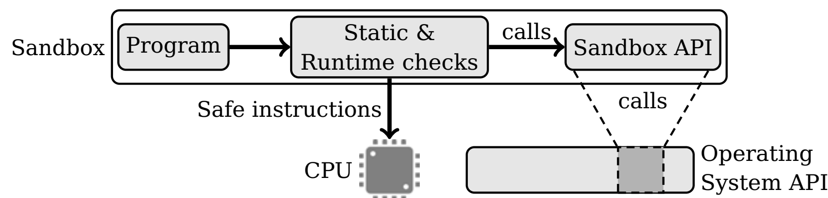


Figure 1.9: Sandbox structure.

The notion of sandboxes was introduced by Wahbe et al. [132] in the context of software-based *fault isolation*. Goldberg et al. [53] extended the sandbox notion to isolate untrusted helper applications, whereas the Native Client [135] defines a sandbox subsystem that enables running untrusted native code inside the Web browser.

6.1.3. Virtual Machines

A virtual machine [115, 111, 43, 94] is a binary translator from a source Instruction Set Architecture (ISA) to a target Instruction Set Architecture. The general definition of virtual machines allows for the source and/or target architectures to be virtual (V-ISA). However, this section discusses virtual machines where the source and target instruction set architectures are both real, i.e., implemented by real processors. Virtual machines often emulate the whole source hardware architecture, which makes them very accurate emulators, but often at a very prohibitive overhead. QEmu [18], VirtualBox [96] and VMWare Workstation [131] are common examples of virtual machines. *Paravirtualization* is a mechanism that significantly reduces the overhead of virtual machines by deferring some frequent virtualization tasks to a hardware chip that is driven by a *hypervisor*, i.e., a small software interface between the real hardware and the virtualized operating systems. Different forms of paravirtualization hardware chips are included in modern processors, e.g., AMD-V, Intel VT-x, Intel VT-d, VIA VT. Examples of software hypervisors include Xen [14] and BHyVe [124].

A virtual machine only emulates the source hardware, but it still requires the software compatible with that hardware to be installed in the virtual machine in order to make use of it. As all the hardware resources are emulated, virtual machines provide an excellent isolation level of almost all resource types.

6.1.4. Containers

A container is a group of processes that has a dedicated container context. The container context contains a process table, a user table, a network stack, a root file system, etc. This container context effectively isolates the group of processes from other processes running in the system outside the container. Therefore, a process inside the container only sees other processes running inside the container, and users defined inside the container, and network traffic happening inside the container, and files created inside the container or inherited from the outside file system. The Figure 1.3 shows the structure of Linux containers [77]. There are other container technologies, e.g., Solaris Zones [95], FreeBSD Jails [106], and OpenVZ [70, 121].

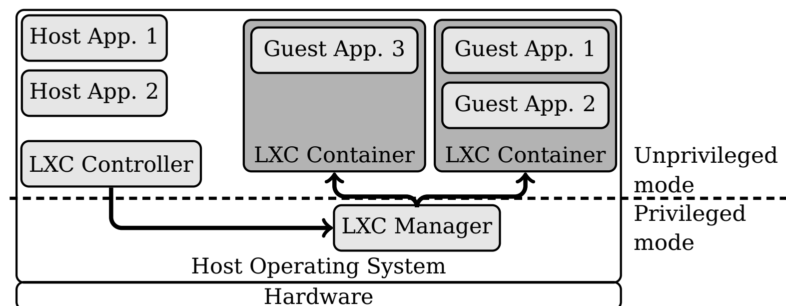


Figure 1.10: Linux Container (LXC) structure.

It is worth noting that processes running inside the container run on top of the same kernel as other processes, so there is no binary translation or emulation. This is why applications running inside containers run at full native speed.

6.1.5. Exokernels and Library Operating Systems

An *exokernel* [45, 2, 105, 16, 66] is an operating system kernel that separates resource *protection* from resource *management*. The *exokernel architecture* is motivated by the simple fact:

> "The lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions." [45]

The exokernel securely *multiplexes* and *exports* available hardware resources through a low-level interface to *untrusted library operating systems* which, in turn, manage those resources and implement higher-level abstractions such as Virtual Memory Management (VMM), Inter-Process Communication (IPC), etc. Applications developers select libraries or implement their own ones. This way, the exokernel OS architecture allows *application-level management* of physical resources. This enables *application-specific customizations* of traditional operating systems abstractions by *extending*, *specializing* and even *replacing* them, which promises a faster evolution of operating system features, while maintaining the system integrity. Examples of the exokernel architecture include the Aegis experimental exokernel [45], the

ExOS experimental library operating system [45], the Drawbridge Windows 7 library operating system [105], the Xok exokernel [66].

Resource Access Control. In order to secure access to physical resources, an exokernel uses three techniques: (1) *Secure binding*, i.e., applications securely *bind* to machine resources and handle *events*, and (2) *Visible resource revocation*, i.e., applications participate in a *resource revocation protocol* and know which resources are revoked and when, and (3) *Abort protocol*, i.e., the exokernel can *break*, by force, the secure bindings of uncooperative applications. In order to allows applications to tailor their resource allocation requests to available resources, the exokernel exposes *book-keeping data structures*, in either read-only or mutable form, such as disk arm positions, cached entries of the Translation Look-aside Buffer (TLB), etc.

Secure Bindings. A secure binding is a protection mechanism that decouples *authorization* from *actual use* of resources. It enables the exokernel to *protect resources without understanding them*, e.g., protect disk access without understanding the file system structure. A secure binding is a type of *capability*, i.e., an *unforgeable* reference to a resource. Secure bindings are based on (1) hardware mechanisms, e.g., processor privilege levels, hardware ownership tags, etc., and (2) software caching, e.g., software Translation Look-aside Buffer (TLB), and (3) downloading application code into the kernel. Downloading application code into the kernel enables *dynamic resource multiplexing* without hard-coding management knowledge into the exokernel. It also enables executing preliminary application logic without having to schedule the application process, therefore avoiding the kernel boundary crossing and process context switch.

Arbitration Policy. The exokernel includes a *policy* to arbitrate between competing library operating systems. The exokernel needs to determine which allocation requests to grant and from which applications to revoke resources. This is why it must determine the *absolute importance* of different applications, their share of resources, etc. Appropriate arbitration policies are determined by the environment rather than by the operating system architecture.

As management is deferred to application-level library operating systems, it is no more protected inside the kernel, so it can be altered in a buggy or malicious way. Even though the scope of these alterations remain local to the application, they can cause system-wide problems by altering stateful hardware resources accessed by other applications. This case is rarer in traditional systems as they take care in the Hardware Abstraction Layer (HAL) to manage the hardware resources correctly, and the HAL can only be altered by trusted applications. This threat becomes more important if the application is of *high priority*.

6.1.6. Comparison of Isolation Mechanisms based on Hardware Protection and Virtualization

In order to choose a mechanism for isolation based on hardware protection and virtualization, we need to consider at least two key aspects of each mechanism: (1) what is the performance overhead of using the mechanism, and (2) how much isolation is provided by the mechanism.

On the *performance* scale, processes are the winner, because (1) they use no extra disk space, and (2) they use very little extra memory and processing power, i.e., mainly for processor context switching. Given this fact, we take processes as a *baseline* when describing the overhead incurred by other mechanisms. Sandboxes hold the second position on the performance scale. A sandbox incurs the following overheads: (1) more memory and disk space to hold the sandbox management software and runtime, and (2) more processing to perform static and dynamic verification of code and data. Containers occupy the third position, even though they do not incur any additional processing overhead compared to processes[4]. The issue is that a container needs a separate set of user land programs, which requires more disk space, and a little more memory. This space waste can be avoided using *copy-on-write* mechanisms available in memory management subsystems and in recent file systems, e.g., ZFS [24], Btrfs [35]. Exokernels and library operating systems hold the fourth position, even though their performance competes with containers. Like containers, exokernels do not incur additional processing overhead compared to processes[4]. However, each application requires a separate library operating system to be loaded in memory, which implies a memory usage higher than containers but far lower than virtual machines [105]. Exokernels do not forcibly need a disk usage higher than processes. On one side, and unlike exokernels, containers run the hardware resource management logic in the kernel and share it across contained applications, which provides (1) higher state consistency guarantees, notably for stateful hardware, and (2) more resource-aware security guarantees, as the exokernel controls access only to hardware resources. Exokernels, on the other side, give more resource control to applications, and make the operating system more flexible, as library operating systems are easier to safely change than traditional operating system kernels. Finally, virtual Machines hold the last position on the performance scale, as they need more memory and disk space than all other mechanisms in order to hold the operating system and the applications to emulate. Virtual Machines also incur a processing overhead that is higher than sandboxes in most cases, even when paravirtualization is used.

We argue that the protection provided by processes is not enough to run multiple untrusted applications. The processing and memory and disk overhead incurred by virtual machine emulation is unaffordable in embedded systems. Sandboxes are less desirable than containers because they incur a high processing overhead, and they severely limit access to the operating system Application Programming Interface (API). Exokernels give an interesting promise to a flexible, yet secure, hardware resource management, but they still lack thorough *security* and *state consistency* analysis in the current state of the art [45, 2, 105, 16, 66]. Furthermore, exokernels require deep changes in operating system design, and they are not currently implemented in operating systems commonly used in the embedded industry, e.g., Linux, NetBSD, FreeBSD, Solaris, Windows, etc. Therefore, we argue that, currently, containers are the best compromise between these five mechanisms, in terms of isolation and performance, especially when copy-on-write optimizations are used to significantly reduce disk and memory

[4] Initialization time excluded.

usage. Still, containers do not provide easy and fast communication mechanisms, so such mechanisms need to be implemented in order to use containers in the smart home gateway.

6.2. Isolation based on Language Type Safety

Some programming language provide safety guarantees that ensure the following: (1) memory accessed by a program code will always be *bounded* to a specific region, and (2) memory blocks are *typed*, and access to a memory block of a specific type will never perform an operation that is unsupported by that type. These safety guarantees are enough to provide a process-like protection of concurrent applications, but without the need for hardware protection mechanisms such as the Translation Look-aside Buffers (TLB). This software-based protection combines static and dynamic verification by the compiler and the virtual machine, and it is used in many existing solutions to isolation, including the Singularity [60] operating system which is based on the safety of the Sing# programming language derived from C#, the SPIN [19] operating system which is based on the safety of the Modula-3 programming language, and the KaffeOS [12] platform which is based on the safety of the Java programming language.

6.2.1. A Typical Example: The Java Language and the Java Virtual Machine

In this section, we expose details of the Java language [55] and the Java Virtual Machine (JVM) [76], in order to support the descriptions of our contributions in the Chapters 3 and 4. Even though we used a JVM implementation based on VMKit [51] to prototype our contributions, the descriptions included in this section also apply to common Java virtual machines.

Type Safety of the Java language. Java is a programming language that is based on a *static* and *safe* type system. A type system is static when the checking of type related operations in a program can be performed using static verification of the program source code. A type-safe programming language does not allow operations or conversions that violate the type system rules. The Java type system forms the basic security guarantee of programs running on several platforms, including Android [113], KaffeOS [12] and MVM [40].

Structure of the JVM. The JVM is a High-Level Language Virtual Machine [115] that loads and executes Java byte code, a.k.a., Java classes (see Section 2.2). Based on Java byte code, the Java virtual machine offers binary portability across different architectures, and ensures safe code execution based on static and dynamic (i.e., runtime) verification. Many Java virtual machine implementations were developed, based on the public JVM specifications [76], including Hotspot JVM, Jikes Research Virtual Machine [1], Dalvik Virtual Machine [93, 25], VMKit [51]. A JVM holds many subsystems that interact heavily in order to execute Java byte code.

Primordial class loader: This subsystem [98] loads Java byte code, then it verifies its validity. The verification procedure is essential to guarantee the safety of the code. For example, accesses to arrays are checked not to exceed the array length. The Java byte code and meta-data are then fed to the machine code generator in order to produce executable code. The Java code can also define new class loaders that can be based on the primordial class loader. This allows Java applications to control how classes are loaded. It can also be used to ensure name space isolation for a group of classes by loading them using dedicated class loader.

Machine code generator: This subsystem is either an emulator, or a binary translator, or a mix of the two [115]. It transforms the internal structures describing the byte code into an executable machine code that is either run on the fly in the case of emulators, or run after generation and optimization passes in the case of binary translators. A binary translator is also called a Just-In-Time compiler (JIT compiler) as it is often a full compiler built into the JVM to compile Java byte code right before it is executed the first time.

Garbage collector: It is an automatic memory manager that takes care of terminating *unreachable* objects, and reclaiming their memory, and unloading byte code and machine code that is no more needed. The garbage collector can operate using different algorithms. Some of those algorithms require the JVM to pause executing all byte code and native code, in order to reclaim garbage objects. Garbage collectors using these algorithms are called *Stop-the-world* garbage collectors, and those that do not require this are called *concurrent* garbage collectors.

Objects finalization: This subsystem is responsible of terminating *unreachable finalizable* objects that the garbage collector chooses the reclaim. A finalizable object is an object whose implementation overloads the `java.lang.Object.finalize()` method. Termination happens by running the `finalize()` methods of the specified objects.

Synchronization: This subsystem is responsible of providing synchronization primitives to the Java byte code, e.g., monitors, locks, semaphores, etc. This subsystem can be implemented in the JVM [13] or directly in the underlying operating system.

Java runtime: The Java language is based on a set of standard classes providing support for various programming tasks, and called the Java runtime. The Java runtime supports the Java object model, Java reflection and persistence, common data structures and containers, etc.

6.2.2. Multi-Tenancy in the Java Virtual Machine

In order to isolate Java applications, the classic method is running each application in a dedicated Java virtual machine, which offers a good isolation level between untrusted applications. In an embedded gateway based on Java, the hardware constraints do not allow running multiple Java virtual machines. Therefore, one virtual machine needs to be shared between all the running applications, which raises the need for a *multi-tenant* [62, 11, 67] Java virtual machine, where the term “tenant” refers to an application vendor, i.e., a tier.

A multi-tenant Java virtual machine shares the Java runtime code and the virtual machine subsystems between all running Java applications. Figure 1.11 illustrates the structure of a multi-tenant Java-based gateway hosting multiple applications from multiple tenants.

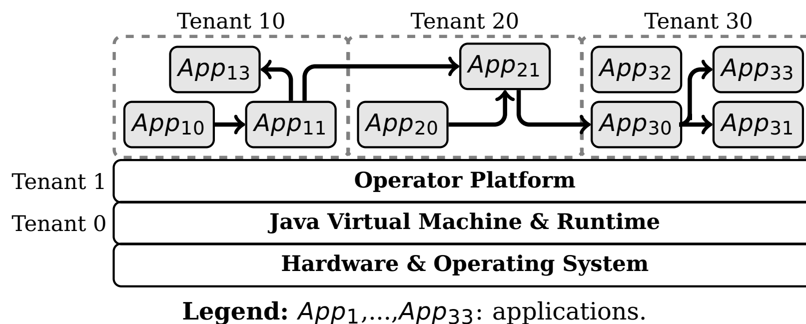


Figure 1.11: Multi-tenant Java-based execution environment

Partitioning the JVM. Majoul et al. [79] propose to partition the multi-tenant Java virtual machine into several isolated execution environments isolated from each another. Applications are grouped by trust level, and deployed in the partition associated with that trust level. Applications deployed in different partitions can only communicate using *communication channels* which are implemented in a protected isolation layer, which enforces security and safety policies on those channels. Kächele and Hauck [67] also argue that partitioning is necessary in the context of Cloud component-based applications, to enable better scalability. The need for easy and fast communication between different applications is the reason this mechanism is insufficient for our needs, especially given that neither Majoul et al. [79] nor Kächele and Hauck [67] discuss the complexity and the performance overhead of cross-application communications.

Consequently, we argue that multi-tenant virtual machines that use this kind of partitioning are unsuitable for the smart home gateway, and that tenants running in one JVM should be able to communicate directly, easily and rapidly.

6.3. Discovering Resource Sharing Conflicts via Monitoring

The smart home applications share the same gateway, thus they share the hardware resources available on the gateway, e.g., processor power, memory space, network bandwidth, etc. The gateway is a hardware constrained, long-running system that hosts untrusted applications. This highers the risk of resource sharing conflicts on the smart home gateway. This section describes resource sharing conflicts, and describes the mechanisms needed to discover them in various software environments.

6.3.1. Resource Sharing Conflicts

Most resources are supposed to be shared by multiple applications at the same time, and each application is supposed to use a “reasonable” amount of each resource. For resources that can be held by one application at a time, the holding application is supposed to hold on the resource for a “reasonable” amount of time, before releasing it. The term “reasonable” is a complex predicate that depends on many factors, including the nature of the resource, the nature of running applications, for how long the resource was used, the judgment of the resource management system (be it human or not), etc. For example, on a system running ten applications, it might seem unreasonable for one application to use 90% of the available physical memory for a long duration. An application that reads a configuration file normally should not lock the file for too long, otherwise a configuration agent might not be able to modify the file when needed.

A *resource sharing conflict* happens when an application uses the resource “unreasonably”, while another application requests the resource. For example, if an application fails to allocate memory because another application is using the majority of system memory since a long time, then a memory space sharing conflict is declared. When an application fails to perform a HTTP request for a long time because another application is using all the network bandwidth, a network bandwidth sharing conflict is declared.

Resource sharing conflicts are common in constrained systems running untrusted code for long periods of time. When resource sharing conflicts cannot be prevented, a resource management system needs to discover when they occur, as a first step toward resolving them. Discovering resource sharing conflicts raises the need for resource monitoring mechanisms. A resource monitoring mechanism does not judge if a resource sharing conflict happened. Rather, the data reported by the monitoring subsystem is used to help decide when a conflict happens, and what should be done about it. Therefore, it is required that the data reported by the monitoring subsystem be as *meaningful* as possible, that is, accurate, sufficiently precise, at the “right” granularity, etc.

6.3.2. Resource Monitoring in Different Environments

Resource monitoring is implemented in many existing operating systems and software environments. The Linux kernel provides Control Groups (CGroups) [84] which enable resource monitoring and reservation based on a hierarchy of processes. Linux CGroups allow process-based monitoring and reservation of memory space, processor usage, network bandwidth, etc. FreeBSD provides Performance Monitoring Counters (PMC) [71] that expose the hardware performance monitoring facilities to the running programs. FreeBSD PMC enable profiling of, not only kernel resources, e.g., kernel threads and hardware events, but also applications resources, e.g., processes, call graphs, threads, dynamically loaded code.

Monitoring resources in Java was investigated in many previous works. JRes [41] defines a framework for resource monitoring and resource limiting, based on runtime configuration of resources available to threads, and callbacks that run when resources are exceeded. JRes implementation is based on byte code rewriting and native code. VisualVM [100], Java Management Extensions (JMX) [52] and Java Virtual Machine Tool Interface (JVM-TI) [99]

are examples of industrial tools and frameworks providing means to monitor resources in Java applications.

The method presented by Miettinen et al. [86], and refined in A-OSGi [46] and also in our previous work on *adaptive monitoring* [83] address CPU usage monitoring of OSGi bundles. These solutions show that observing resource consumption at the bundle granularity can be performed without modifying bundles. The goal of adaptive monitoring was to provide support for the on-the-fly activation/deactivation of bundle (service) bindings monitoring without stopping bundles and losing states.

In Chapter 4, we present our third contribution: an OSGi-aware memory monitoring system. Our monitoring system is transparent to OSGi application developers, and it does not have any special isolation requirements. It reports monitoring results at tier granularity, and it accurately accounts for various interactions between tiers.

6.3.3. Direct and Indirect Accounting

When an actor A calls a service provided by another actor B, B consumes an amount of resources in order to provide the service. We say that A provides the *intent to execute the service*, and B provides the *logic to execute the service*. Therefore, the resources needed to provide the service are consumed both because (1) A asked for the service, and (2) because B provided the logic of fulfilling the service. Thus, resource consumption responsibility is actually *shared* between both actors. Resource accounting is about determining the distribution of the shared responsibility of resource consumption among the actors.

Definition 1.8 (Direct Resource Accounting)**.** Direct resource accounting is based on the following idea: the actor that controls the logic flow and directly consumes resources is *entirely* responsible of resource consumption. Simply put, the currently running method is accounted for the resources it consumes.

For example, the memory allocated by executing code provided by an actor is accounted to that actor.

Definition 1.9 (Indirect Resource Accounting)**.** Indirect resource accounting is based on another idea: the actor that asks for a service is *entirely* responsible for the resources consumed to fulfill that service. Simply put, the caller of the currently running method is accounted for the resources consumed by the method.

For example, the memory allocated by executing a service provided by an actor is accounted to the actor that asked for the service in the first place.

Most existing monitoring mechanisms are not *accurate* enough, mainly because they perform either direct accounting or indirect accounting, for all situations. To explain the inaccuracies, we use the following simple rule (Definition 1.10):

Definition 1.10 (Accurate resource accounting)**.** Resources used to provide a service are accounted to the bundle that requests the service.

Direct Accounting Issues. I-JVM [50], for example, accounts for resources inaccurately in many situations. I-JVM is a Java Virtual Machine that isolates OSGi bundles to a certain extent, and that enables only *direct* memory and CPU accounting, i.e., resources consumed during a method call are always accounted to the *called* bundle. This method produces inaccurate monitoring statistics when a bundle calls a service method provided by another bundle. Direct accounting implies that the called bundle providing the service is accounted for resources used to provide the service, which is the opposite of the Definition 1.10 for accurate accounting. In this situation, an accurate accounting should account the bundle calling the service method for resources used to provide the service.

Indirect Accounting Issues. Miettinen et al. [86] always perform *indirect* monitoring by attaching every thread to one bundle and accounting resources consumed by that thread to the attached bundle, no matter what code the thread actually executes. This method produces inaccurate monitoring statistics in some cases, such as bundles that manage a thread pool, and bundles that notify about events. A bundle that creates and manages threads in a pool will have all the threads attached to it, so resources consumed by the threads will be accounted to that bundle. This accounting is inaccurate, because a pool thread executes code only when requested by another bundle to do it. Therefore, it is the other bundle that requests the thread pool service. This is why an accurate accounting should account resources consumed by the pool thread to the bundle requesting the code execution. An event source bundle notifies about events when they occur, often by calling event handlers in a thread that it creates. In this situation, indirect accounting implies that resources consumed by event handlers will be accounted to the event source bundle. This accounting is inaccurate, because an event handler is called only because it was registered by another bundle for that particular event. Therefore, it is the other bundle that requests the event notification service. This is why an accurate accounting should account resources consumed by the event handler to the bundle implementing it.

6.3.4. Accounting for Resources Consumed during Cross-Application Interactions in OSGi

Authors of existing work about monitoring resources in OSGi [46, 86, 83] admitted that, during a service method call between two applications, correct resource accounting needs information related to *business logic* between the caller application and the service being called, which is neither provided by OSGi nor Java. This is why existing OSGi-related resource monitoring systems *avoid* the problem instead of solving it.

To avoid this problem, I-JVM runs each bundle in a dedicated isolate[5], composed of a separate class loader and a private copy of some Java objects, e.g., static variables, strings and `Java.lang.Class` objects. Isolates run in the same address space, and objects are passed by reference between isolates. Each isolate has a context[6] that enables access to its own private copy of objects. Each time a thread executes code from a given isolate, it sets its isolate context pointer to access the private objects of that isolate. This particularly happens when cross-bundle method calls are performed, or when methods return, or when exceptions are

[5] Not to be confused with Java Isolates defined in the JSR 121 [101]
[6] Called "Task class mirrors" in I-JVM.

thrown/caught. This isolate context switch is the reason only direct resource accounting is implemented in I-JVM. With only direct accounting possible, the authors of I-JVM avoided addressing the challenge of identifying which bundle should be accounted for the consumed resources when a cross-bundle call occurs.

Another approach to avoid this problem is *delegating* it to application developers. In particular, Makewave [80, 73] and ProSyst [107] follow this approach in their OSGi implementations, which was taken as the basis for the OSGi standards established by the OSGi Alliance [23]. The ProSyst solution to memory monitoring requires the framework developer and every bundle developer to call a standardized API method *before* performing a cross-bundle method call. The API method indicates to the monitoring subsystem which entity should be accounted for the resource consumption during the cross-bundle method call. Of course, access to this API is granted only to trusted entities, e.g., the OSGi framework and the bundles provided by the OSGi platform operator. Giving access to this API to untrusted bundles can cause wrong accounting of resources and cause false monitoring reports. This is why third party service providers sharing the platform cannot access this standard API. This approach is *too intrusive*, as it requires explicit effort from developers. Furthermore, this approach does not work in an environment where *untrusted* service providers share the platform and interact directly via local method calls.

6.4. Stale References – A Typical Source of Memory Conflicts

In this section, we first define stale references. Then, we discuss the reasons that make them a source of a memory conflicts in Java by illustrating their main consequences. Finally, we describe several approaches that try to resolve the problems caused by stale references.

6.4.1. Stale Reference Characterization

The lifetime of a Java object or class is defined differently depending on the perspective. Here, we discuss the Java language perspective, and a business logic perspective that defines additional states on objects and references.

From the Java point of view, an object starts living once it is created and constructed, i.e., after executing the `new` instruction. A class starts living once it is loaded and constructed, i.e., the first time it is accessed. Furthermore, an object dies when it is no more referenced, and a class dies when no code from that class is being executed and the class is no more referenced. A dead object is likely to be collected in the following garbage collection cycle.

From a business logic perspective, the lifetime could be different. For example, an object might need to be initialized via a call to an initialization method before it starts being useful. An object could also require explicit termination via a call to a termination method that makes it *unusable*, a.k.a., *stale*, consequently ending its lifetime. For example, an object that represents the contents of a file becomes usable after calling an `open()` method to open the file, and becomes stale after closing the file via a call to a `close()` method. Figure 1.12 illustrates the difference between the Java lifetime and the business logic lifetime.

Definition 1.11. From a business logic perspective, an object that was terminated is a *stale* object. Access to the stale object is considered an *illegal* operation. A reference to the stale object is called a *stale reference*.

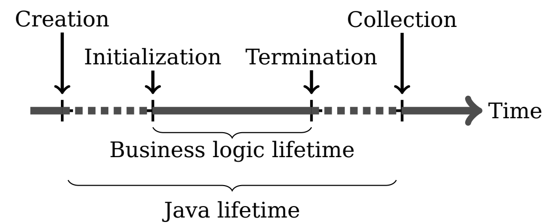


Figure 1.12: Java object lifetime from different perspectives.

6.4.2. Consequences of Stale References

In frameworks that run multiple applications in a single JVM (e.g., OSGi), each application is often associated to a class loader that loads its classes, and terminating an application often boils down to collecting its class loader. In such architecture, stale references cause mainly the two following problems:

- Because the object pointed by a stale reference is stale, making use of that object is considered an *illegal* operation from a business logic perspective, i.e., a bug, and such an access can result in *undefined behavior*, which compromises the application *state consistency*. For applications that control external devices, state inconsistency can induce unexpected behavior of controlled devices and even threaten the environment of the devices.

  Hot-swapping is often used to update applications by unloading the previous version and installing the new one. Code that updates some references to the newer version and keeps holding some stale references to the old version can result in undefined behavior if it feeds data or objects retrieved from the newer version of the application to a method of the older version, or vice versa. This can lead to state inconsistencies in either or both versions of the application.

- Due to Property 1.1, referencing a stale object forces the JVM to keep in memory: the object, and its class, and its class loader, and all the classes loaded by its class loader. Keeping all these elements in memory causes a significant *memory leak*, as is disallows the JVM from collecting the application loaded with the stale class loader, unless the entire JVM is shut down. This memory leak hinders the availability of long-running systems because it highers the risk of random allocation failures, a.k.a., out of memory exceptions, which often render the JVM unstable.

6.4.3. Stale References in OSGi

When a bundle is *uninstalled*, OSGi removes all references from the framework to the bundle's class loader, and broadcasts an event to other bundles asking them to release their references to that class loader (see Section 4.3.2). The class loader is subsequently considered *stale*. When a bundle is *updated*, OSGi first uninstalls it, making its class loader stale. OSGi then

creates a new class loader that loads the new bundle archive. Finally, OSGi updates the bundle information to make it refer to the new class loader, and broadcasts an event to other bundles asking them to update their references to point to the new version of the bundle.

A stale class loader should not be referenced anymore, in order for the garbage collector to end its lifetime and reclaim its memory. But, given the Property 1.2, in order to end the lifetime of a stale class loader, the garbage collector needs to end the lifetime of all the classes it loaded and all objects of these classes. Therefore, making a class loader stale is equivalent to making stale all the classes it loaded and all objects of those classes.

Definition 1.12 (Stale References in OSGi)**.** In OSGi, uninstalling a bundle B_i or updating it to a different version B_{i+1} makes B_i *stale*. This makes *stale* the class loader of B_i, and all classes it loaded, and all objects of those classes. Based on Definition 1.11, access to that class loader or those classes or objects is considered an *illegal* operation, and any reference to one of them is a *stale reference*.

Service Coroner [47] is a profiling tool that reveals OSGi stale references in testing and production environments. Experiments made by Gama and Donsez [47], the authors of Service Coroner, already showed that many stale references exist in several open source applications and frameworks, as illustrated in Table 1.1. It is worth mentioning that Service Coroner detects only stale *service* references. Because there are also many cases of stale references that go undetected by Service Coroner, we believe that the real amount of stale references is even higher than what is shown in the Table 1.1.

Application	Number of stale references detected
Newton 1.2.3 / Equinox 3.3.0	58
Jitsi alpha3 / Felix 1.0	19
JOnAS 5.0.1 / Felix 1.0	7
Sling 2.0 / Felix 1.0	3

Table 1.1: Stale references found by Service Coroner [47].

Weak References are Not a Solution. The Java specification defines a *weak reference* [119] as a Java reference that enables accessing an object without ensuring that the object will stay alive. If the garbage collector finds that an object is only accessible by weak references, then it can choose to set all the remaining weak references to `null` then collect the referenced object. Weak references are not adequate in the context of OSGi, because a cross-bundle reference needs to ensure that the referenced bundle stays alive, at least until the bundle is uninstalled or updated. That is, a cross-bundle reference needs to behave as a normal reference (a.k.a., strong reference) before the bundle is uninstalled or updated, and needs to behave as a weak reference after that. This is why using weak references do not solve the problem of stale references.

6.4.4. Attempts to Avoid Stale References

The two main causes of stale references are running applications in the same address space while providing the guarantee given by strong references, i.e., strongly referenced object will

not be reclaimed. Strong references are the default type of references in many programming languages with automatic memory management, such as Java. This is why frameworks based on those languages often choose to isolate applications in different address spaces in order to avoid the problem of stale references. The Multitasking Virtual Machine (MVM) [40], Android [113], Singularity [60] and KaffeOS [12] are examples of application frameworks where processes are fully isolated. Isolation is either achieved at the operating system level by using distinct address spaces or at the process level by using a single address space while preventing applications to directly share objects. In these frameworks, communication between applications involves remote procedure calls, which require data marshaling that add a significant overhead, especially compared to direct procedure calls inside a single address space.

Incommunicado [102] proposes an API enabling fast communication between isolated Java applications running on the Multitasking Virtual Machine. Even though it is ten times faster that regular Java RMI [133], Incommunicado is still hundreds times slower that direct method calls, as pointed out by Geoffray et al. [50]. Furthermore, in Incommunicado, cross-application references are *weak references*, which are not an acceptable solution in the context of OSGi, as previously indicated.

Stale References are Hard to Debug. Avoiding stale references is challenging for the OSGi application developer. From the Java point of view, a stale reference is just a Java reference like any other. Its only specificity is that it crosses bundle frontiers to refer to an object previously obtained from a bundle that has since been updated or uninstalled. To *manually* detect stale references, the developer must track all cross-bundle references in the source code and carefully check that these references are always kept consistent when bundles are updated or uninstalled. Since manual checking is difficult and error-prone, we argue that an automated approach is required. Because a bundle can be uninstalled or updated by a user, and this is not necessarily apparent in the application source code, the validity of cross-bundle references cannot be determined by static analysis. Thus, run-time analysis is required.

6.4.5. Detecting Stale References

Jump and McKinley [65] defines *unnecessary references* as “pointers to objects the program never uses again”. Our definition of stale references implies that they are also unnecessary references. The problem of detecting unnecessary references was investigated by several existing tools mostly based on heuristics and unsound program transformations. We describe here some tools used to detect or resolve stale references. The Chapter 3 presents our second contribution called *Incinerator*, which is a Java virtual machine subsystem that detects and eliminates stale references in OSGi-based applications, with a very low overhead.

Service Coroner. Service Coroner [47] is a profiling tool that reveals OSGi stale references in testing and production environments. It uses Aspect-Oriented Programming (AOP) techniques to hook up into the OSGi interface calls and traces objects passed between bundles using Java weak references. Service Coroner does not identify the bundles retaining

stale references, but it provides intermediary information to help the user identify the buggy bundles. Service Coroner can run on the same machine as the profiled JVM, or it can run in distributed mode, where it is controlled remotely. Service Coroner can be used in active mode, where the user requests repetitive deployment tasks to happen (`stop`, `update`,...). This use case is useful in test environments. Service Coroner can be used in passive mode, where the bundles are left running undisturbed and reports are generated on user demand. This replicates the real behavior of the bundles but needs more time to reveal stale references. This use case is useful in production environments where availability is crucial.

Cork. Cork [65] is an extension to the garbage collector that monitors the runtime evolution of the *volume* of references to Java classes, in order to discover situations where references to a particular class do not stop growing. The volume of references is the number of references multiplied by the referenced objects sizes. From Cork perspective, if the number of references to objects of a given class grows continuously, then it is highly likely that many of the references to that class are stale references. Interestingly, this heuristic incurs a low overhead to be calculated, and indicates many situations of stale references. Cork is made for long-running system because it needs to monitor the evolution of references during multiple garbage collection cycles. In some situations, references are collected sometimes, but not all the time. Those references actually cause a memory leak, but that leak might not grow indefinitely. These types of leaks are not detected by Cork.

LeakBot. LeakBot [87] does a good job at detecting probable stale references using a multitude of *structural* and *temporal* heuristics calculated on the object graph. LeakBot performs detection in two phases. In the first phase, LeakBot ranks candidate leaks, by (1) eliminating references which cannot be leaks, based on binary metrics, then (2) ordering the remaining references by likelihood of being leaks, then (3) applying non-linear gating functions to differentiate more clearly between ordered candidates, then (4) ranking objects based on the calculated ranks of objects related to them. The second phase aims to give a more complete picture, that is why LeakBot spots co-evolving regions based on similarity of evolution patterns of data structures. Then, each co-evolving region is ranked based on estimated number of leaks inside it. LeakBot goes beyond offline analysis of object graphs, by sampling the dynamic evolution of co-evolving regions, in order to update the rankings of these regions. Highly ranked objects are very probable leaks, and highly ranked regions are likely to contain multiple memory leaks. LeakBot is mainly targeted at server-scale long-running applications.

Cyclic Memory Allocation. Nguyen and Rinard [90] finds and eliminates certain memory leaks by looking for *m-bounded* allocation sites in the program, then using *cyclic memory allocation* at those sites. The tool requires running the program on a set of manually crafted inputs. The executed program is instrumented to look for m-bounded allocation sites, i.e., allocation sites that verify the following: “at any time during the execution of the program, the program accesses at most only the last m objects allocated at that site”. For each of those sites, if the difference between the number of allocations and deallocations exceeds m, then there is a memory leak at the site. This is why the tool replaces the allocation mechanism at each m-bounded allocation site by a cyclic memory allocation mechanism, which can be

resumed in four points. (1) The first allocations starts by preallocating a buffer holding *m* entries suitable for the type allocated at the allocation site, then returns the first entry in the buffer. (2) Each subsequent allocation merely returns the next entry in the buffer. (3) At the end of the buffer, the mechanism wraps around by returning the first entry in the buffer, again. (4) Deallocation is a *no-op*. This technique is unsound because it relies on accurate detection of *m*, which is performed empirically. Wrong estimation of *m* causes higher memory usage if estimated *m* is greater than the real value, and would cause *undefined behavior* if estimated *m* is less than the real value. Furthermore, given the statistics performed by Nguyen and Rinard [90], half the allocation sites are *m*-bounded, so the rest of allocation sites is unsupported by this approach.

Melt. Melt [22] is a memory leak tolerance mechanism for Java. It is a fine-grained memory paging mechanism working at object granularity instead of *page* granularity. Melt defines stale objects as objects that were not accessed since the last garbage collection cycle. When it spots a stale object, Melt removes the object from physical memory and stores it on disk, in order to allow the program to run longer and to give developers more data to fix the leaks. When a swapped object is used again, Melt reloads it from disk and makes it usable again without semantic changes.

7. Conclusion

This chapter explores the challenges of the smart home, driven by its particular properties: open, dynamic, in rapid development, heterogeneous, distributed and embedded. Each property comes with a set of challenges, and the composition of these properties also requires some trade-offs to be made, and custom solutions to be studied.

The dynamic aspect of the smart home implies the need for *hot-swapping* of applications, i.e., loading, starting, stopping, updating, unloading applications at runtime without restarting the software platform. We investigated common mechanisms to hot-swap, not only native code [63, 85, 128], but also Java byte code [76]. Then, we described the delicate conditions necessary to unload a Java application loaded with a dedicated Java class loader, which is a common technique in *multi-tenant* Java virtual machines. This unveiled the risk implied by Java *stale references* on resource-constrained long-running systems, such as the smart home gateway. The Chapter 3 presents our proposed solution to detect and eliminate Java stale references with a very low overhead.

Because it is hard to share memory in a distributed environment, we briefly described how applications can interact, both when they can share memory, and when they cannot. We therefore described *local* procedure calls [7], and *remote* procedure calls [133, 75, 102], involving expensive serialization routines. The drop in communication performance induced by remote procedure calls is one of the reasons that encourage running multiple applications in the same memory addressing space whenever possible, especially when those applications communicate frequently.

In order to rapidly develop applications that take advantage of the wealth of devices, sensors and actuators available at the smart home, we proposed to develop applications based on reusable components, which led us to describe a generic *component model*. Then, we illustrated examples of component models [38, 122] and component model implementations that are suitable for constrained embedded environments and compatible with other properties of the smart home, e.g., Fractal[7] [29, 68, 28], MIND[8] [92], OSGi [7,8] [30, 127, 26, 82, 54], OSGi Declarative Services[7,8].

To cope with the heterogeneity of the smart home, we suggested the use of the *service-oriented* architecture [103] to design the smart home platform and applications. For this purpose, we first described the key elements of the service-oriented architecture, with some commonly used design patterns [114, 48, 72]. Then, we presented examples of service-oriented middleware implementations, e.g., OSGi, OSGi Declarative Services.

The last, and most problematic properties of the smart home are open and embedded aspects. In a gateway running numerous applications delivered by multiple untrusted service providers, *isolation* becomes necessary to avoid certain security threats and to enable a basic level of robustness against misbehaving applications. Therefore, we discuss five mechanisms of isolation based on hardware protection and virtualization approaches, i.e., processes [37, 94], sandboxes [132, 53, 135], containers [77, 95, 106, 70, 121], virtual machines [115, 111, 43, 94], exokernels and library operating systems [45, 2, 105, 16, 66]. We compare those based on their performance overhead versus the isolation they provide, and we argue that container technologies offer the best bet. Chapter 2 describes Jasmin: a middleware for development, deployment, isolation and administration of component-based and service-oriented applications targeted at embedded systems. Jasmin is based on the MIND implementation of the Fractal component model. It provides isolation of applications based on the Linux containers technology, and it extends the service-oriented architecture to isolated applications, and it enables transparent and fast communication between isolated applications. Afterwards, we invoke isolation mechanisms based on language type safety, e.g., Singularity [60], SPIN [19], KaffeOS [12]. We illustrate the Java language as an example of a *static* and *safe* language, and we briefly describe the major parts of the Java Virtual Machine. We also discuss techniques to run multiple isolated applications in a single Java virtual machine, and we argue that partitioning applications [79] is unsuitable for smart home applications that need to communicate easily and rapidly.

Most isolation approaches are not perfect, leaving room for resource sharing conflicts to happen. In our quest to solve these conflicts, we need tools to discover them in the first place. In other words, we need to monitor these sharing conflicts. Thus, we begin by discussing the notion of "resource conflict", then we expose different monitoring approaches in different software environments [84, 71, 41, 100, 52, 99, 86, 46, 83], and we describe fundamental complementary points of view in resource accounting. Finally, we detail a typical source of memory conflicts in multi-tenant Java virtual machines, discussing its major implications and examples of state-of-the-art efforts [47, 65, 87, 22, 90] to tackle it. The contribution presented in Chapter 4 proposes a memory monitoring mechanism that is aware of the component-based design of the smart home platform, and which provides monitoring data that is accurate and relevant to that design.

[7] A component model.
[8] An implementation of a component model.

Chapter 2.

Jasmin – Isolating Native Component-based Applications

Contents

1 Jasmin Component-Based and Service-Oriented Applications **40**
2 The Standalone Jasmin Architecture **41**
3 The Distributed Jasmin Architecture **42**
3.1 Architectural Separation of Jasmin Applications 42
3.2 Jasmin Distributed Service-Oriented Architecture 43
4 Multi-Level Isolation of Jasmin Applications **43**
4.1 Zero Isolation 44
4.2 Process-based Isolation 44
4.3 Container-based Isolation 44
5 Transparent and Fast Communication between Jasmin Applications **45**
6 Evaluation **46**
6.1 Communication Speed Benchmarks 46
6.2 Disk Footprint 47
6.3 Memory Footprint 48
6.4 Performance and Porting Efforts of a Legacy Multimedia Application 48
6.5 Summary of Results 51
7 Conclusion **51**

Jasmin [4] is a middleware for development, deployment, isolation and administration of *component-based* and *service-oriented* applications targeted at *embedded* systems. Applications running on Jasmin are built using the MIND tool-chain and components (see Section 4.2.2). Jasmin inherits the Fractal component model implemented by MIND, and extends it by adding a distributed *service registry* to provide a service-oriented [103] platform. Consequently, Jasmin defines a *life cycle* for applications, and controls that life cycle either on-demand, or automatically when resolving dependencies. Jasmin supports *multiple isolation levels* for applications, ranging from no isolation to container-based isolation (see Section 6.1). This allows platform administrators to choose the suitable isolation level depending on their constraints. Jasmin enables *transparent communication* between isolated applications, by offering automatic proxy setup for application interfaces. It also provides an *Operating System Abstraction Layer* (OSAL) that makes applications portable across different architectures, and that eases porting of the Jasmin middleware itself. Finally, Jasmin provides a *scriptable*

administration shell to deploy and control applications at runtime, either manually or via scripts.

In this chapter, we present the design of Jasmin middleware, by first describing Jasmin application life cycle, then presenting the basic services provided by the middleware. Then we illustrate the distributed architecture variant of Jasmin, and how it allows isolating applications, which continue to communicate transparently thanks to Jasmin proxies. Afterwards, we evaluate Jasmin features and performance, based on micro-benchmarks and a typical smart home use case.

1. Jasmin Component-Based and Service-Oriented Applications

A *Jasmin application* is a MIND component additionally providing a set of non-functional interfaces required by Jasmin. Using those interfaces, Jasmin manages the life cycle of applications, and performs structural exploration (a.k.a., runtime reflection) and adjustments of the applications internal components. A Jasmin application may also provide some optional interfaces recognized by Jasmin, in order to enable richer levels of runtime reflection and deeper automatic resolution of components dependencies. We refer to Jasmin applications simply as “applications” throughout the rest of this chapter.

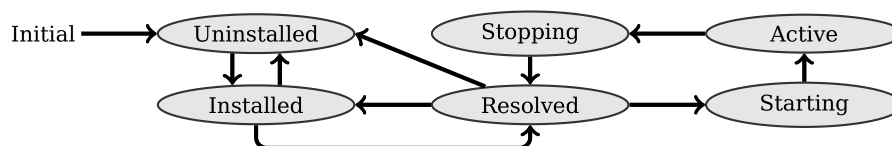


Figure 2.1: Jasmin application life cycle.

Jasmin considers applications stored in the repositories it manages. These applications follow the life cycle described in Figure 2.1 with the specified states:

Uninstalled: Stored in a repository but not loaded into the Jasmin execution environment.

Installed: Loaded into the execution environment, but some of the required interfaces might still be unbound.

Resolved: Installed, and all required interfaces are bound and resolved.

Starting: Resolved, and the application is currently initializing, possibly invoking other interfaces.

Active: The application was initialized successfully, it is currently providing its interfaces and possibly invoking other interfaces.

Stopping Application is currently terminating execution.

2. The Standalone Jasmin Architecture

This section presents the simplest architecture of Jasmin, the *standalone Jasmin architecture*, which can be entirely executed within the same memory addressing space, such as a device without a Memory Management Unit (MMU), or a single operating system process.

The simplest scenario addressed by Jasmin is grouping multiple Jasmin applications in a MIND composite called an *application container*. The application container is controlled by a *Standalone Jasmin execution environment*, which is a MIND composite holding components that provide a set of services to the applications. In the rest of this chapter, we refer to the Standalone Jasmin execution environment simply as the “standalone Jasmin”. Figure 2.2 illustrates this configuration.

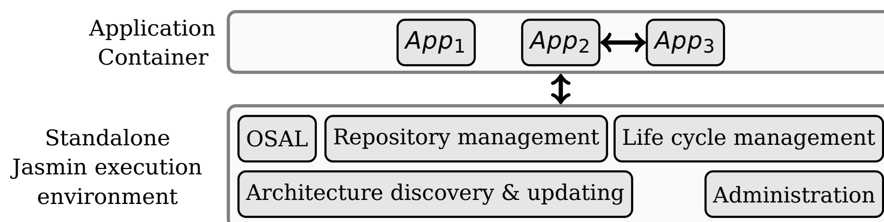


Figure 2.2: Standalone Jasmin architecture.

The standalone Jasmin applies the Service-Oriented Architecture (SOA) paradigm inside the application container, by tracking services provided and required by applications, and performing dependency resolution inside the application container. Additionally, the standalone Jasmin provides the following services to the applications:

Repository management: Jasmin handles multiple local or remote repositories, that can be sorted according to criteria such as provider or domain, and searches them for the application binaries it is requested to install.

Life cycle management: By defining a *service registry*, Jasmin handles application states, and performs transitions between states shown in Figure 2.1, and resolves service dependencies between applications. Jasmin detects circular dependencies and prevents circular bindings.

Architecture discovery and updating: Jasmin enables runtime discovery and exploration of the component-based architectures of loaded applications. It also enables updating the architecture by adding or removing components or adjusting bindings.

Operating System Abstraction Layer: Jasmin provides an Operating System Abstraction Layer (OSAL) to help make applications portable to other architectures. The OSAL also eases porting of the Jasmin execution environment.

Administration: This service monitors and controls all the aspects of the Jasmin execution environment and exposes the state of loaded applications to administration agents, e.g., command-based consoles, script files engines, autonomous management systems.

3. The Distributed Jasmin Architecture

The process of isolating applications begins by separating them into groups, and running each group in an application container controlled by a dedicated standalone Jasmin. This simply means running multiple independent standalone Jasmin execution environments. The limits of this approach appear when the number of standalone Jasmin execution environments becomes important. First, platform administration becomes more complex, as each standalone Jasmin architecture has to be managed separately. Second, communication between applications deployed in different application containers becomes more difficult as the SOA paradigm and dependency resolution happen only within one application container.

3.1. Architectural Separation of Jasmin Applications

To overcome the problems invoked above, we created the *Root Jasmin execution environment* which is a MIND component responsible of managing several Jasmin execution environments. In the rest of this chapter, we refer to the Root Jasmin execution environment simply as the "root Jasmin". The root Jasmin performs remote administration of multiple Jasmin execution environments, and remote service discovery, and remote dependency resolution, therefore effectively extending the SOA paradigm to multiple application containers.

We also redesigned the standalone Jasmin architecture to cope with this new distributed design, therefore we replaced the standalone Jasmin by a *Local Jasmin execution environment*, which, not only provides the features of the standalone Jasmin, but also allows administration by the root Jasmin, and participates in remote dependency resolution. In the rest of this chapter, we refer to the Local Jasmin execution environment simply as the "local Jasmin". For a given local Jasmin, *local applications* are applications running inside the application container it controls. *Remote applications* are those running inside application containers not controlled by the local Jasmin itself. *Local services* are services provided by local applications, and *remote services* are provided by remote applications. The resulting architecture is illustrated in Figure 2.3.

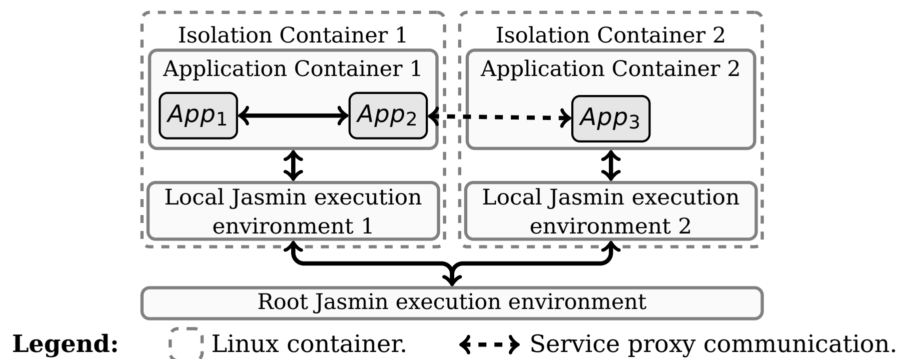


Figure 2.3: Distributed Jasmin architecture.

3.2. Jasmin Distributed Service-Oriented Architecture

This distributed aspect makes communication between remote applications more difficult. Therefore, we supported communication between applications running in different application containers using the proxy pattern to enable invoking interfaces implemented by applications, independently of the applications locations (see Section 5).

In order to allow seamless service-oriented interaction between separated applications, we created a *service registry coordinator* in the root Jasmin. The purpose of the service registry coordinator is to enable applications to consume services easily without having to worry about the location of the service provider, be it local or remote. Therefore, the service registry coordinator can be seen as a global service registry that virtually aggregates all the service registries in the local Jasmin environments managed by the root Jasmin. Using this coordinator, service dependency resolution can be performed, not only locally, but also remotely, transparently to the applications providing and requiring the services. The dependency resolution protocol can be described with the help of Scenarios 2.1 and 2.2.

Scenario 2.1 (Jasmin Local Dependency Resolution)**.** When an application A provides a service, the local Jasmin running A registers the service in its service registry. If a local application B requires the service provided by A, then the local Jasmin finds the service in its service registry, and it *binds* the applications A and B, allowing B to invoke the service. This protocol occurs entirely within the local Jasmin.

Scenario 2.2 (Jasmin Remote Dependency Resolution)**.** When an application A provides a service S, the local Jasmin L_A running A registers the service in its service registry. If a remote application B running in a local Jasmin L_B requires the service S, and the local Jasmin L_B cannot find S in its service registry, then L_B asks the root Jasmin to find the service S. The root Jasmin, consequently, asks all the local Jasmin environments it manages (except L_B) to look for the service S. The local Jasmin L_A receives a remote request for S from the root Jasmin, in which case it creates a server proxy P_S for the service S in the application A, and it gives back the proxy reference to the root Jasmin. The root Jasmin, in turn, routes the proxy reference to L_B. L_B, then, creates a client proxy P_B that has the same interface as S, and which communicates with P_S, then L_B binds B to P_B. Now, invocations of S's methods are automatically routed from B to P_B, then to P_S which invokes A. The binding $B \rightarrow P_B \rightarrow P_S \rightarrow A$ is known as a *complex binding* in the Fractal component model.

4. Multi-Level Isolation of Jasmin Applications

The smart home presents several risks due to its openness to different untrusted service providers. The standalone Jasmin architecture (see Figure 2.2) is vulnerable to multiple security issues, because applications running in the application container are all exposed to each other. An application can, for example, monitor another application, or steal information from it, or corrupt its memory, etc. Therefore, application resources, including CPU, memory, file system, and network, need to be isolated to provide a robust execution environment. The distributed Jasmin architecture (see Figure 2.3) already eases this task by separating application containers while allowing unified administration of local Jasmin

execution environments and distributed Service-Oriented Architecture (SOA) paradigm and transparent communication.

Isolating applications goes beyond separating them, and it typically involves making a compromise between safety and performance. Jasmin implements multiple levels of isolation, in order to allow platform administrators to adapt this compromise depending on the situation at hand. The different isolation levels implemented by Jasmin are described next.

4.1. Zero Isolation

In this level, Jasmin applications are merely separated because they are contained in different MIND components, i.e., in different design units. Remote service invocation is implemented as local method calls. This level enforces no means of isolation, and executes all applications within one address space, e.g., a device without a Memory Management Unit (MMU), or a single operating system process.

This level is suitable for applications that communicate frequently, because it allows local method calls between applications. The targeted environment is either too constrained, or very controlled, e.g., development and testing environments.

4.2. Process-based Isolation

In this level, the root Jasmin runs in a dedicated process. Each local Jasmin run in a dedicated process with its controlled application container, which isolates the address space of applications belonging to the application container. Remote service invocation is implemented via Jasmin proxies (see Section 5).

This level is suitable for concurrent applications which seldom communicate.

4.3. Container-based Isolation

In this level, Jasmin runs each local Jasmin with its controlled application container in a dedicated process which is executed in a dedicated Linux container. The root Jasmin runs in a dedicated process directly on the host, i.e., outside Linux containers. Remote service invocation is implemented via Jasmin proxies (see Section 5).

In a nutshell, a Linux container is an isolated group of processes, having a dedicated process table, and user table, and device table, and file system, and network stack. Therefore, a process inside a container can neither see (1) processes running outside the container, nor (2) users defined outside it, nor (3) network transmissions happening outside it, nor (4) files stored outside its dedicated file system, nor (5) devices not inherited from the host system.

This isolation level is suitable for untrusted applications running on a kernel that provides container isolation support, such as the Linux kernel 2.6.29 and above.

5. Transparent and Fast Communication between Jasmin Applications

Running application containers inside processes or Linux containers makes communication between isolated applications more complicated, because local method calls are not possible between different address spaces. This is why we created *Jasmin proxies*. A Jasmin proxy is a MIND component that *transparently* performs *data marshaling* and *transmission*, needed to call an interface method of a remote service. Jasmin automatically loads and binds proxies to make invocation of service interfaces independent of where the provider application is actually deployed. As described in Scenarios 2.1 and 2.2, a Jasmin proxy is loaded by the local Jasmin, either to expose a local service interface (server-side proxy), or to consume a remote service interface (client-side proxy). The proxy is loaded on requests from the root Jasmin when performing remote dependency resolution, i.e., when an application requires a service provided by a remote isolated application.

Transparent Data Marshaling. In order to make remote service invocation transparent to developers, Jasmin takes advantage of the versatile *binding* notion of the Fractal component model (see Section 4.2) by replacing a simple MIND binding, i.e., a simple function reference, by a complex binding to a Jasmin proxy which serializes the parameters of the method to invoke, then deserializes the results when the method finishes, as described in Section 3. What is particular about Jasmin proxies is that they are *component-aware*, i.e., they take care of routing interface calls to specific components implementing the interface in the application providing the service. In other words, Jasmin proxies perform remote *double dispatching*, the first dispatching selects the component implementing the interface to call, and the second dispatching selects the method to call. Jasmin does not perform conversions of local data representations to/from the network data representation [120], because it is a useless effort, as the communicating applications run on the same operating system and the hardware architecture. A Jasmin proxy does not handle data transmission between applications. Rather, it uses another subsystem in the Jasmin middleware that handles local communication between Linux containers.

Fast Communication between Containers. Jasmin contains a subsystem that enables fast communication between Linux containers. The subsystem uses *local sockets* or *named pipes* Inter-Process Communication (IPC) mechanisms, unlike the Remote Procedure Call [118] (RPC) protocol on Linux which uses the Transport Control Protocol (TCP) over Internet Protocol version 4 (IPv4). This eliminates the overhead of the networking protocol.

6. Evaluation

In this section, we evaluate the performance of different Jasmin aspects, e.g., communication speed, memory and disk footprints. As a macro-benchmark, we evaluate the performance of a multimedia application. We also evaluate the efforts of porting such application from a legacy form to Jasmin.

Benchmarks are performed on a computer equipped with 12 gigabytes of RAM and 12 processor cores running at 2.67 GHz. The computer runs the operating system *Debian 6.0* on the *Linux kernel* version `2.6.39-amd64`.

6.1. Communication Speed Benchmarks

Aiming to assess Jasmin remote calls efficiency, we compared Jasmin proxies with the standard RPC implementation of Linux Debian 6.0. The comparison with an industry standard such as RPC proves that Jasmin meets common requirements for industrial deployment.

Communication speed evaluation is based on two functions: (1) a *simple* function taking no input and producing 4 bytes, and (2) a *complex* function taking 27 bytes of input and producing 12 bytes. The main difference between the two functions is the input/output bytes involved in each function call. We measure time taken to call these functions:

- Inside one application container, i.e., a local function call.
- Between two Jasmin applications running in different Linux containers and communicating via Jasmin proxies (see Section 5).
- Between two processes communicating via RPC.

The results of this benchmark are given in Table 2.1. RPC achieves poorer performances than Jasmin proxies but they are in the same order of magnitude compared to local function calls. The results of RPC calls reveal that the size of input/output parameters has little effect on call performance, i.e., the overhead is 0.16% per transferred byte. This indicates a stable marshaling mechanism, but this means that the minimum overhead is already too high. The significant difference seen between the results of Jasmin proxies calls indicates that the marshaling mechanism performance highly depends on the input/output parameters of called functions, as we observe an overhead of 2.08% per transferred byte. Furthermore, the minimum overhead of Jasmin proxies is 60% less than the minimum RPC overhead.

	Simple function call	Complex function call	Overhead per transferred byte
Local call	9 ns	1 421 ns	402.28%
Jasmin Proxies	10 767 ns	19 486 ns	2.08%
RPC	25 842 ns	27 470 ns	0.16%

Table 2.1: Comparison of execution speeds of function calls.

6.2. Disk Footprint

The disk footprint of Jasmin is composed of several parts: (1) the binaries of the Jasmin middleware in different forms, and (2) the binaries of Jasmin applications, and (3) the set of user-land programs local to every Linux container. This section quantifies these parts and provides optimization opportunities.

The Table 2.2 reports a size on disk of 51 megabytes for the executable files representing the different parts of Jasmin and of Linux containers created by Jasmin. The major part of disk space usage is required by the Linux containers, as Jasmin executable files form no more than 2% of the required disk space. This relatively large disk footprint is due to the need for a Linux container to have a minimal set of user-land programs installed in a dedicated file system. However, if Linux containers isolation is not used, then the disk footprint drops to 762 kilobytes for a configuration of a root Jasmin controlling one local Jasmin running zero Jasmin applications.

	Size on disk, in kilobytes		
Standalone Jasmin execution environment	398		
Root Jasmin execution environment	426		51962
Local Jasmin execution environment	336	51536	
Linux Container	51200		

Table 2.2: Jasmin disk footprint.

Optimizing the Disk Footprint

To reduce Linux containers disk footprint, we create the Linux container file system using a modified version of the `multistrap` program of the Emdebian [125] project. This results in disk footprint of 50 Megabytes, as shown in Table 2.2. On an ARM-based machine, the footprint goes down to 38 megabytes thanks to the high density of the ARM machine code *Thumb-2*.

Multiple Linux containers often have many identical files (e.g., programs, libraries) and few different files (e.g., configuration files, logs). Thus, using a file systems that offers the *Copy-on-Write* feature can strip the disk footprint down to a few megabytes. Examples of these file systems include ZFS [24], Reiser4 [109], and Btrfs [35]. This still ensures correct file system isolation between Linux containers, because if a file shared between two Linux containers is altered, then the file content will be duplicated to give each Linux container a private copy of the file. Using these file systems makes Linux containers better scaled for embedded systems and for systems running a very large number of Linux containers.

6.3. Memory Footprint

The Table 2.3 shows the sizes in memory of the processes running different parts of Jasmin and of Linux containers. For a configuration of one root Jasmin controlling one local Jasmin inside a Linux container with no Jasmin applications running, the Table 2.3 reports 552 kilobytes of memory usage, 54% of which is consumed by Jasmin processes. The more local Jasmin execution environments are run and managed by the root Jasmin, the lower the memory consumption ratio of Jasmin execution environments becomes.

	Size in memory, in kilobytes		
Standalone Jasmin execution environment	128		
Root Jasmin execution environment	172		552
Local Jasmin execution environment	124	380	
Linux Container	256		

Table 2.3: Jasmin memory footprint upon start up.

6.4. Performance and Porting Efforts of a Legacy Multimedia Application

This section first shows the steps and costs needed to port a legacy multimedia application, originally written as a simple `C` application, to Jasmin. Then, it shows the application deployment steps. Finally, it measures performances of the ported application and compares them with its legacy version performances. It is worth noting that new applications targeted at Jasmin are developed directly as MIND components based on `OSAL`; porting is only needed for legacy code.

Evaluation Metrics. We evaluate porting costs both in number of lines of code to write and in number of components to build. We evaluate deployment costs in terms of number and nature of operations to perform. Then we evaluate performances by measuring CPU and memory consumption of the application in its legacy version and in its component-based version targeted at Jasmin execution environment.

Jasmin Player Application. *Jasmin Player* is a Jasmin multimedia application made for demonstration purposes of Jasmin. The application displays a window presenting a list of media files, and enabling selection of a media file to play on the window. The media files can be local or remote. Some of their meta-data are stored in a *media library*: an SQLite [57] database managed by the application. Their contents are decoded and rendered by `libVLC` [130] library. The application windows are created and managed via the `GTK+` [134] library.

Porting a Legacy Multimedia Application to Jasmin

First, porting Jasmin Player to Jasmin required redesigning it in terms of components, instead of procedures. We defined five top-level components, each assigned a simple responsibility. Then, we defined and implemented interface methods provided by each component, inspired of the procedures previously defined in the legacy version. Finally, we completed components dependencies in terms of interfaces.

Porting Legacy Libraries. `SQLite` and `libVLC` legacy libraries were partially ported to Jasmin in matter of minutes. We ported only the parts required by the application:

- Porting `SQLite` required 65 lines of code in 3 files to produce the `sqlite.sqlite` component.
- Porting `libVLC` required 53 lines of code in 5 files to produce the `libvlc.vlc` component.

This shows that porting legacy code in Jasmin applications is *fast* and can be *progressive*. We could also use the MIND `@Wrap` annotation which automatically wraps entire libraries as components, but we did the job by hand for demonstration purposes.

The `GTK+` library was not wrapped in a Jasmin component. Instead, it was directly used from interfaces implementations of application components. This illustrates that porting libraries is an optional task in Jasmin.

Porting Application Legacy Code. Porting the application code is slightly more complicated as it involves redesigning it in component-oriented paradigm. However, most of the application code is reused as component interface implementations. Some minor code changes are required to convert system calls to `OSAL` method calls, and to convert function calls into method calls. This produced 3 components:

- `db.media` manages the multimedia database based on `sqlite.sqlite`.
- `gui.window` manages the application window and GUI events.
- `player.player` manages the media player state and controls `libvlc.vlc` to decode and render the media on the window.

Deployment of Jasmin Player. The Jasmin Player application is compiled using the MIND compiler into 5 binaries that are copied into a Jasmin repository. The Jasmin interactive administration console exposes a *unified* set of commands for local Jasmin and standalone Jasmin. These commands invoke administration services provided by Jasmin. Thus, deploying the components in a standalone Jasmin (see Section 2) or in a local Jasmin (see Section 3) is very similar. What is mainly needed is asking the administration console to `install` each of the 5 components, then `resolve gui.window`, then `start` it. Once installed, the Jasmin execution environment manages dependencies automatically. Because `gui.window` depends directly or

indirectly on all remaining components, it is enough to issue `resolve` and `start` commands on it, and let the Jasmin execution environment automatically resolve and start the remaining components as needed.

Evaluation of Jasmin Player. Figure 2.4 and Figure 2.5 respectively illustrate the CPU and memory footprint of running Jasmin Player in different deployment environments:

- Direct running of legacy application on the host.
- Running of component-based Jasmin Player inside a standalone Jasmin.
- Running of component-based Jasmin Player inside a local Jasmin execution environment running directly on the host system and managed by a root Jasmin.

Figure 2.4 shows that Jasmin Player behaves almost identically in the three configurations, consuming about 4% of CPU to start, and around 6±3% of CPU when playing a sample video clip, and nearly 1% of CPU to terminate. The difference in CPU consumption patterns is due to measure imperfection [88] and to other programs running in background.

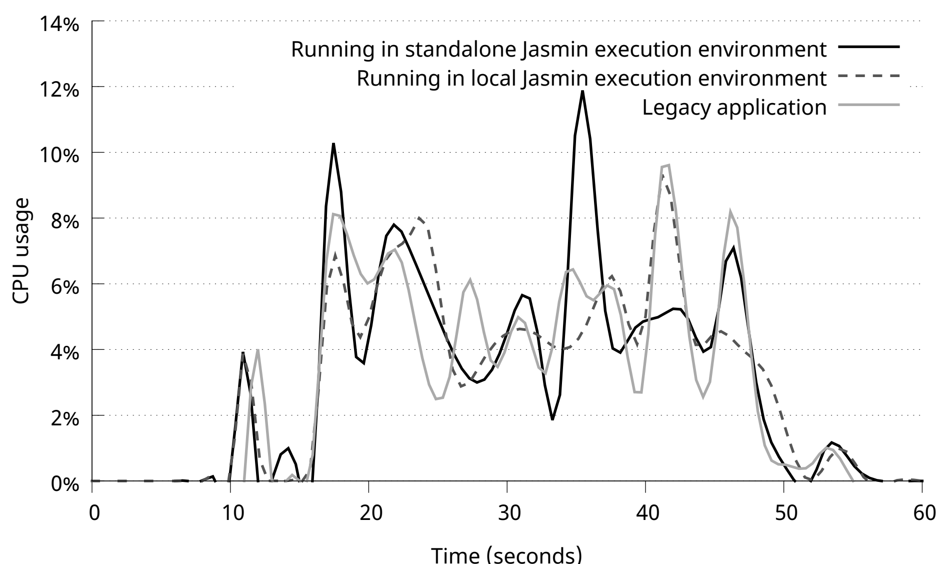


Figure 2.4: CPU footprint of running Jasmin Player on different Jasmin execution environments.

Figure 2.5 illustrates a memory usage behavior that is very similar in the 3 execution environment configurations. Starting the application consumes roughly 4 megabytes of memory. The VLC library first allocates around 23 megabytes of memory to start playing the video clip, then drops to 16 megabytes while playing. But, when the application runs in the local Jasmin, memory consumption is 3 megabytes higher while playing the video clip. This is not due to internal management performed by the local Jasmin itself, as it allocates most of

its data structures when the application starts. This is, most likely, a secondary effect of a caching mechanism performed by the VLC library.

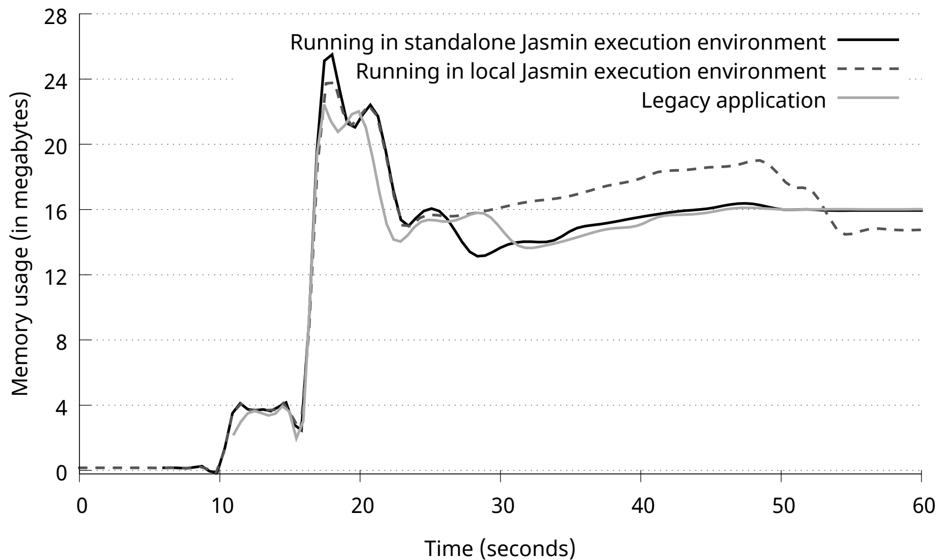


Figure 2.5: Memory footprint of running Jasmin Player on different execution environments.

6.5. Summary of Results

The micro-benchmarks of communication speed, memory and disk footprint indicate that Jasmin has a relatively small overhead, particularly if Linux containers isolation is not used. If Linux containers are used, then more disk space is required, but the disk footprint can be stripped down to a few megabytes as described in Section 6.2.

The evaluation of Jasmin Player application shows the easiness of porting legacy applications and using legacy libraries. The Jasmin administration console makes it easy to deploy Jasmin applications, automating most of deployment tasks. We also illustrate that CPU and memory overhead of different types of Jasmin execution environments is negligible compared to the consumption of the application itself.

7. Conclusion

In this chapter, we presented Jasmin, an open and robust smart home middleware that hosts applications providing services targeted at the end user. The services offered by the Jasmin middleware pave the ways to new and attractive business models, such as *Business to Business to Consumer* (B2B2C), by opening the smart home platform to any service provider that aims to expose his services directly inside the end user home.

Jasmin follows the Service-Oriented Architecture paradigm [103] and enables easy and dynamic deployment of applications by automating most deployment steps. Jasmin runs applications based on the MIND framework implementing the Fractal component model which urges developers to produce cleanly designed services. This allows rapid and easy development and deployment of services, and thus contributes to making Jasmin an attractive platform for numerous service providers.

The security and robustness risks introduced by this openness are solved by Jasmin through isolation containers. Jasmin offers isolation containers in multiple selectable levels suitable for different applications requirements and trust levels. It implements its highest level of isolation based on Linux containers, which offer high isolation guarantees at a low performance cost.

In order to build rich services, service providers need to easily invoke existing services. This implies that isolated applications must still communicate. Jasmin takes care of the complexity of remote service invocation by automatically loading interface proxies and performing marshaling and data transfer as needed. This enables services consumers to easily and seamlessly invoke local and remote services.

Jasmin runs applications on top of an Operating System Abstracting Layer (OSAL). This enables easy portability of applications to different operating systems, and eases porting of Jasmin itself to other systems. It is mainly a big step to master the smart home heterogeneity.

Jasmin evaluation shows that Jasmin respects the embedded nature of home devices. In fact, Jasmin not only has a low resource usage in terms of CPU and memory and disk space, but also incurs a very low overhead on the applications it hosts.

Chapter 3.

Incinerator – Stale references detection and elimination

Contents

1 Memory Leaks and State Inconsistencies **54**
2 Introducing Incinerator **54**
3 Detecting and Eliminating Stale References **55**
3.1 Stale Class Loaders 56
3.2 Synchronization Handling in Incinerator 56
3.3 Finalization Handling in Incinerator 57
4 Implementation Extents **59**
4.1 Changes to the Java Virtual Machine 59
4.2 Monitoring Bundles Updates and Uninstallations 61
4.3 Modifications to the Just-In-Time Compiler 61
5 Functional Validation and Performance Evaluation **62**
5.1 Stale Reference Micro-Benchmarks 62
5.2 Bundle Conflicts 65
5.3 Memory Leaks 66
5.4 Stale References in Knopflerfish 67
5.5 Performance Benchmarks 67
6 Conclusion **68**

This chapter describes Incinerator, our proposed solution to detect and eliminate stale references in Java. We first describe our prototype in the context of a middleware for the smart home gateway based on OSGi and Java. We show that stale references represent a recurring and severe issue in this context. Then we illustrate our solution to the problem, in terms of design decision, and implementation extents. We illustrate the interaction between Incinerator, which is implemented inside the Java virtual machine, and the rest of the subsystems composing the Java virtual machine and the OSGi framework. Following this, we evaluate Incinerator features using numerous stale reference scenarios to show that Incinerator detects all stale references. Then, using an example smart home application, we illustrate the state inconsistencies caused by stale references and the hazards it can cause to the end-users. We also show how Incinerator enabled us to contribute back to the open source community by detecting and correcting a stale reference in the Knopflerfish OSGi framework. Finally, we evaluate the overhead of Incinerator based on DaCapo benchmarks.

1. Memory Leaks and State Inconsistencies

The Home Gateway Initiative [59] endorses the OSGi framework as a middleware to host applications running in the smart home gateway. OSGi supports application *hot-swapping*, and *isolates* applications, and allows *direct communication* between applications by running them in the same address space (see Section 4.3).

Running all bundles in a single address space makes OSGi prone to *inconsistencies* and *memory leaks*, as described in Sections 6.4.2 and 6.4.3, and as stated by the OSGi specification [127] and as demonstrated in practice by Gama and Donsez [47]. These problems may arise when a bundle is uninstalled or updated, if a reference obtained before the bundle uninstall or update is retained by another bundle. The consequences of memory leaks may range from user annoyance, for entertainment services, to life critical issues for health care and security-related home services.

Experiments made by Gama and Donsez [47], the authors of Service Coroner, already showed that numerous stale references exist in several open source applications and frameworks, as the Table 1.1 illustrates. Furthermore, because Service Coroner detects only stale *service* references, many other types of stale reference go undetected, which is why we believe that the real amount of stale references is even higher than what is shown in the Table 1.1.

2. Introducing Incinerator

We address the problems raised by OSGi stale references at the Java Virtual Machine level by proposing Incinerator [9]: an OSGi-aware extension to garbage collector. Incinerator's approach is to integrate stale reference detection into the garbage collection phase. This approach induces low overhead since the garbage collection phase already traverses all live objects, and checking the staleness of a reference needs few operations. This approach is also independent of the specific garbage collector algorithm as it only requires the modification of the function that scans the references and objects contained inside a given object. When Incinerator finds a reference, it checks whether the referenced object belongs to an uninstalled bundle or to a previous version of an updated bundle. In this case, the reference is identified as stale and Incinerator sets it to `null`, as illustrated in Figure 3.1. As a consequence, no stale object remains reachable at the end of the collection and the associated memory is reclaimed by the garbage collector.

Compatibility Notes

Incinerator changes the behavior of the Java Virtual Machine, because it nullifies references that are found to be stale. We investigated the compatibility issues that could arise from such change. It should be noted that a correctly written bundle that releases its stale references, when a bundle is uninstalled or updated, is never affected by Incinerator. When a bundle does not release some stale reference, our design choice is to minimize the impact of nullification

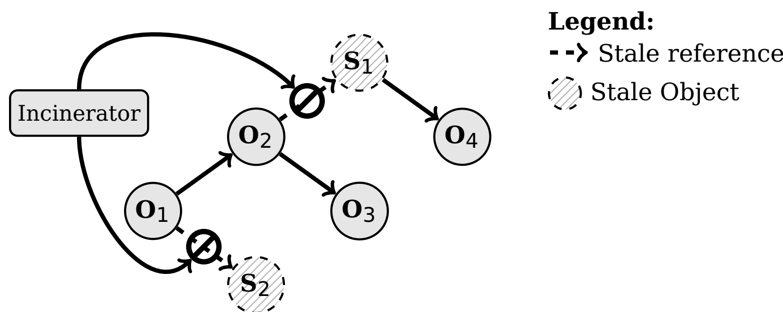


Figure 3.1: Incinerator eliminating stale references

while ensuring that the memory is released. Three situations can occur, depending on how the stale reference is used:

- If the stale reference is never used, then nullifying it has no impact on the other bundles.
- If the stale reference is only accessed as part of a cleanup operation, i.e., in `finalize()` method, then Incinerator executes this cleanup operation in order to avoid other kinds of leaks.
- If the stale reference is used elsewhere, either to access or to synchronize on the stale object, then the bundle that holds the stale reference is *buggy* since using the stale object would lead to possibly conflicting operations. Since the reference has been nullified, such a buggy bundle receives a `NullPointerException`, which helps the developers track down the bug, by making it visible. If a thread is blocked while waiting for a synchronization on the stale reference, Incinerator also unblocks the thread, in order to prevent dead locks, or leaking of the thread and its reachable objects.

3. Detecting and Eliminating Stale References

The goal of Incinerator is to eliminate stale references by setting them to `null`. In order to do so, Incinerator needs to scan all live references in the Java memory space to determine whether they are stale. Since such a scan is already done in the garbage collector, we chose to design Incinerator as an extension to the garbage collector. This approach has a low performance penalty since in most cases the overhead is limited to the cost of checking the staleness of each reference during the heap traversal performed by the garbage collector.

In this section, we first present how stale references are identified in class loaders. Then, we discuss more specifically the other JVM features impacted by stale references: synchronization and finalization.

3.1. Stale Class Loaders

Based on Definition 1.12, a stale class is a class loaded by a stale class loader. A stale object is an object of a stale class. And a stale reference is a reference to a stale class loader, or a stale class, or a stale object. Consequently:

Corollary 3.1. A stale reference is a references to (1) a stale class loader, or (2) a class loaded by a stale class loader, or (3) an object of a class loaded by a stale class loader.

The Java language and the JVM specifications do not define any notion of stale references. Stale references are consequence of the additional object states introduced by OSGi. This is why the JVM cannot distinguish a stale reference or class or object from non-stale ones. Therefore, we had to store the additional stale state, in order to enable Incinerator to distinguish stale references from non-stale ones. Thanks to the Corollary 3.1, we know that the stale property concerns only class loaders, which is why we added a hidden field in each class loader to store the *stale flag*. The stale flag is cleared when the class loader is constructed. If the class loader is associated to a bundle, then the stale flag is set when the bundle is uninstalled or updated. When Incinerator analyzes a reference, it accesses its referenced object, then its class, then its class loader, to finally read the stale flag, which indicates how Incinerator should treat the reference.

3.2. Synchronization Handling in Incinerator

In Java, each object has an attached monitor, whose purpose is to provide thread synchronization. The list of the threads blocked while waiting for the monitor is stored in the monitor structure, which can only be retrieved through the object. Therefore, if a thread is holding the monitor at the time when the associated object becomes stale and Incinerator nullifies all references to the object, the holding thread will become unable to reach the monitor structure to unblock any blocked threads. These threads would remain blocked, leaking both their thread structures and any referenced objects.

Incinerator addresses this issue by waking up the blocked threads in a cascaded way. To allow each wakened thread to detect that the monitor is stale, we add a *stale* flag to the monitor structure. During a collection, when Incinerator finds a stale object with an associated monitor, it nullifies the stale reference, marks the monitor as stale, and then wakes up the first blocked thread. The thread wakes up at the point where it blocked, in the monitor acquiring function. We thus modify this function so that when a thread wakes up, it checks the stale flag. If the flag indicates that the monitor is stale, the monitor acquiring function wakes up the next blocked thread and throws a `NullPointerException` to report to the current thread that the object is stale. Note that there is no special treatment of the thread that is actually holding the monitor. This thread will receive a `NullPointerException` when it next tries to access the stale object.

Most modern Java Virtual Machines allocate a monitor structure that is separate from the object and is managed explicitly [13]. This monitor structure is normally freed during a collection when the memory of its associated object is reclaimed. With Incinerator, when a

stale object is reclaimed, its monitor structure has to survive the collection, if threads are blocked on it, so that each thread can wake up the next one. We thus further modify the monitor acquiring function so that it frees the monitor structure at the end, when it detects that there are no remaining blocked threads.

3.3. Finalization Handling in Incinerator

In Java, a `finalize()`[1] method defines clean up code that is executed exactly once by the garbage collector before reclaiming the memory associated with the object [55]. If a `finalize()` method accesses a stale reference that was nullified by Incinerator, then it would encounter a `NullPointerException` and it would not able to complete the clean up. This may lead to other kinds of resource leaks, such as never closing a file descriptor or a network connection. We have encountered this case when a bundle defines *finalizable* objects, i.e., objects that implement a `finalize()` method that has not yet been called. Indeed, when the bundle is uninstalled or updated, its finalizable objects become unreachable. These finalizable objects often use cross-bundle references, which are thus stale, and nullifying these references prevents the execution of the `finalize()` methods.

3.3.1. Garbage Collector Handling of `finalize()`

In a standard JVM, a collection is usually performed in two phases. During the first phase, the garbage collector identifies unreachable objects by scanning the heap. During the second phase, it manages finalizable objects. The garbage collector does not reclaim the memory of an unreachable finalizable object at this time because the object can *resurrect* [97], i.e., it can become reachable again during the execution of its `finalize()` method, for example, by storing a reference to itself in a reachable object or in a static variable. Instead, during the second phase, the garbage collector marks each unreachable finalizable object and its reachable sub-graph as reachable. It also marks the unreachable finalizable objects as *finalized* to ensure that their `finalize()` methods are not executed more than once. Finally, after the second phase, the garbage collector executes the `finalize()` methods. Since the objects are now finalized, they are managed as normal objects by the garbage collector on the next collection cycle, and their associated memory will be reclaimed later if they are again detected as unreachable.

3.3.2. Incinerator Handling of Finalizable Objects

Incinerator is designed with the goal of preventing resource leaks. This is the reason Incinerator allows `finalize()` methods to run without introducing exceptions due to `null` pointers, by deferring the nullification of stale references to the collection following the execution of the `finalize()` method. After the marking phase of the garbage collector, we distinguish two kinds of finalizable objects: reachable objects and unreachable ones.

[1] `java.lang.Object.finalize()`

For a finalizable object that is reachable at the end of a collection, i.e., a resurrected object, it is not known when and if the `finalize()` method will be executed. Deferring nullification of stale references until after the `finalize()` method is executed may indefinitely prevent Incinerator from performing nullification, and thus cause memory leaks. In this case, Incinerator avoids the memory leak by nullifying the reference, at the risk of failing during a later execution of the `finalize()` method.

For a finalizable object that is unreachable at the end of a collection, the `finalize()` method is run just after the garbage collection. In order to ensure that the `finalize()` method will complete its execution successfully in this case, Incinerator defers the nullification of the stale references reachable from this object to the next collection cycle. To defer nullification, we have modified the function that scans objects during the collection and added code at the end of the scanning phase of the collection cycle. Incinerator performs the following algorithm, illustrated in Figure 3.2:

1. The first step is performed when the garbage collector initially scans the heap. Figure 3.2a is an example graph object in the heap. The garbage collector discovers unreachable objects in this step. For each stale reference scanned, Incinerator (1) saves the reference address in an internal list (see *StaleRefList* in Figure 3.2b), and (2) aborts the scanning of the referenced object, which makes the object and its sub-graph unreachable. This is case of C, F and H in Figure 3.2b.

2. The second step is performed when the garbage collector marks as reachable each unreachable finalizable object and its reachable sub-graph, as illustrated for the object D in Figure 3.2c. Incinerator leverages the scanning of these objects performed by the garbage collector. For each stale reference scanned, Incinerator removes it from the internal list. This is demonstrated by removing the reference D→C from *StaleRefList* in Figure 3.2c. Furthermore, during this step, Incinerator does not abort the scanning of the stale reference, and consequently lets the garbage collector scan its reachable sub-graph, which is the reason C becomes reachable again in Figure 3.2c. After the garbage collector has marked all the unreachable finalizable objects as reachable, the internal list of Incinerator contains only references that are both (1) stale, and (2) not reachable from an unreachable finalizable object.

3. The third step is performed at the end of the scanning phase of the garbage collection cycle. Incinerator sets to `null` all references remaining in the internal list, while handling the issues related to synchronization on stale objects, discussed in Section 3.2. This renders many stale objects unreachable, which enables the garbage collector to reclaim their memory in the remaining of this garbage collection cycle. This explains why objects F and H disappear in Figure 3.2d.

4. After this first garbage collection cycle, the JVM finalization threads execute the `finalize()` methods of objects that were unreachable and finalizable in the previous garbage collection cycle. This is the case of the object D in Figure 3.2e. Once `finalize()` executed for each of those objects, the JVM removes its references to the object. If the object becomes unreachable, then it will be reclaimed in the following garbage collection cycle, as illustrated in Figure 3.2f. But if the object is still reachable, then the object was resurrected. If the resurrected object still holds stale references, then these will be removed in the following garbage collection cycle by Incinerator without

delay, because the object was already finalized, and thus became non-finalizable.

The algorithm presented above is designed to protect against a buggy bundle, but not a deliberate attack. As such, it is possible to construct a malicious bundle that can keep a stale object from ever being reclaimed by the garbage collector. As Incinerator defers the nullification of a stale reference reachable from an unreachable finalizable object, the stale reference will survive a collection cycle. A `finalize()` method of a malicious bundle can force the stale reference to survive one more collection cycle by creating a new unreachable finalizable object that references the stale object. By repeating the same pattern, the malicious bundle can indefinitely delay the nullification of a stale reference to the stale object. To protect against such attacks, Incinerator adds a counter to each stale class loader to indicate how many collection cycles it survived. Each time a reference is removed from the stale references internal list, the counter of the class loader of its referenced object is incremented, if not already done during that collection cycle. When the counter value of a stale class loader is greater than zero, Incinerator does not delay the nullification of stale references that point to objects of that class loader. This ensures that a stale object cannot be resurrected more than once.

4. Implementation Extents

We prototyped Incinerator in J3: an experimental Java Virtual Machine based on VMKit [51] and the Low-Level Virtual Machine (LLVM) [74] and the Memory Management Toolkit (MMTk) [20]. We tested Incinerator on the *Mark&Sweep* garbage collector of MMTk, and the Knopflerfish [81] 3.5.0 framework, one of the main OSGi implementations. Implementing Incinerator required adding 1000 lines of code to J3, mostly in C++.

4.1. Changes to the Java Virtual Machine

Within the JVM, Incinerator requires changes in the garbage collector, in the support for class loading, and in the monitor implementation. The garbage collector is modified as described in the previous section. Incinerator additionally creates a map in which to store the association between a bundle ID and its class loader. Such map is needed because OSGi does not provide an interface to retrieve the class loader of a bundle. Finally, the monitor implementation is modified to support the algorithm described in Section 3.2.

As an optimization based on Definition 1.12, Incinerator is only enabled when stale references potentially exist. For this purpose, we have added a global flag that is set when a bundle is uninstalled or updated, since a new stale reference can only appear under these conditions. Incinerator clears the flag at the end of a collection when it is certain that all stale references were eliminated, i.e., if Incinerator did not find any stale references that are reachable from an unreachable finalizable object. Incinerator checks this flag at the beginning of a collection and appropriately switches between the original scan function and the Incinerator scan function.

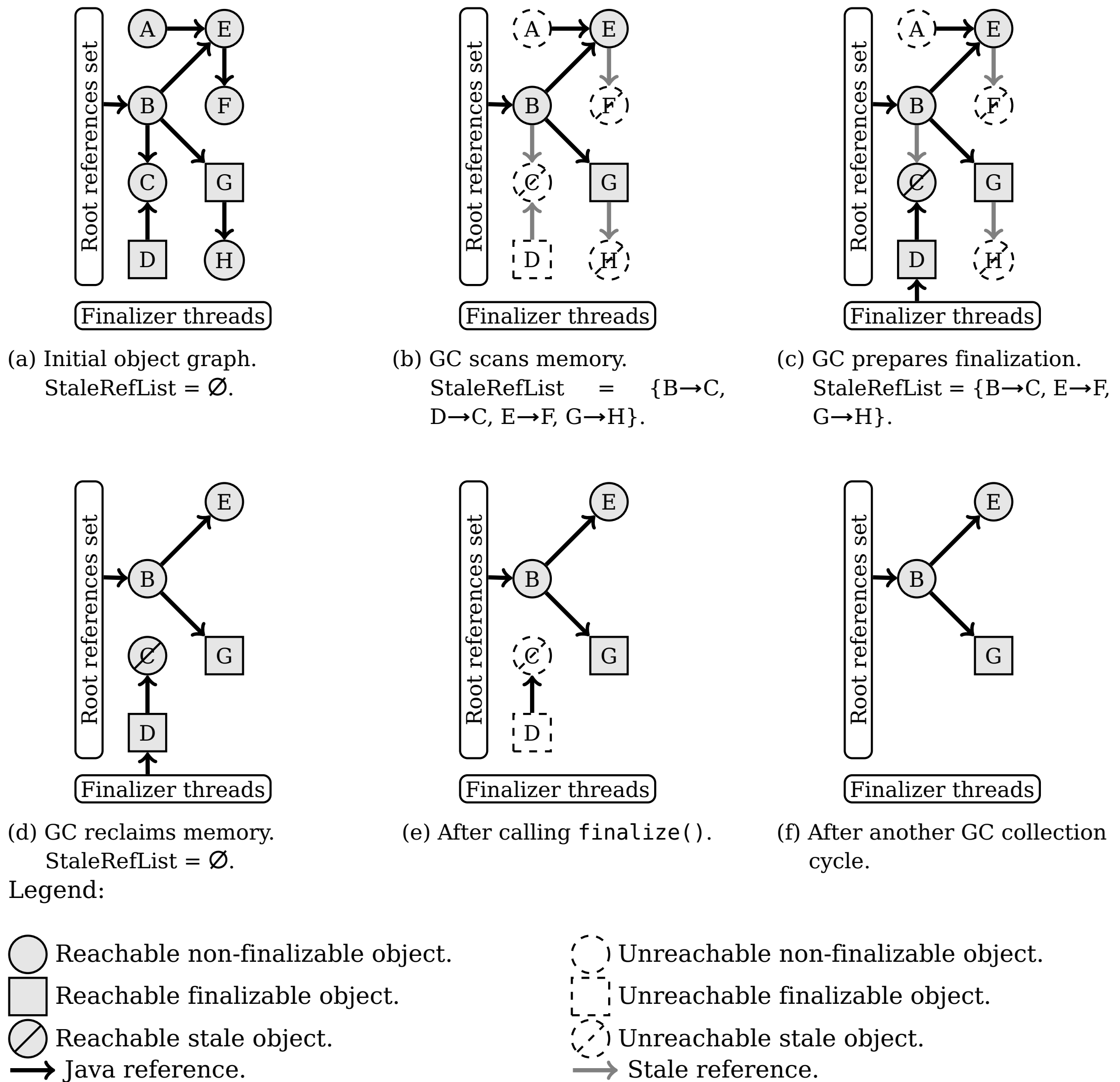


Figure 3.2: Incinerator handling algorithm for finalizable objects.

4.2. Monitoring Bundles Updates and Uninstallations

Incinerator runs an administration bundle that listens to other bundles' changes. When a bundle is uninstalled, the administration bundle calls the native method provided by Incinerator to set the *stale* flag of the associated class loader. We also modify the OSGi framework to populate the Incinerator bundle ID association map when a bundle is updated or a new bundle is installed. These modifications are straightforward and do not alter the behavior of OSGi. For example, the Knopflerfish OSGi framework 3.5.0 requires only 10 lines of additional Java code.

4.3. Modifications to the Just-In-Time Compiler

The Java language specification [55] states that, if an exception is raised while holding a monitor, the Java compiler has to generate an exception handler that releases the monitor. This introduces the risk of an infinite loop when using Incinerator, as follows:

1. Incinerator nullifies the reference to the object containing the monitor, because the object is stale.

2. Some Java code accesses the reference and therefore receives a `NullPointerException`.

3. The execution is transferred to the exception handler created by the compiler.

4. The exception handler accesses the nullified reference to release the monitor lock, and consequently receives a `NullPointerException`, which transfers execution to point 3.

This causes an infinite loop between points 3 and 4.

To avoid this issue, we have modified the Just-In-Time compiler so that the code generated for a monitor lock release[2] simply leaves the block silently when the monitor is `null`, rather than raising a `NullPointException` exception. This workaround does not change the behavior of programs compiled from Java source code, as the Java compiler caches the reference given to a `synchronized` block in a local variable hidden from the Java code, but visible to the JVM. As a consequence, the argument of the monitor lock release instruction can never be `null`, because if it were `null`, the preceding monitor lock acquire instruction would have thrown a `NullPointerException`. However, our workaround is incompatible with the Java specification and could change the behavior of programs written directly in Java bytecode or generated in an ad-hoc fashion.

[2] The `MONITOREXIT` bytecode instruction.

5. Functional Validation and Performance Evaluation

This section presents an evaluation of Incinerator in terms of both the increase in robustness and the performance penalty. We evaluate robustness from a number of perspectives. First, we present a test suite of ten micro-benchmarks that we designed to cover the possible sources of stale references. This test suite is used to compare the behavior of Incinerator with Service Coroner [47], a tool that instruments the OSGi reference management calls and that detects stale references by analyzing the object references in a memory dump. Second, we show the potential impact of bundle conflicts in the context of a simple gas alarm application. Then, we show the impact of repeated memory leaks caused by stale references. We then present a concrete case of a stale reference bug found in the Knopflerfish OSGi framework. Finally, we study the performance overhead of Incinerator by running the DaCapo 2006 benchmark, which is representative of standard Java applications.

We executed all benchmarks on two computers, both of which run Debian 6.0 with a Linux 2.6.32-i386 kernel:

- A low-end computer with a 927 Mhz Intel Pentium III processor, 248 megabytes of RAM and 4 gigabytes of swap space, which has a performance comparable to that of a typical system in a smart home environment. The swap space is necessary, because J3 requires 1 gigabyte of address space to run the DaCapo benchmark.

- A high-end computer having two 2.66 Ghz Intel Xeon X5650 6 core processors, 12 gigabytes of RAM and zero swap space.

Service Coroner was executed using Knopflerfish 3.5.0 on top of Sun JVM 6.

5.1. Stale Reference Micro-Benchmarks

In order to assess the scope of the stale reference problem, we designed a test suite of ten micro-benchmarks that cover the possible sources of stale references and their impact on the JVM. The scenarios of these micro-benchmarks are classified by four criteria: *OSGi visibility*, *scope*, *synchronization*, and *finalization*. We executed the micro-benchmarks using J3, Hotspot 6, Service Coroner/Hotspot 6, and Incinerator.

Figure 3.3 shows the bundle configurations we used in our scenarios. We consider three bundles A, B, C. A creates the object S, and B creates the finalizable object F.

Stale Reference OSGi Visibility. OSGi visibility refers to whether the reference is visible to the OSGi framework, i.e., whether a bundle must call the OSGi framework to obtain the reference. By instrumenting calls to these API, one could keep track of the references given to bundles. Service Coroner uses this technique for detecting stale references. We have developed Scenarios 3.1 and 3.2 to illustrate reference visibility to OSGi.

Scenario 3.1 (OSGi-visible stale reference, see Figure 3.3a)**.** C holds a reference to S which is managed by OSGi. The reference C→S is made stale by uninstalling A.

Scenario 3.2 (OSGi-invisible stale reference, see Figure 3.3b)**.** C holds a reference to S which is not managed by OSGi. The reference C→S is made stale by uninstalling A.

Stale Reference Scope. Scope refers to the location of the reference, i.e., in a local variable, in a global variable, in an object field, or in a thread-local variable. Different locations are scanned in different ways and orders by the garbage collector. We designed Scenarios 3.3 to 3.6 in order to check that Incinerator finds stale references in all kinds of locations.

Scenario 3.3 (Stale reference in local variable, see Figure 3.3b)**.** C holds a reference to S in a local variable. The reference C→S is made stale by uninstalling A.

Scenario 3.4 (Stale reference in global variable, see Figure 3.3b)**.** C holds a reference to S in a global variable. The reference C→S is made stale by uninstalling A.

Scenario 3.5 (Stale reference in object field, see Figure 3.3b)**.** C holds a reference to S in an object field. The reference C→S is made stale by uninstalling A.

Scenario 3.6 (Stale reference in thread-local storage, see Figure 3.3b)**.** C holds a reference to S in a thread-local variable. The reference C→S is made stale by uninstalling A.

Synchronization on Stale Objects. Synchronization refers to whether the referenced object monitor is used to synchronize threads. As stated in Section 3.2, if threads are blocked while waiting to obtain the monitor of a stale object, then Incinerator wakes up the blocked threads and only releases the memory of the object monitor when the last thread wakes up. Scenario 3.7 illustrates this situation.

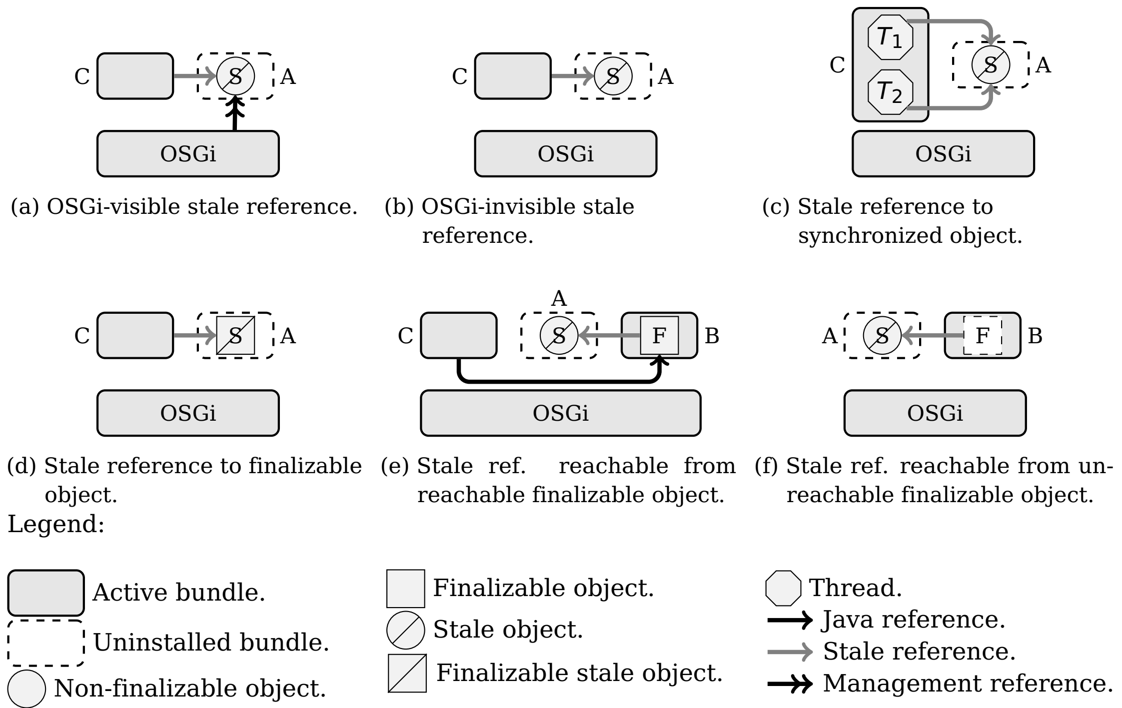
Scenario 3.7 (Stale reference to a synchronized object, see Figure 3.3c)**.** C runs two threads T_1 and T_2 that synchronize on S via two references. Both references T_1 →S and T_2 →S are made stale by uninstalling A.

Finalization of Stale Objects. Incinerator allows `finalize()` methods to run without introducing `NullPointerExceptions` by deferring nullification of stale references reachable by unreachable finalizable objects. To check the possible cases of finalizable objects, we defined Scenarios 3.8 to 3.10.

Scenario 3.8 (Stale reference to a finalizable object, see Figure 3.3d)**.** C holds a reference to S which is finalizable. The reference C→S is made stale by uninstalling A.

Scenario 3.9 (Stale reference reachable from reachable finalizable object, see Figure 3.3e)**.** C holds a reference to F which is finalizable. F retains a reference to S. The reference F→S is made stale by uninstalling A.

Scenario 3.10 (Stale reference reachable from unreachable finalizable object, see Figure 3.3f)**.** F is finalizable and unreachable. F retains a reference to S. The reference F→S is made stale by uninstalling A.



(a) OSGi-visible stale reference. (b) OSGi-invisible stale reference. (c) Stale reference to synchronized object.

(d) Stale reference to finalizable object. (e) Stale ref. reachable from reachable finalizable object. (f) Stale ref. reachable from unreachable finalizable object.

Figure 3.3: Stale reference scenarios.

Results and Conclusions

We executed our scenarios using Hotspot 6, J3, Service Coroner/Hotspot 6, and Incinerator. We did not evaluate Service Coroner with J3 because it requires a full Java 6 environment, which J3 does not support currently.

	Scenario 3.1	Scenarios 3.2 to 3.10
J3 or Hotspot 6	**Undetected**	
Service Coroner/ Hotspot 6	**Detected** Conditions: Service unregistered & garbage collection	**Undetected**
Incinerator	**Detected & Eliminated** Conditions: Bundle uninstalled or updated & garbage collection	

Table 3.1: Micro-benchmark execution with standard JVMs, Service Coroner and Incinerator.

Table 3.1 summarizes the behavior. Both J3 and Hotspot 6 suffer from memory leaks caused by stale references that go undetected in all scenarios. Service Coroner, used with Hotspot 6, detects OSGi-visible stale references in Scenario 3.1, thanks to the instrumentation of the OSGi calls transmitting references to the calling bundles. ServiceCoroner detects OSGi-visible stale references, but it does not eliminate them which leads to memory leaks. Service Coroner does not, however, detect OSGi-invisible stale references, as demonstrated by Scenarios 3.2 to 3.10.

Furthermore, Incinerator detects and eliminates all the stale references illustrated in the ten scenarios. In particular, Incinerator handles correctly the case of the stale references used for synchronization (see Scenario 3.7): the blocked thread is woken up by Incinerator when the reference to the stale object used for synchronization is nullified. Both threads receive a `NullPointerException`: (1) the thread holding the lock when it tries to release the lock, and (2) the thread blocked in the lock-acquiring method. Incinerator also handles correctly the three cases of stale references used by a finalizable object (see Scenarios 3.8 to 3.10), correctly executing the `finalize()` method as expected. After the execution of the `finalize()` methods, the memory of the stale objects is properly reclaimed by the garbage collector.

5.2. Bundle Conflicts

To demonstrate the risk of physical hazards and data corruption that can be caused by stale references, we prototyped an alarm application that is representative of a typical smart home system. Figure 3.4 shows an overview of the structure of this application. The application monitors a gas sensor device and fires a siren if it detects an abnormal gas level. The application accesses physical devices via two driver bundles: `SirenDriver` and `GasSensorDriver`.

The following experiment is performed:

1. Initially, the bundles `SirenDriver` 1.0 and `GasSensorDriver` are installed and started. Each bundle connects to its physical device (the alarm siren and the gas sensor, respectively) and exposes its features to smart home applications. `SirenDriver` 1.0 saves the alarm siren configuration in a simple text file describing parameters and their values.

2. When the bundle `AlarmApp` is installed and started, it obtains references to the services provided by `SirenDriver` 1.0 and by `GasSensorDriver`.

3. We upgrade the bundle `SirenDriver` from version 1.0 to 2.0. As part of the upgrade, the siren configuration file is converted to an XML[3]-based format, to simplify the addition of new configuration options. `SirenDriver` 1.0 is stopped and uninstalled, and thus disconnected from the alarm siren. When `SirenDriver` 2.0 is started, it connects to the alarm siren and exposes its new features. After this upgrade, the OSGi framework broadcasts an event to all bundles indicating that the bundle `SirenDriver` was updated.

4. We deliberately introduced a bug in `AlarmApp` so that it does not modify the reference it holds to the service provided by SirenDriver 1.0 when it receives the broadcast update event. This reference becomes stale.

After the upgrade, we observed three problems while executing the alarm application:

- The memory used by the JVM increased. We executed a garbage collection and observed, via debug logs, that `SirenDriver` 1.0 was not collected, thus leaking memory.

[3] eXtensible Markup Language

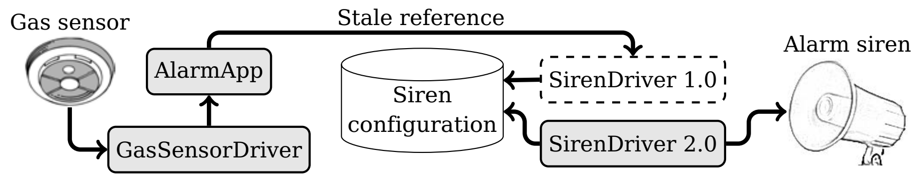


Figure 3.4: Hardware and software configuration of the Alarm controller application.

- We observed that changing the settings of the siren overwrites the XML configuration file by a file in the old text format. In fact, to change the settings of the siren, the `AlarmApp` invokes the service provided by `SirenDriver` 1.0 via its stale reference to the bundle. By doing so, `SirenDriver` 1.0 overrides the XML configuration file that was previously migrated and saved by the version 2.0. This problem is an example of *data corruption* caused by stale references.

- After simulating a gas leak to the gas sensor, we observed alarm signals repeatedly shown by the `AlarmApp`, but the siren remains silent. Despite the fact that the `AlarmApp` knows about the gas alarm reported by the gas sensor via `GasSensorDriver`, calling `SirenDriver` 1.0 does not activate the physical siren because that version is disconnected from the device, and only `SirenDriver` 2.0 provides the service. This problem makes the siren device unusable, and represents a *physical hazard* to the home inhabitants. This example is only meant for demonstration purposes, and such a bug would be easy to identify during the test phase because of the simplicity of the scenario. However, similar problems can occur in real applications which are more complex and thus harder to test exhaustively.

5.3. Memory Leaks

To investigate the memory leaks caused by stale references in quantitative terms, we repeated Scenario 3.1 (see Section 5.1) multiple times in order to create many stale references. In this experiment, a bundle C holds a reference to an object S (see Figure 3.3a). For the sake of clarity, we call S_i the object created by the ith version of A, called A_i. Each time we update the bundle A from version A_i to version A_{i+1}, the old object S_i becomes stale, and C keeps its reference to S_i and obtains a new reference to the new S_{i+1} object created by A_{i+1}. The bundle A is a small unit test bundle with 150 lines of Java code distributed over three classes.

For the baseline, J3, each update of the bundle A makes one more reference stale and costs the JVM 892 kilobytes of leaked memory. This is due to the need to keep all the bundle class information, static objects and generated machine code. After only 230 updates of the bundle A, the amount of leaked memory reaches 200 megabytes and J3 starts raising `OutOfMemoryExceptions` on the low-end test machine. Incinerator, however, continues to use the same amount of memory. These results show that, even stale references in small bundles, such as A, may leak significant amounts of memory.

5.4. Stale References in Knopflerfish

Knopflerfish is an open source OSGi framework implementation that is commonly used in smart home gateways because it is stable, and because it can run on the old version 1.4 of the Java runtime, still commonly used in the embedded market.

`HTTP-Server` is a bundle delivered with Knopflerfish and used by other bundles that expose Web-based interfaces. Web-based interfaces are commonly used in the context of smart home applications to interact with the end user. Therefore, `HTTP-Server` is a key bundle.

Using Incinerator, we identified a bug in `HTTP-Server` version 3.1.2. We discovered that, while updating `HTTP-Server`, some references to the objects of this bundle are not set to `null` as required. `HTTP-Server` defines a group of threads to handle transactions. When the bundle is uninstalled or updated, these threads are not destroyed by `HTTP-Server` as should be. As these threads reference objects allocated by the `HTTP-Server` bundle, the stale class loader stays reachable. `HTTP-Server` suffers thus from two different leaks: leaked threads, which silently continues to run, and stale references from these threads.

Stale references in the `HTTP-Server` bundle of Knopflerfish cause a loss of 6 megabytes of memory on each bundle update. Indeed, the HTTP-Server bundle contains 46 classes, which in total contain 8551 lines of Java code. The results also show that Incinerator does an efficient job by eliminating stale references in the long run, thus avoiding the memory leaks they cause and increasing the availability of the JVM, even in presence of stale reference bugs in running applications. When the leaked threads further access a stale reference, they receive a `NullPointerException`, which is not caught by `HTTP-server`, causing the threads to terminate execution. Therefore, Incinerator simultaneously solves the two leaks: the leaked thread is terminated and the memory of the stale bundle is reclaimed.

We sent a bug description and a patch [8] to the Knopflerfish development community. The patch destroys the leaked threads when the bundle is uninstalled and updated, thus avoiding the leaked threads and consequently the stale references. The patch was approved and has been integrated in the framework since the release 4.0.0. This shows that even well-recognized frameworks can suffer from the problem of stale references.

5.5. Performance Benchmarks

In order to measure the performance impact of Incinerator on real Java applications, we ran the DaCapo 2006 benchmark suite [21] on J3 and on Incinerator. The DaCapo [21] benchmarks includes eleven real Java applications stressing many JVM subsystems. We only use nine of the DaCapo applications, due to the constraints of the low-end computer. We excluded the “chart” benchmark because it requires an X Windowing Server running, and we excluded the “pmd” benchmark because of memory constraints.

This evaluation assesses the minimal impact of Incinerator, when there are no bundle updates, because DaCapo 2006 applications do not define bundles, except for the `eclipse` benchmark application. As compared to the baseline J3 JVM, Incinerator introduces an overhead in

the garbage collector for each scanned object in order to determine whether checking for stale references is required, and for each monitor acquisition in order to check whether the monitor belongs to a stale object.

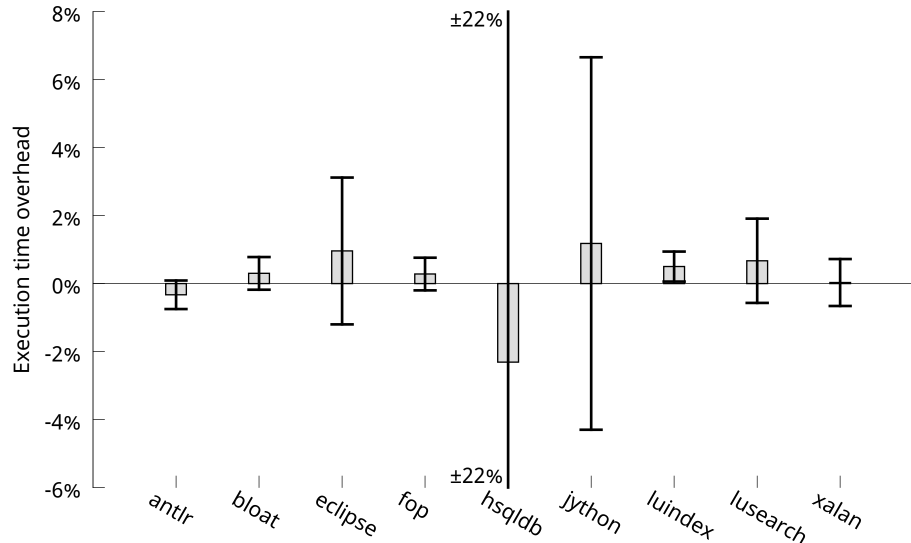


Figure 3.5: Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a low-end computer. `hsqldb` has a standard deviation of 22%, truncated here for clarity.

We performed 20 runs of all DaCapo benchmark applications on J3 and on Incinerator, on the low-end and the high-end computers described in the beginning of this section. On the low-end computer, Figure 3.5 shows that J3 performs better than Incinerator in 7 out of 9 applications, with a worst slowdown of 3.3%. On the high-end computer, Figure 3.6 shows that J3 performs better than Incinerator in 4 out of 9 applications, with a worst slowdown of 1.2%. But, as indicated by the standard deviations on Figures 3.5 and 3.6, these comparisons are inverted in some runs, i.e., Incinerator performed better than J3 in those runs. This is mostly due to disk and processor cache effects, and measurement bias [88].

Overall, our evaluation shows that Incinerator has only a marginal impact on performance and that it could be used in a production environment.

6. Conclusion

OSGi is increasingly being used in the smart home environment as a framework to host service-oriented applications delivered by multiple tiers. This makes the possibility of stale references a growing threat to the framework and to the running applications. In this chapter, we present Incinerator, which addresses the problem of OSGi stale references and the memory leaks they cause by extending the garbage collector of the Java virtual machine to take into account bundle state information.

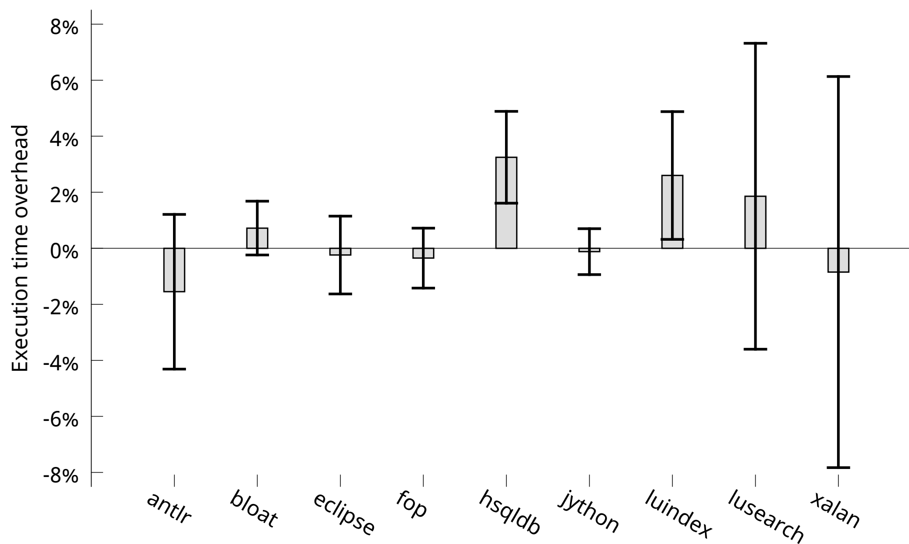


Figure 3.6: Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a high-end computer.

Incinerator detects more stale references than the existing stale reference detector, Service Coroner. Furthermore, while Service Coroner only detects stale references, Incinerator also eliminates them by setting them to `null`. This allows the garbage collector to reclaim the referenced stale objects. Indeed, we have found that stale references can cause significant memory leaks, such as the 6 megabytes memory leak on each update of the `HTTP-Server` bundle caused by the stale reference bug we discovered in Knopflerfish. Preventing memory leaks increases the *availability* of the JVM, which is an important metric in smart home gateways.

Incinerator is mostly independent of a specific OSGi implementation and, indeed, only 10 lines need to be modified in the Knopflerfish OSGi framework in order to integrate Incinerator. The CPU overhead induced by Incinerator is always less than 1.2% on the applications of the DaCapo benchmark suite on a high-end computer, and less than 3.3% on a low-end computer. The latter result shows that Incinerator is usable in smart home systems that have a limited computational power.

Chapter 4.

OSGi-Aware Memory Monitoring

Contents

1 Memory Monitoring Issues **71**
2 Our Solution to Memory Monitoring in OSGi **73**
3 Accurate and Relevant Memory Monitoring **73**
3.1 Assumptions 74
3.2 Goals 74
3.3 Domain-Specific Language for Resource Accounting Rules 75
3.4 Resource Accounting Algorithm 77
4 Implementation Details **78**
4.1 OSGi State Tracker 79
4.2 Accounting Configuration Manager 79
4.3 Monitoring Manager 81
5 Functional Validation and Performance Evaluation **81**
5.1 Functional Tests 81
5.2 Performance Micro-Benchmarks 85
5.3 DaCapo Benchmarks 87
6 Conclusion **88**

This chapter describes the issues of memory monitoring in an open middleware for the smart home gateway, and proposes a solution for these issues. We present a memory monitoring system that is mostly transparent to application developers, that provides monitoring reports that are accurate enough and relevant, with respect to the particular composition of the gateway middleware. We illustrate the design of the system in addition to the implementation details of our prototype. Then we evaluate the system on the functional and performance levels, using micro-benchmarks and real life DaCapo applications.

1. Memory Monitoring Issues

The smart home ecosystem, as conceived by the Home Gateway Initiative [59], is based on OSGi and Java, in an effort to support the openness of a multi-tenant execution environment hosting applications collaborating to render services (see Sections 2.2.3 and 6.2.2). All

tenants deploy their bundles on the same execution environment, as shows the Figure 4.1. This sharing by untrusted competing tenants raises the need to “protect the box against badly written bundles” [104, 50]. A *badly written* bundle is a bundle that consumes resources, such as CPU power and memory space and network bandwidth, far above normal expected levels. Therefore, mechanisms that regulate resource consumption at bundle granularity are necessary, and in particular, *resource monitoring* mechanisms. Resource monitoring answers essentially two questions: (1) *counting*, i.e., how much of the resource is consumed?, and (2) *accounting*, i.e., which entity should be charged for using that quantity of the resource?

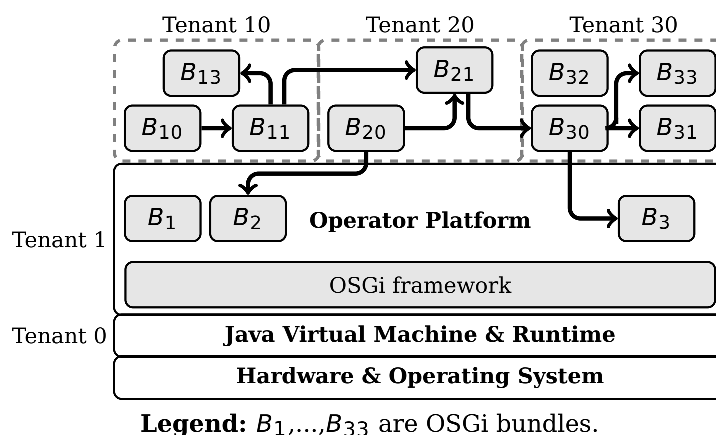


Figure 4.1: Multi-tenant OSGi-based execution environment

Implementing a sufficiently *accurate* monitoring system with a minimal performance overhead is a challenge. The problem is that accurate resource accounting during cross-application interactions requires information related to *business logic* between the caller application and the service being called, which is neither provided by OSGi nor Java. This is why most existing solutions tend to *avoid* the problem instead of solving it (see Section 6.3.4), either by prohibiting direct cross-application communication, or by adopting direct accounting all the time [50], or by adopting indirect accounting all the time [86] (see Section 6.3.3), or by delegating the problem to the developers [23, 73]. All in all, existing OSGi-related resource monitoring approaches do not perform accurate resource accounting when cross-application interactions occur.

Several monitoring tools were developed for Java-based systems, such as A-OSGi [46], JMX [52], JVM-TI [99], the method described by Miettinen et al. [86], and our previous work on *adaptive monitoring* [83]. All these solutions are designed to monitor low granularity Java elements, e.g., threads, classes, objects, methods. The problem with these solutions is that, taken as is, the information they produce is of limited interest in OSGi platforms. This is why we need to *raise the abstraction* to, at least, the primitive deployment unit in OSGi, i.e., the OSGi bundle. On a higher level, information about OSGi applications (set of bundles) and application tenants is also useful in industrial OSGi platform.

2. Our Solution to Memory Monitoring in OSGi

In this chapter, we present an *OSGi-aware memory monitoring system* [10] that is mostly *transparent* to application developers, and that allows *collaboration* between distinct tenants sharing the same OSGi execution environment. The system monitors calls between tenants and provides on-demand snapshots of memory usage statistics for the different tenants.

To solve the problem of resource accounting during interactions between tenants, the monitoring subsystem has predefined *implicit* resource accounting rules that describe correctly most interactions between tenants. For those interactions that are not correctly accounted for, the system allows specifying *explicit accounting rules* in the form of simple configuration files loaded by the monitoring system at JVM start-up. Our prototype requires the resource accounting rules to remain constant during the lifetime of the JVM. At runtime, the monitoring system applies implicit and explicit rules to correctly account for memory used by the bundles in local variables, loaded classes, and created objects. In practice, most bundles do not need to write accounting rules because implicit rules handle their interactions correctly. Bundles that need to write explicit accounting rules are mainly those that generate *asynchronous activity*, such as those that publish events. Examples include the OSGi framework, and bundles exposing data of home sensors in real time. However, most bundles consume services sequentially and provide services only on-demand.

3. Accurate and Relevant Memory Monitoring

In this section, we present the design of a memory monitoring subsystem that is intended to be a part of a more complete system for resource management inside a long running JVM running OSGi. The memory usage information reported by this subsystem would indicate memory leaks and too high memory usage patterns.

Scenario 4.1 (Motivating Scenario: Service Method)**.** Often in practice, a tenant would require the caller of the service it provides to be accounted for resources consumed by that service. For example, a tenant that provides a service `playRingTone()` would require the tenant calling that service to be accounted for memory consumed in order to play the ring tone. No rules should be needed in order to fulfill this accounting, i.e., it should be the default behavior of resource accounting.

Scenario 4.2 (Motivating Scenario: Event Handlers)**.** Less often, a tenant A provides an interface to notify observers about an event. Another tenant B subscribes to the event and expects to be called when the event occurs. When the event occurs, A calls B as contracted. In this case, implicit rules (see the step `3(d)` in Section 3.4) specify that A should be accounted for resources consumed during the call. However, this is unfair given the fact that A calls B only because B asked for that by subscribing to the event. In this case, A needs to write an accounting rule for the notification interface, accounting resources for the called entity (see Section 3.3).

3.1. Assumptions

We assume a number of preconditions on the system where our monitoring subsystem runs.

No Need for Isolation. We do not suppose any form of isolation beyond what is provided by OSGi. Therefore, all tenants are able to communicate via service method calls. This enables different tenants to collaborate to create integrated services and user experience, even when the user is using sensors and actuators and applications from different manufacturers and editors.

Constant Monitoring, Infrequent Reporting It is typical for a long-running system to have a *resource manager* subsystem that periodically requests memory usage statistics from a memory monitoring subsystem. In the smart home gateway, memory usage statistics would be requested once or twice a day in relatively stable configurations, and the period can be as frequent as every hour when hardware or software configurations change. The memory usage statistics enable detecting *abnormal* memory usage situations, e.g., memory leaks. When such abnormal activity is detected, the resource manager subsystem carries out actions to restore the system to a normal state by making memory available again, such as terminating some applications. The resource manager subsystem can be human driven or autonomous.

Our memory monitoring subsystem runs *continuously*, collecting *raw* memory usage data on running applications. This generates a *persistent* overhead that must be kept to a minimum. Furthermore, in order to report relevant memory statistics, raw data need to be *aggregated* and *filtered*, generating an overhead every time a memory monitoring report is requested. We target systems where memory monitoring reports will be requested *sparingly* in time, in order to check for abnormal resource usage. This is the case of the smart home gateway, where memory monitoring reports would be generated from once per hour to once per day. Therefore, we tolerate the aggregation and filtering overhead needed to generate memory monitoring reports, and we rather focus on the persistent overhead caused by continuous monitoring.

Constant Resource Accounting Configuration. In order to simplify our prototype and keep performance overhead acceptable, we require that resource accounting rules and tenants list remains *constant* during the JVM lifetime. This allows performing early calculations in order to accelerate inference of accounting rules.

3.2. Goals

We designed our monitoring subsystem in order to fulfill the following goals.

Detailed Memory Monitoring. Our prototype monitors memory usage of every tenant in *call stack* space and *heap* space. Call stack space is where most methods variables and

parameters are stored. The subsystem reports the number of bytes accounted to each tenant, in the call stacks of all running threads. Heap memory is used to store Java classes and objects, particularly their static fields and object fields. The subsystem reports the number of classes loaded by each tenant and the number of bytes used by those classes. It also reports the number of reachable objects that are accounted to each tenant, in addition to the number of bytes used by those objects.

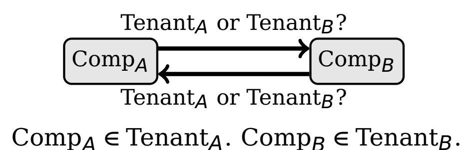
Expressive Resource Accounting Rules. The tenants and the platform operator are required to provide accounting rules that describe how accounting should occur when two tenants interact. We designed a Domain-Specific Language (DSL) to enable expression of resource accounting rules.

Only Specify Special Cases. We want our subsystem to require the smallest configuration possible. The default configuration should work well for most of the cases, and developers and platform administrators should configure and maintain only the special cases of interactions. For this, we armed our subsystem with a list of implicit rules to handle most cases correctly. We also defined a resource accounting algorithm that decides which tenants to account for resources, so that:

- Explicit rules always override implicit rules.
- The order of processing rules is from the most specific, to the most generic.

3.3. Domain-Specific Language for Resource Accounting Rules

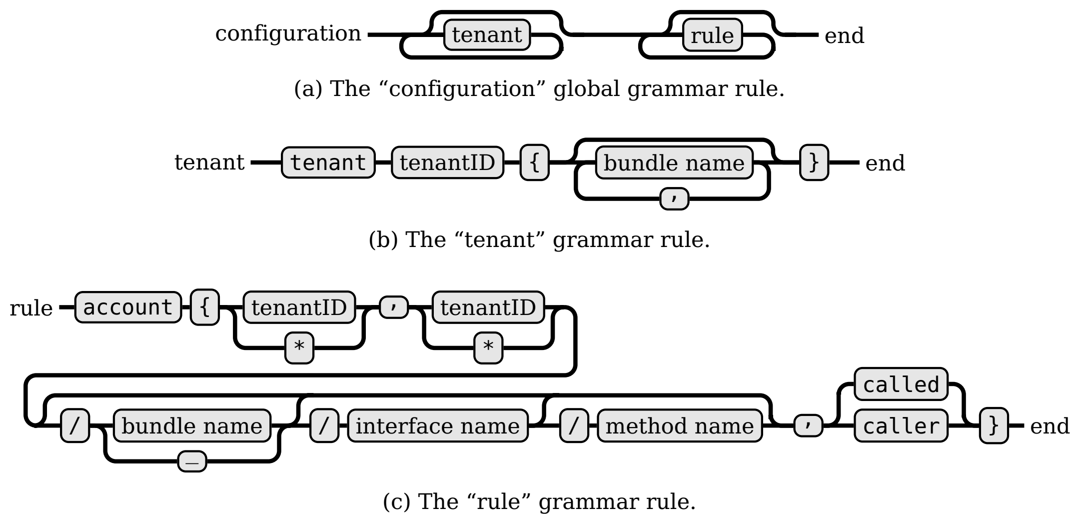
We defined a DSL that enables developers and platform administrators to specify the rules to decide which tenant should be accounted for resources consumed during an interaction, as illustrated in Figure 4.2. Illustrated in Figure 4.3, the DSL allows rules of varying levels of precision, which allows factoring the rules, thus writing and maintaining less of them. It also allows correct handling of all possible cases of bundle interactions.



$Comp_A \in Tenant_A$. $Comp_B \in Tenant_B$.

Figure 4.2: Resource accounting during interaction between two tenants

The DSL describes separately two aspects: a list of tenants, then a list of rules. Each element in the tenants list describes the identifier assigned to a tenant, and the names of bundles it deploys. Each element in the rules list describes a method call between two tenants, and which tenant should be accounted for resources consumed during that call. We reserve `0` as the tenant ID of the Java runtime and the JVM native code, and we reserve `1` as the tenant ID of the platform operator (see Figure 4.1). A rule starts with the caller tenant ID (an integer),



(a) The “configuration” global grammar rule.

(b) The “tenant” grammar rule.

(c) The “rule” grammar rule.

Figure 4.3: DSL of resource accounting configuration

which can be “*” to indicate that any caller tenant matches this rule. Then, the called site is specified, followed by the accounting decision. The called site is (1) a method of (2) an interface defined in (3) a tenant and implemented by (4) a bundle. The four components are optional, and omitting a component matches all possible values, e.g., specifying “_” as the implementation bundle name matches any bundles implementing the specified interface or method. Finally, the accounting decision specifies which tenant is accounted for resources consumed during the call. Note that when the called tenant should be accounted for resources, it is the tenant holding the implementation of the called method that is accounted, not the tenant defining the interface.

```
tenant   1 { org.knopflerfish.framework }
tenant  20 { tests.A                    }
tenant 200 { j3mgr                      }

account { 0, 0/_/java.lang.Runnable/run, called }
account { *, 200/j3mgr/j3.J3Mgr,         caller }
account { *, 20/tests.A/tests.A.A,       caller }
```

Figure 4.4: Sample resource accounting configuration

Figure 4.4 shows an example of resource accounting configuration. It first associates bundles with tenants, then it declares rules, each of which specifies which tenant is accounted for resources consumed during a given method call. So, first it declares the platform operator as `tenant 1`, holding the main `framework` bundle of the Knopflerfish OSGi implementation. It declares two other tenants whose identifiers are 20 and 200, with one bundle for each tenant. Then it declares that a call from the Java runtime to the method `run` of the `java.lang.Runnable` interface implemented by any bundle will be accounted to the tenant of the called bundle, i.e., the tenant of implementation bundle. Next, a call from any tenant to a method of the interface `j3.J3Mgr` implemented in the bundle `j3mgr` of the tenant 200 is accounted to the tenant of the caller. Finally, a call from any tenant to a method of the interface `tests.A.A`

implemented in the bundle `tests.A` of the tenant 20 is accounted to the tenant of the caller.

Back to the Scenario 4.2 previously described, all what the developer needs to write is the following accounting rule:

```
account(42, 42/_/notify.event/happened, called)
```

This rule specifies that a call from the tenant 42 (the tenant ID of A) to the method `happened()` of the interface `notify.event` defined in the tenant 42 (in A) and implemented by any tenant (the "_") is accounted to the called tenant implementing the interface, i.e., the tenant subscribed to the event published by A.

3.4. Resource Accounting Algorithm

This section describes the resource accounting algorithm that decides which tenant is accounted for resources consumed during an interaction.

The list of tenants described in Section 3.3 is stored in the following data structure associating each tenant ID with the list of bundles it is responsible of:

$$map = \{\ldots, (\text{TenantID}_i \rightarrow \{\ldots, \text{BundleName}_j, \ldots\}), \ldots\}$$

The list of resource accounting rules is stored in the following data structure associating accounting rules with decisions:

$$\begin{aligned} map =& \{\ldots, (\text{CallConfig}_i \rightarrow \text{caller}|\text{called}), \ldots\}, \text{where:} \\ \text{CallConfig}_i =& ([\text{CallerTenantID}], \text{CalledSite}), \text{where:} \\ \text{CalledSite} =& (\text{TenantID}, [\text{BundleName}], [\text{InterfaceName}], [\text{MethodName}]) \\ & \text{where: } [x] \text{ means } x \text{ is optional} \end{aligned}$$

Given a call configuration λ_i (a.k.a. CallConfig_i in the map expression above) as an input, the algorithm proceeds as follows:

1. The accounting rules map is searched for an *exact match* for the key λ_i. If that is found, then its associated value indicates explicitly which tenant is accounted for the resource usage, and the process ends.

2. If λ_i is totally *generic*, i.e., if its method name and interface name and bundle name are all missing, then continue to 3, otherwise, continue to 4.

3. No rules are defined for this interaction. Apply the following *implicit rules*:

   (1) If the call is an internal operator platform call, then account resource usage to the caller, i.e., the platform operator.

   (2) If the platform operator is calling a tenant, then account resource usage to the called entity, i.e., the tenant.

(3) If a tenant is calling the platform operator, then account resource usage to the caller, i.e., the tenant.

(4) Otherwise, account resource usage to the caller.

And the process ends here.

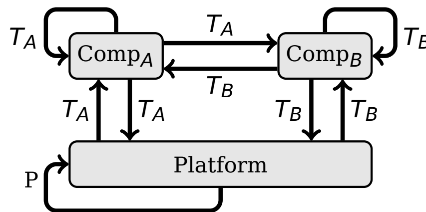
4. Make λ_i more generic, i.e., remove one non-missing piece on information from it, in the following order: method name, then interface name, then bundle name.

   Loop to 1.

The illustrated decision algorithm ensures that:

- Accounting rules order is unimportant, i.e., rules are always matched from the most specific, to the most generic.
- Implicit accounting rules account the platform operator only when the interaction is an internal operator platform call.
- Implicit accounting rules between two tenants accounts the caller.

These accounting decisions are illustrated in Figure 4.5.



T_A, T_B: Tenants. P: Platform. $\text{Comp}_A \in T_A$. $\text{Comp}_B \in T_B$.

Figure 4.5: Accounting decisions made by the accounting algorithm

4. Implementation Details

This section describes our implementation of the design we promote in the previous section. We divided the monitoring subsystem into three major components that run inside the JVM:

- *OSGi state tracker* that acts as a bridge between the JVM subsystems and the OSGi framework.
- *Accounting configuration manager* that parses accounting rules and performs accounting decisions.

- *Monitoring manager* that generates, on-demand, snapshots of detailed memory statistics.

The monitoring subsystem implementation is around 2000 lines of C++ code mostly inside the J3 JVM based on VMKit [51], LLVM [74] and MMTk [20]. The OSGi framework used is Knopflerfish 5.0.

4.1. OSGi State Tracker

In resource accounting rules, each bundle is identified by its name. But, at runtime, each bundle instance is identified by a unique framework-assigned ID that is not reused even if that instance is uninstalled. The link between the bundle name and the bundle ID enables the monitoring subsystem to know which rule to apply when two bundles interact. The OSGi state tracker component makes the links between the static bundle information (e.g., bundle names) given in the resource accounting rules and the dynamic states of OSGi bundles at runtime (e.g., bundle ID, bundle class loaders). It tracks the OSGi states (e.g., `resolved`, `uninstalled`) of bundles installed in the OSGi framework running on top of the JVM, by listening to events of bundle state changes from the JVM and from the OSGi framework.

This component associates bundle identifiers with their respective bundle information. The bundle information includes the bundle name, and its current and previous class loaders. The association map is expressed as:

$$map = \{\ldots, (\text{BundleID}_i \rightarrow \text{BundleInfo}_i), \ldots\}, \text{ where:}$$
$$\text{BundleInfo}_i = (\text{bundleName}, \{\ldots, \text{ClassLoader}_j, \ldots\})$$

In order to discover bundle state changes, this component places two hooks in the Knopflerfish 5.0 OSGi framework, which makes it dependent on that particular OSGi implementation. However, those hooks need no more than 10 lines of code inserted into the framework code. Therefore, the dependency is fairly limited.

This component encapsulates all the logic necessary to interact with the OSGi framework. Therefore, porting the monitoring subsystem to another OSGi framework implementation only requires porting this component. This makes the major part of the subsystem independent of any particular OSGi framework implementation.

4.2. Accounting Configuration Manager

When the JVM starts up, the accounting configuration manager component loads the resource accounting configuration, before loading the OSGi framework code. The configuration is stored in memory in the data structures described in Section 3.4, and it remains constant during the execution of the JVM. We implemented a parser inside this component, in order to load the configuration from any text file (e.g., disk files, names pipes, sockets) that conforms to the DSL described in Section 3.3.

At runtime, each time a bundle calls a method via direct invocation (i.e., `invoke*`) or object construction (i.e., `new`), this component decides which tenant should be accounted for resources consumed during that call. The decision is based on the algorithm described in Section 3.4, which takes into account the resource accounting configuration, default accounting rules, and runtime information. Some runtime information, e.g., states of bundles, is provided by the OSGi state tracker component previously described. The accounting decision concerns solely resources consumed in the thread running the called method, and it remains effective until another method is called in that thread.

For each Java thread, we set a thread-local variable γ_i holding the thread's *currently accounted tenant ID*. Initially, γ_i is set to `0`: the special tenant ID reserved to the Java runtime. Before every method call, the resource accounting algorithm decides which tenant is accounted for the resources consumed during the call. If the algorithm decides that the called tenant shall be accounted for resources, then γ_i is set to the tenant ID of the called method. Otherwise, γ_i remains unchanged. In all cases, the tenant identified by γ_i is accounted for resources consumed by the thread. Most changes needed to track method calls and to invoke the decision algorithm are implemented in the Just-In-Time (JIT) subsystem of the JVM.

The monitoring subsystem adds, in every Java object, a hidden field that holds the tenant ID accounted for the object. Every time the JVM creates a new object, i.e., executes a `new` instruction, it reads γ_i to determine which tenant is accounted for the newly created object. The object is consequently tagged with the tenant ID set in of γ_i.

This mode of operation regarding γ_i ensures that resources are accounted to the caller tenant, unless otherwise specified by an implicit or declared accounting rule. The notion of the caller tenant is transitive. Consider the example call sequence:

$$S_1 = \ldots \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow \ldots$$

where M_i is a method belonging to a tenant T_i, and arrows indicate method calls. In each method call, before entering the called method M_i, the decision algorithm makes the decision D_i, which determines the new value of γ_i. An example of decisions follows:

$D_1 = \ldots \rightarrow M_1$: the **called** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_2 = M_1 \rightarrow M_2$: the **caller** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_3 = M_2 \rightarrow M_3$: the **caller** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_4 = M_3 \rightarrow M_4$: the **called** tenant is accounted. $\Rightarrow \gamma_i = T_4$

This example implies that resources consumed during the execution of M_1, M_2 and M_3 are all accounted to T_1, whereas resources consumed in M_4 are accounted to T_4.

It is worth noticing that γ_i is restored to its previous value in any scenario that makes the method return to one of its callers, e.g., when a method returns, or when an exception is caught outside the method where it is thrown. The previous values of γ_i are stored in the call stack, as hidden local variables.

This frequent execution of the decision algorithm in every method call is the primary source of the permanent overhead added to the normal execution of Java code.

4.3. Monitoring Manager

The monitoring manager component generates, on-demand, snapshots of detailed memory statistics. In order to do so, this component triggers a special garbage collection cycle, during which it scans the object graph and call stacks of running threads, while accumulating statistical counters. This scan is performed by placing hooks in the garbage collector code which scans objects and threads. The counters accumulate the following information, grouped by tenants (see Section 3.2):

- Number of reachable objects and their size.
- Number of loaded classes and their size.
- Amount of used stack space.

It is worth noting that the modifications performed on the garbage collector do not depend on the algorithm used for collection, and do not modify the accessed objects and classes and threads. We only depend on the fact that the garbage collector can perform a collection cycle during which it scans the whole objects graph, and during which all threads are suspended. Otherwise, this component is independent of the garbage collector implementation.

5. Functional Validation and Performance Evaluation

The benchmarks were executed on a computer running the 32-bits version of the Linux 3.12 kernel, on an Intel Xeon CPU running at 2.7 GHz, with 12 megabytes of cache, 12 gigabytes of RAM[1] and 1 terabyte of disk space.

5.1. Functional Tests

In order to ensure that the monitoring subsystem works as intended, we performed functional unit tests. To verify correct accounting, we ask the monitoring system to output detailed accounting calculations.

```
tenant 1  { org.knopflerfish.framework }
tenant 50 { tests.A }
tenant 60 { tests.B }
tenant 70 { tests.C }
tenant 80 { tests.D }
tenant 90 { tests.E }
```

Figure 4.6: Tenants declarations in functional tests.

[1]The kernel allowed addressing the whole RAM via Physical Address Extensions.

5.1.1. Operator Platform Internal Accounting

A part of the detailed accounting calculation is shown in Figure 4.7, consisting of a thread call stack prefixed with monitoring information, i.e., `T`: identifier of the tenant that is accounted for resources consumed during the method frame, and `Size`: the total size, in bytes, of the method frame. For example, in Figure 4.7, 1456 bytes of stack space are accounted to the Java runtime (tenant 0), and 1120 bytes of stack space are accounted to the operator platform (tenant 1). The package `java.lang` belongs to the Java runtime. The line `1` in Figure 4.6 states that the package `org.knopflerfish.framework` belongs to the operator platform.

```
T Size MethodFrameName
1 480  java.lang.VMObject.wait
1 128  java.lang.Object.wait
1 240  org.knopflerfish.framework.Queue.removeWait
1 272  org.knopflerfish.framework.StartLevelController.run
0 96   java.lang.Thread.run
0 304  java.lang.VMThread.run
0 1056 <native>
```

Legend:
`T`: Currently accounted tenant ID, i.e., γ_i.
`Size`: Stack size, in bytes, of the method frame.

Figure 4.7: Detailed memory accounting calculations on a call stack of a thread internal to the OSGi framework.

The choice of which tenant is accounted for a particular method frame is based on the algorithm described in Section 3.4. As specified in Section 4.2, γ_i denotes the *currently accounted tenant ID*. In fact, γ_i corresponds to `T` in Figures 4.7 and 4.8. Initially, $\gamma_i \leftarrow 0$: the tenant ID of the Java runtime. If the algorithm decides that the *called* tenant shall be accounted for resources, then γ_i is set to the tenant ID of the called method. Otherwise, γ_i remains unchanged. In all cases, the tenant identified by γ_i is accounted for resources consumed by the method frame.

In Figure 4.7, the `<native>` frame is the first code to run in the thread, consisting of operating system thread initialization routine and JVM routines that prepare for Java code execution. Initially, the native frame is accounted to the Java runtime, i.e., the tenant 0 ($\gamma_i \leftarrow 0$). The native code calls the `java.lang.VMThread.run()` method implemented in Java runtime, which makes it an internal Java runtime call. Step 3(a) in the algorithm described in Section 3.4 accounts the call to the caller, i.e., tenant 0 ($\gamma_i = 0$) which is the Java runtime. The same accounting goes for the next method frame.

Next, the Java runtime calls the method `run()` of the class `org.knopflerfish.framework.StartLevelController` defined in the operator platform (tenant 1), and implementing the interface `java.lang.Runnable` defined in the Java runtime (tenant 0). This call matches the predefined explicit rule: `account {0, 0/_/java.lang.Runnable/run, called}`. This is why the called, i.e., the operator platform (tenant 1) is accounted for the call, and $\gamma_i \leftarrow 1$. The next call is internal to the tenant 1, so the caller, i.e., tenant 1 ($\gamma_i = 1$) is accounted for it.

Later, the operator platform calls the Java runtime method `java.lang.Object.wait()`. The step 3(c) in the algorithm implies that the caller, i.e., tenant 1 ($\gamma_i = 1$), is accounted for the call. The rest of the calls are internal Java runtime calls, for which the caller tenant, i.e., the tenant 1 ($\gamma_i = 1$) is accounted for the resources consumed during the calls.

5.1.2. Implicit and Explicit Accounting

To test monitoring of other types of interactions between different tenants, we define five components `tests.A` through `tests.E`, each belonging to a tenant, as declared in Figure 4.6 and illustrated in Figure 4.9. Then we examine a thread that executes methods from these different components. We execute the thread without defining explicit rules (except a few predefined explicit rules, such as the one shown in the previous subsection), and we show the accounting results in Figure 4.8a, then we execute it after adding one explicit rule, and we show the accounting results in Figure 4.8b.

The Figure 4.8 describes the call stack of a thread that is created by the operator platform in order to start and stop components. A generic description of the thread activity goes as follows. The thread begins running operator platform code in method `org.knopflerfish.framework.BundleThread.run()`. Later, it starts the component `tests.A` by calling the *event handler* `tests.A.Activator.start()` that indirectly calls the method `e()` of the class `tests.B.C` which implements the interface `tests.C.C`, as shown in Figure 4.9. `tests.B.C` performs some processing before notifying the component `tests.D` of an event by calling the event handler method `handler()` of the class `tests.D.D` which implements the interface `tests.E.E`. Next, `handler()` calls a Java runtime service to sleep.

In Figure 4.8a, and starting from the first frame executed by the thread, i.e., `<native>`, the three following frames are accounted as previously described for Figure 4.7. Then, the operator platform (tenant 1) calls method `start()` of the class `tests.A.Activator` (tenant 50) which implements the operator platform interface `org.osgi.framework.BundleActivator`. Step 3(b) of the algorithm described in Section 3.4 states that the called tenant, i.e., tenant 50, is accounted for resources consumed in the method frame, which sets γ_i to 50. In the next five call frames (up to `tests.D.D.sleep()`), step 3(d) of the algorithm states that the caller tenant, i.e., tenant 50 ($\gamma_i = 50$), is accounted for resources consumed in method frames. Then, step 3(c) of the algorithm accounts consumed resources to the caller, i.e., tenant 50 ($\gamma_i = 50$). The remaining method frames are internal Java runtime calls, so step 3(a) of the algorithm accounts consumed resources to the caller, i.e., tenant 50 ($\gamma_i = 50$).

The problem observed in Figure 4.8a is that tenant 50 is held accounted for resources consumed during execution of the event handler method `handler()` and all the methods the latter executes, even though `handler()` is being executed only because the component `tests.D` requested it. Due to this issue, the monitoring system inaccurately accounts to tenant 50 the following amounts of memory: 1296 bytes of stack space, and 50 000 objects totaling 600 004 bytes of heap space, even though that memory was consumed by the event handler. To correct this inaccuracy, we add the following explicit rule: `account {*, 90/_/tests.E.E, called}`, which indicates that `tests.E.E` is an event handling interface (for any caller tenant, for any implementation component, and for any method in the interface).

```
T  Size MethodFrameName
50 480  java.lang.VMObject.wait
50 64   java.lang.Object.wait
50 256  java.lang.VMThread.sleep
50 112  java.lang.Thread.sleep
50 224  tests.D.D.sleep
50 160  tests.D.D.handler
50 144  tests.B.C.someProcessing
50 256  tests.B.C.e
50 144  tests.A.Activator.heavyInitialization
50 128  tests.A.Activator.start
1  384  org.knopflerfish.framework.Bundle.start0
1  272  ...flerfish.framework.BundleThread.run
0  272  java.lang.VMThread.run
0  1056 <native>

T  Stack Objects Classes HS   HV
0  14816 7964    129     1600 167336
1  4176  35908   154     1597 1107636
50 1968  55525   16      313  666500
60 0     15      13      104  316
80 0     19      14      104  384
```

(a) Only implicit accounting used.

```
T  Size MethodFrameName
80 480  java.lang.VMObject.wait
80 64   java.lang.Object.wait
80 256  java.lang.VMThread.sleep
80 112  java.lang.Thread.sleep
80 224  tests.D.D.sleep
80 160  tests.D.D.handler
50 144  tests.B.C.someProcessing
50 256  tests.B.C.e
50 144  tests.A.Activator.heavyInitialization
50 128  tests.A.Activator.start
1  384  org.knopflerfish.framework.Bundle.start0
1  272  ...flerfish.framework.BundleThread.run
0  272  java.lang.VMThread.run
0  1056 <native>

T  Stack Objects Classes HS   HV
0  14816 7964    129     1600 167336
1  4176  35906   154     1597 1107592
50 672   5525    16      313  66496
60 0     15      13      104  316
80 1296  50019   15      168  600388
```

(b) Implicit and explicit accounting used.

Legend:
T: Currently accounted tenant ID, i.e., γ_i.
Size: Stack size, in bytes, of the method frame.
HS: Heap size, in bytes, of Static data, e.g., class fields, etc.
HV: Heap size, in bytes, of Virtual data, e.g., object fields, etc.

Figure 4.8: Detailed memory accounting calculations on a thread call stack.

Figure 4.8b shows the accounting results after adding the explicit rule and restarting the JVM. Adding the explicit rule effectively solves the problem by accounting the resource consumed by `tests.D.D.handler()` to the called tenant, i.e., tenant 80, which accordingly sets γ_i to 80. This boost in accuracy is also visible in the final statistics, as the monitoring system now accounts tenant 80 for the stack and heap memory that was wrongly accounted to tenant 50.

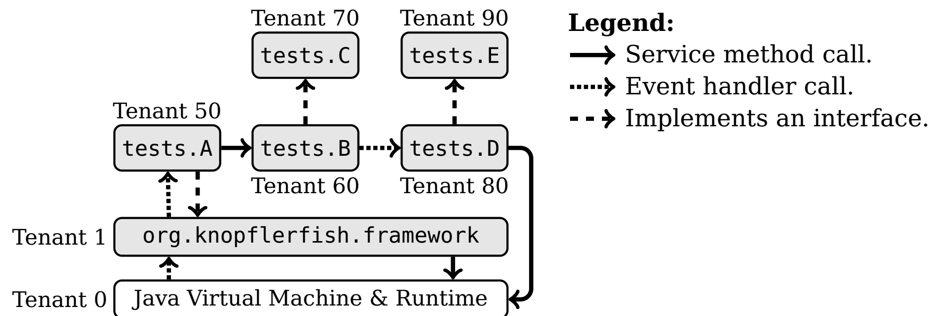


Figure 4.9: Functional tests components.

5.2. Performance Micro-Benchmarks

In order to show the details of the performance overhead of monitoring, we made four versions of the monitoring subsystem implementation, as follows:

1. *Zero implementation*, i.e., a JVM without monitoring.

2. *Object tag added*, i.e., the only thing modified in the JVM is adding 4 bytes to every Java object to hold the ID of the tenant accounted for the object.

3. *No memory access around `invoke`*, i.e., the whole monitoring subsystem is implemented in the JVM, except that we do not generate code to save and restore the thread's currently accounted tenant ID (6 `store` and 2 `load` instructions, see γ_i in Section 4.2) around the `invoke` bytecode instruction.

4. *Complete implementation* of the monitoring subsystem, i.e., the same as the previous version, in addition to 6 `store` and 2 `load` instructions generated around every `invoke` bytecode instruction.

We also performed some micro-benchmarks that test very specific aspects of execution.

5.2.1. Method Call Micro-Benchmark

This benchmark calls many times one Java method that does nothing special. We are only interested in the overhead of calling a method, with and without the monitoring subsystem.

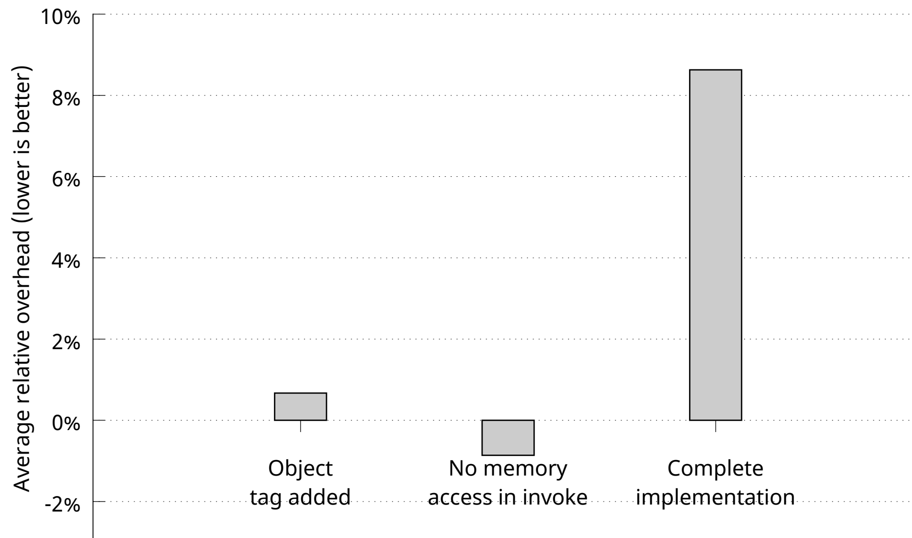


Figure 4.10: Execution overhead of the method call micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the "Zero implementation". Overhead is an average of 10 runs.

Figure 4.10 illustrates the results of this benchmark. A comparison of the results of running this micro-benchmark on the "No memory access around `invoke`" version and the "Complete implementation" version reveals that method invocation performance decreases by 9%. We did not observe a significant additional loss in performance with other versions, i.e., less than 1%. Therefore, method invocation performance is only affected by the additional memory access performed by the monitoring subsystem around every `invoke` instruction.

5.2.2. Object Creation Micro-Benchmark

This benchmark performs numerous creations of objects of the class `java.lang.Integer`. The objects themselves are relatively small. We are only interested in the overhead of creating an object, with and without the monitoring system.

Figure 4.11 illustrates the results of this benchmark. A comparison of the results of running this micro-benchmark on the "Zero implementation" version and the "Object tag added" version reveals that object creation performance decreases by 44%. We did not observe a significant additional loss in performance with other versions, i.e., less than 7%. Therefore, object creation performance is mostly affected by the addition of 4 bytes to every Java object.

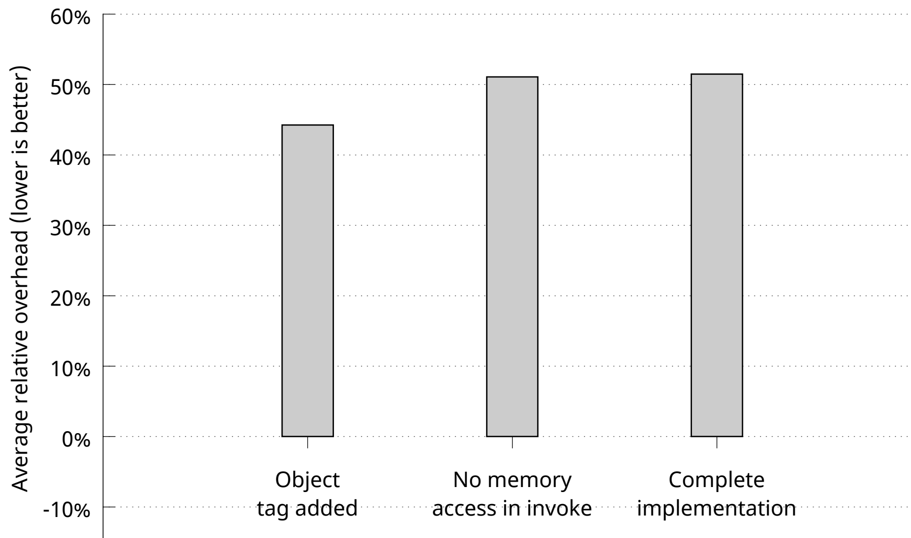


Figure 4.11: Execution overhead of the small objects micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the “Zero implementation”. Overhead is an average of 10 runs.

5.3. DaCapo Benchmarks

In order to measure the overhead of the monitoring subsystem on real life Java applications, we performed the DaCapo 2006 benchmarks[2] [21]. Benchmark results in Figure 4.12 show that the monitoring subsystem overhead is always below 46% for real life DaCapo applications.

It is worth noticing that the monitoring overhead in Figure 4.12 stays below 6% for all DaCapo applications (except `hsqldb`) when we run DaCapo benchmarks using the “No memory access around `invoke`” version. This suggests that accessing memory (`load`, `store`) in every `invoke` bytecode instruction is an expensive operation.

We also notice that the performance of the application `hsqldb` drops by 33% when we run it using the “Object tag added” version. This suggests that this application creates many objects and would suffer from changes to the object structure. The performance of this application drops by another 13% between the “Object tag added” version and the “Complete implementation”. This further indicates that `hsqldb` performs much more objects creations than method invocations.

[2] www.dacapobench.org

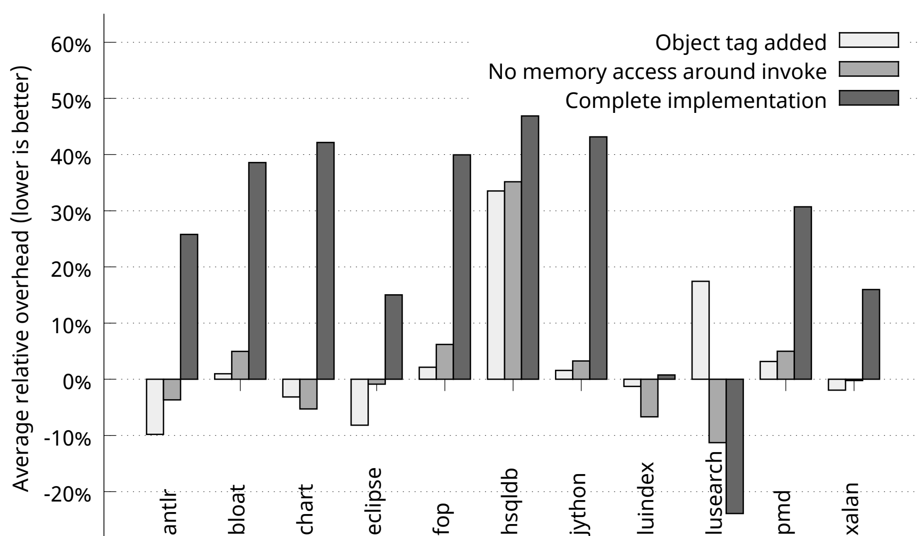


Figure 4.12: Execution overhead of DaCapo 2006 benchmark applications when run on partial implementations of the monitoring subsystem, compared to the original JVM. Overhead is an average of 10 runs.

6. Conclusion

OSGi gets increasingly adopted in the smart home as a framework to host service-oriented applications delivered by multiple untrusted tenants. This raises the need for monitoring systems that can provide useful accurate information about OSGi bundles and applications.

In this chapter, we present a monitoring system that monitors memory usage at the bundle granularity, without requiring isolation of distinct tenants. Our system is far less intrusive than existing systems and methods, and it does not assume trusted tenants. It is based on a list of accounting rules that enables correct resource accounting in all cases. The monitoring system is modular and mostly independent of the implementations of the OSGi framework and the garbage collector.

Based on DaCapo benchmarks, we showed that the overhead of our system was below 46% for real life Java applications. This overhead is acceptable in development and testing time, and it is tolerable in slow pace applications which are frequent in the Smart Home. Our thorough investigation of performance overhead showed the specific aspects of monitoring that caused the most overhead, and the types of applications that would suffer the most from memory monitoring.

Conclusion

Throughout this thesis, we described our work on managing resource sharing conflicts in the smart home environment. We began our quest by describing the smart home hardware environment, in order to synthesize its main properties, i.e., openness, dynamicity, rapid development, heterogeneity, distributed and embedded aspects. We investigated each property to reveal the challenges it entails, then we focused on each of those challenges, looking for existing work tackling them. Given the particular smart home properties, we argued that the smart home gateway needs to embed *component-based* [38] and *service-oriented* [103] platform and applications. This led us to consider component models such as Fractal [29] and OSGi [127], and service-oriented platforms such as OSGi Declarative Services [126].

In order to manage resource sharing conflicts, one needs to *discover* them, and to *prevent* them as far as possible, and to *resolve* them when prevention is out of reach. Our first contribution, called Jasmin [4], is an effort to prevent resource sharing conflicts in the smart home gateway. Jasmin is a middleware for development, deployment, isolation and administration of component-based and service-oriented applications targeted at embedded systems. The Jasmin prototype is based on the MIND [92] implementation of the Fractal component model. Jasmin provides multiple levels of *application isolation*, from weak design-level separation, to strong isolation based on the Linux containers [77] technology that provides an excellent trade-off between isolation and performance overhead. Jasmin implements a service-oriented architecture and extends it to isolated applications by defining a *service registry coordinator* that enables service dependency resolution independently of the service location. Furthermore, Jasmin enables *transparent communication* between isolated applications by automatically loading and binding of service proxies. These service proxies are based on a cross-container communication mechanism implemented by Jasmin, which is 60% faster than machine-local communications via a standard Linux RPC [118] implementation. As part of the evaluation, we describe the effort required to port a legacy multimedia application to the Jasmin middleware. We show that the porting effort is relatively low, and that it can be performed progressively. We also show that the performances of Jasmin applications are almost identical to the performances of legacy applications.

Given the properties of the smart home, we suggested running a *multi-tenant* Java virtual machine on the smart home gateway, which would host multiple smart home applications. Being a hardware-constrained long-running system, the smart home gateway is vulnerable to *stale references*, which cause considerable memory leaks, consequently increasing the risks of memory conflicts in the gateway and of state inconsistencies in the devices controlled by the gateway. Our second contribution, called Incinerator [9], resolves the problem of stale references in Java by detecting and eliminating them, with less than 4% CPU overhead. Compared to the state-of-the-art stale reference detection tool, Service Coroner [47], Incinerator detects more stale references, and goes further by eliminating

them. Incinerator also helped us contribute back to the community by discovering and resolving a stale reference bug [8] in the Knopflerfish [81] open source OSGi framework implementation.

Due to the limited amount of memory available in the smart home gateway, memory sharing conflicts are a significant issue in the gateway, especially when running a multi-tenant Java virtual machine. The problem of memory monitoring naturally pops up in this context, which is what our third contribution tackles: memory monitoring in OSGi-based platforms. We propose an OSGi-aware monitoring system [10] that is mostly transparent to application developers, and which allows collaboration between the different service providers sharing the same OSGi execution environment. The system monitors calls between applications and provides on-demand snapshots of memory usage statistics for the different service providers, at an overhead lower than 46%. The reported data is accurate enough because the system switches between direct and indirect accounting depending on the context. This switching is based on an algorithm that recognizes most call contexts and accurately accounts for the resources consumed during the call. The algorithm decisions can be overridden by the platform administrator via a set of simple rules that are fed to the monitoring system. The rules are written in a simple and versatile Domain-Specific Language defined for the case.

Future Work

We believe that management of resource conflicts in the smart home should be the responsibility of multiple layers of software and approaches. Given the smart home properties, we think that security should be discussed during the first stages of development of platforms and applications targeted at the smart home, as security often have system-wide implications. Security is a *design decision*, not an *option*.

Very close isolation approaches should be analyzed further, such as containers [77, 95, 106, 70, 121], exokernels [45, 2, 105, 16, 66], sandboxes [132, 53, 135], etc. Because relatively few smart home applications exist, even backward-incompatible solutions can be attempted, such as operating systems and middlewares based on programming language safety [60, 12, 19].

In order to resolve resource conflicts and regulate resource usage, one might attempt to migrate applications between the different devices in the smart home [44]. This, not only helps balance resource usage, but also reduces electrical consumption when very few resources are needed, by migrating applications into the gateway and “hibernating” other devices.

Jasmin exposes an administration service to administration agents which are currently exclusively local. Remote administration is becoming a strong requirement in the smart home. Different remote administration agents could be built, implementing the protocols CWMP, a.k.a., TR-069 [27], and UPnP-DM [129]. Currently, isolation container implementation is dependent on the Linux kernel. However, the distributed Jasmin architecture can be adapted to support other operating system containers, such as FreeBSD Jails [106] and Solaris Zones [95]. Developers should become able to control or assist dependency resolution performed by Jasmin. This could be done adding properties and filters to provided or required interfaces. The extensibility features of the MIND tool-chain are appropriate to add this support. This might also become handy to avoid circular dependency loops between applications.

The prototype of Incinerator defines stale references based on the OSGi component model. Porting Incinerator to the component model of OSGi Declarative Services should help detect and eliminate stale references to Declarative Services components, consequently making the component model more robust.

The relevant information reported by the monitoring system can become an accurate input to an *autonomous resource manager* [46, 83], enabling the latter to automatically detect resource conflicts, as a first step toward resolving them. We are looking for ways to make the list of accounting rules *dynamic* while keeping the overhead acceptable. Making the explicit rules further more generic, for example based on *regular expressions*, would allow factoring of many explicit rules, but can reduce performance of matching against rules, so we are still unsure if more expressiveness would be worth the performance loss. We equally long for monitoring other relevant resources, e.g., CPU usage, disk usage, and network bandwidth.

Appendices

Appendix A.

Résumé de Thèse

Ce chapitre présente un résumé de cette thèse de doctorat en Français.

1. Introduction

Nos maisons sont pleines d'appareils électroniques qui assistent notre vie de diverses façons. Ces appareils deviennent de plus en plus intelligents, c'est-à-dire, ils sont constamment équipés de plus de puissance de traitement, plus de capteurs et d'actionneurs et une meilleure connectivité aux appareils qui l'entourent et à Internet. Ces appareils intelligents sont de plus en plus populaires à la maison grâce aux services avancés qu'ils fournissent à un coût abordable. Les services incluent le divertissement, la sécurité de la maison, l'efficacité énergétique, les soins de santé, le bien-être, confort, et le partage de contenu. Grâce à ces appareils, nous devenons les habitants des maisons intelligentes.

La maison intelligente offre de nouvelles opportunités aux différents fournisseurs de services, qui désirent développer et déployer rapidement des services qui profitent de, non seulement les dispositifs présents à la maison, mais aussi les services offerts par d'autres fournisseurs. Pour permettre la cohabitation de différents fournisseurs de services, les opérateurs de maison intelligente conçoivent une passerelle qui héberge des applications fournies par différents acteurs [59, 112]. La passerelle fournit des services de base aux applications hébergées, y compris le déploiement, la connectivité, le stockage de données, et la découverte de service. Afin de permettre le partage des appareils et des services entre les applications hébergées, les opérateurs de maisons intelligentes standardisent une interface qui est utilisée pour fournir et accéder aux services fournis par les appareils intelligents et les fournisseurs. Par exemple, un fournisseur de services peut utiliser l'interface standard pour commander un appareil fabriqué par un autre fournisseur de service qui est conforme à la même interface.

Les applications s'exécutant sur la passerelle domotique commandent des actionneurs et des appareils à l'intérieur de la maison, et certains de ces appareils gèrent la sécurité et la santé des habitants, tels que les détecteurs de gaz, les climatiseurs, les alarmes de portes, etc. Par conséquent, le redémarrage abrupt de la passerelle peut être dangereux. En fait, la passerelle domotique est une plate-forme s'exécutant pendant de longues durées et qui doit être hautement disponible et suffisamment robuste pour gérer les erreurs logicielles sans avoir à redémarrer abruptement. Cette nature d'exécution de longue durée est la raison pour laquelle la passerelle domotique prend en charge le remplacement à chaud des applications, c'est-à-dire, le démarrage et l'arrêt et la mise à jour des applications à la volée, sans avoir

besoin de redémarrer tout l'environnement. Pour implémenter le remplacement à chaud, les applications hébergées sont fabriqués à base de composants qui peuvent être démarrés, arrêtés et remplacés individuellement.

1.1. Problématique

Partager les ressources de la maison intelligente entre les fournisseurs de services est nécessaire pour fournir des services riches et intégrés à l'utilisateur final, tout en gardant le choix à l'utilisateur de mélanger librement des appareils intelligents et des services provenant de différents fabricants de matériel et des éditeurs de logiciels. Toutefois, le partage des ressources pose naturellement le risque de conflits de ressources entre les fournisseurs de services, tels que la privation de ressources, l'accès illégal aux ressources, et l'utilisation abusive des ressources. Ces conflits menacent la confidentialité, ralentissent les performances de la passerelle, et privent le partage correct des ressources disponibles.

Dans cette thèse, nous étudions le problème des conflits de partage des ressources dans la passerelle domotique vu de différentes perspectives. Nous confrontons la question de prévenir les conflits de ressources de se produire en premier lieu. Mais puisque nous ne pouvons pas éviter tous les conflits de partage de ressources, nous nous concentrons également sur la résolution des conflits quand ils se produisent ou quand ils sont découverts. Découvrir les conflits de ressources requiert des approches de surveillance des ressources, qui sont notre troisième préoccupation.

Les applications déployées par les fournisseurs de services peuvent contenir des bugs. Ces bugs peuvent entraîner l'injection d'un comportement malicieux au sein des applications. Ces risques sont les raisons pour lesquelles nous ne faisons pas confiance aux applications développées par les fournisseurs de services [79], ce qui explique pourquoi les solutions que nous présentons ont tendance à éviter, autant que possible, de déléguer des responsabilités supplémentaires aux développeurs d'applications.

Pour résumer, nous argumentons que la passerelle domotique a besoin d'un environnement logiciel ouvert qui offre les fonctionnalités suivantes:

1. Empêche de ressources autant que possible, en utilisant différentes formes d'isolation et de protection.

2. Surveille les conflits de ressources qui peuvent encore se produire, et quantifie l'utilisation des ressources aussi précisément que nécessaire.

3. Résout les conflits de partage de ressources quand ils se produisent, et les rend visibles aux développeurs d'applications ou aux administrateurs système.

1.2. Contributions

Dans cette thèse, nous présentons trois contributions.

Nous avons commencé par étudier les moyens de prévenir les conflits de partage de ressources entre les applications natives s'exécutant dans la passerelle domotique. La manière la plus évidente d'éviter les conflits est l'isolation. Donc nous voulions obtenir la meilleure isolation possible, mais sans nuire aux performances. Parmi les différents mécanismes d'isolation disponibles, les technologies de conteneurs sont le meilleur choix accomplissant nos besoins. Les conteneurs Linux isolent la mémoire des applications, les communications réseau, les fichiers, les utilisateurs, etc. Nous avons créé Jasmin: un environnement d'exécution qui isole les applications natives à base de composants et orientées services à l'intérieur des conteneurs implémentés par le noyau Linux. Jasmin [4] configure et déploie des conteneurs et exécute les applications à l'intérieur d'eux, tout en contrôlant les cycles de vie des applications. Jasmin résout aussi des services entre les conteneurs et offre des mécanismes transparents d'invocation de services.

Certains conflits de partage de ressources ne peuvent pas être évités à cause de l'environnement logiciel très dynamique envisagé pour la maison intelligente, et à cause de des divers appareils présents à la maison. C'est pourquoi nous explorons également des approches pour résoudre les conflits entre les applications s'exécutant dans la passerelle domotique. Un des problèmes que nous avons abordés est le problème de références obsolètes, qui est commun dans les plate-formes qui supportent le remplacement à chaud. Le remplacement à chaud permet de mettre à jour ou de désinstaller des applications sans avoir à redémarrer la plate-forme. Dans les situations normales, quand une application est désinstallée, toutes les autres applications suppriment leurs références à celle-ci, afin de permettre à la passerelle d'enlever l'application désinstallée de la mémoire. Cependant, si une application défaillante continue à garder une référence à l'application désinstallée, alors cette référence est dite référence obsolète. La référence obsolète force la passerelle de garder l'application désinstallée en mémoire, entraînant ainsi une fuite de mémoire importante. Si l'application défaillante essaie d'utiliser l'application désinstallée via sa référence obsolète, les résultats sont indéfinis, et les états des applications en cours d'exécution peuvent devenir incohérents, car l'application désinstallée ne s'attend pas à être invoquée après avoir exécuté ses routines de terminaison lors de sa désinstallation. Si l'application désinstallée contrôlait un actionneur, l'incohérence d'état peut endommager le matériel contrôlé, ce qui menace la sûreté des habitants de la maison intelligente. Pour résoudre ce problème, nous avons créé Incinerator, un système qui identifie les références obsolètes et les supprime. Après le remplacement à chaud d'une application, Incinérateur examine toutes les références dans la passerelle, à la recherche de références obsolètes. Quand une référence obsolète est trouvée, Incinérateur la supprime, et interdit l'application défaillante d'utiliser cette référence à l'avenir, et exécute le nettoyage qui aurait dû être fait par l'application défaillante. En découvrant les références obsolètes, Incinérateur aide les développeurs à déboguer ce problème qui est difficile à percevoir. En supprimant les références obsolètes, Incinérateur non seulement réduit le risque d'incohérence d'état, mais permet également d'éviter la fuite de mémoire causée par les références obsolètes, permettant ainsi à la passerelle de continuer à travailler sans manquer de mémoire. Le prototype de l'incinérateur a été testé avec Knopflerfish [81], l'un des principaux implémentations à source ouverte de OSGi [127]. Grâce à Incinerator, nous avons découvert et corrigé un bug de référence obsolète [8] dans Knopflerfish.

1.3. Structure de Document de Thèse

Cette thèse est structurée en quatre chapitres principaux. Le premier chapitre commence par explorer l'état de l'art pour décrire l'écosystème existant de la maison intelligente, en focalisation sur les propriétés particulières de l'environnement, et les défis qu'elles induisent. Nous détaillons ces défis afin d'illustrer les verrous conceptuels et techniques qui doivent encore être résolus ou dont les solutions existantes devraient être améliorées. Nous illustrons pourquoi la dynamicité de la maison intelligente nécessite le remplacement à chaud des applications. Ensuite, nous discutons de la nature distribuée de la maison intelligente et de son effet sur la communication entre applications. Nous argumentons que les fournisseurs de services ont besoin de développer rapidement des applications pour la maison intelligente, et nous montrons comment le modèle à composant peut aider à atteindre cet objectif. Par ailleurs, nous montrons comment l'architecture orientée service permet de simplifier l'hétérogénéité de la maison intelligente. Enfin, nous illustrons les concepts et les efforts requis pour supporter la nature ouverte et embarquée de la passerelle domotique.

Le deuxième chapitre décrit notre première contribution, c'est-à-dire Jasmin. Il décrit d'abord les deux architectures de Jasmin qui pourraient être utilisées dans différents types de passerelles domotiques. Ensuite, nous décrivons le support d'isolation multi-niveau fourni par Jasmin et basé sur les processus et sur les conteneurs Linux. Ensuite, nous discutons le mécanisme transparent et rapide fourni par Jasmin pour la communication entre applications, et comment il contribue à fournir une plate-forme orientée services distribuée. Nous avons enfin évalué les performances et les fonctionnalités de Jasmin à l'aide de micro-tests, et nous discutons de l'effort nécessaire pour porter les applications existantes vers Jasmin à l'aide d'une application domotique typique.

Dans le troisième chapitre, nous présentons Incinerator, la deuxième contribution. Nous rappelons que le bug de des références obsolètes en Java introduites au premier chapitre, puis nous expliquons comment Incinerator détecte et résout ce problème. Ensuite, nous montrons les domaines qui nécessitaient des modifications afin de mettre en œuvre Incinerator. Nous évaluons ensuite les fonctionnalités d'Incinerator à l'aide de dix scénarios où des références deviennent obsolète. Nous démontrons également que les références obsolètes peuvent mettre en danger les habitants de la maison intelligente si une application défaillante contrôle des appareils domotiques critiques. Enfin, nous évaluons les performances de Incinerator en utilisant la suite de tests DaCapo.

Notre troisième contribution est présentée dans le quatrième chapitre, où nous commençons par décrire le problème de la surveillance précise et pertinente de mémoire dans une plate-forme OSGi ouverte à de multiples fournisseurs de services non fiables. Nous discutons de nos objectifs conceptuels et nous décrivons l'algorithme que nous avons défini pour fournir un système de surveillance de mémoire précis et pertinent pour la passerelle domotique. Nous décrivons également l'étendue de l'implémentation de notre système, avant d'évaluer les fonctionnalités et performances du système à l'aide de micro-tests et en utilisant la suite de tests DaCapo.

Nous concluons notre thèse en reprenant les problèmes que nous abordons et les contributions que nous proposons, et en suggérant de nouvelles mesures à prendre et des chemins qui nécessitent encore de l'exploration.

2. État de l'Art

Nous explorons les défis de la maison intelligente (voir Figure 1.1), orientés par ses propriétés particulières: ouvert, dynamique, en développement rapide, hétérogène, distribué et embarqué. Chaque propriété génère un ensemble de défis, et la composition de ces propriétés nécessite aussi des compromis à faire, et des solutions personnalisées à étudier.

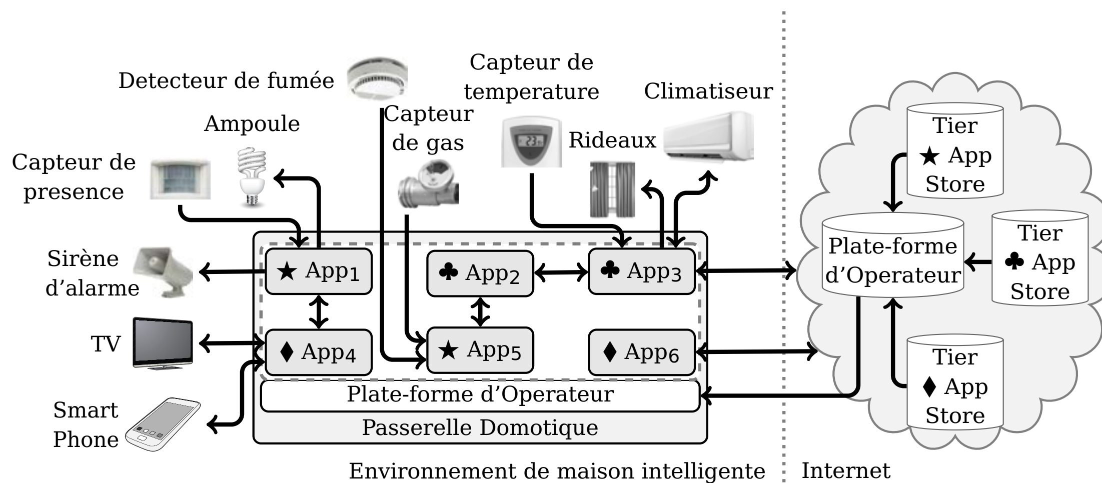


Figure 1.1: Structure domotique. Les communications entre la «Plate-forme d'Operateur» et App_1,...,App_6 sont omis pour simplifier la figure.

L'aspect dynamique de la maison intelligente implique le besoin pour le remplacement à chaud des applications, c'est-à-dire, le chargement, le démarrage, l'arrêt, la mise à jour, et le déchargement des applications en cours d'exécution sans avoir à redémarrer la plate-forme logicielle. Nous avons étudié les mécanismes communs pour le remplacement à chaud, non seulement du code natif [63, 85, 128], mais aussi du byte code Java [76]. Ensuite, nous avons décrit les conditions délicates nécessaires pour décharger une application Java chargé avec un chargeur de classes Java dédié, ce qui est une technique courante dans plusieurs machines virtuelles Java multi-locataires. Ceci a dévoilé le risque impliqué par les références obsolètes Java concernant les systèmes s'exécutant pour de longues durées et à ressources limitées, tels que la passerelle domotique. Le Chapitre 3 présente notre solution proposée pour détecter et éliminer les références obsolètes Java avec un coût très faible.

Parce qu'il est difficile de partager la mémoire dans un environnement distribué, nous avons brièvement décrit comment les applications peuvent interagir, à la fois quand elles peuvent partager de la mémoire, et quand elles ne le peuvent pas. Nous avons décrit donc les appels aux procédures locales [7], et les appels aux procédure distantes [133, 75, 102], impliquant des routines de sérialisation coûteuses. La baisse en performances de communication induite par les appels aux procédures distantes est une des raisons qui encouragent l'exécution de plusieurs applications dans le même espace d'adressage mémoire tant que possible, en particulier lorsque ces applications communiquent fréquemment.

Afin de développer rapidement des applications qui tirent parti de la richesse des équipements,

de capteurs et d'actionneurs disponibles dans la maison intelligente, nous avons proposé de développer des applications basées sur des composants réutilisables, ce qui nous a conduit à décrire un modèle de composant générique (voir Figure 1.2). Ensuite, nous avons illustré des exemples de modèles à composants [38, 122] et des implémentations de modèles à composants qui conviennent pour les environnements embarqués contraints et compatibles avec les autres propriétés de la maison intelligente, par exemple, Fractal[1] [29, 68, 28], MIND[2] [92], OSGi[1,2] [127, 26], OSGi Declarative Services[1,2].

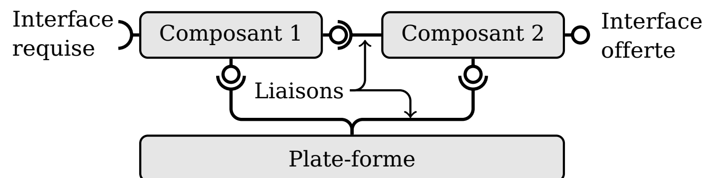


Figure 1.2: Un système à base de composants.

Pour faire face à l'hétérogénéité de la maison intelligente, nous avons proposé l'utilisation de l'architecture orientée services [103] pour la conception de la plate-forme et des applications de maison intelligente. Dans ce but, nous avons décrit d'abord les éléments clés de l'architecture orientée services, avec quelques modèles conceptuels [114, 48, 72] couramment utilisés. Ensuite, nous avons présenté des exemples d'implémentations d'intergiciels orientées services, par exemple, OSGi, OSGi Declarative Services.

Les propriétés les plus problématiques de la maison intelligente sont les aspects ouverts et embarqués. Dans une passerelle exécutant de nombreuses applications livrées par plusieurs fournisseurs de services non fiables, l'isolation est nécessaire pour éviter certaines menaces de sécurité et pour permettre un niveau minimum de robustesse contre les applications à comportement anormal. Par conséquent, nous discutons cinq mécanismes d'isolation basés sur la protection matérielle et des approches de virtualisation, à savoir, les processus [37, 94], bacs-à-sable [132, 53, 135], conteneurs [77, 95, 106, 70, 121], machines virtuelles [115, 111, 43, 94], exokernels et systèmes d'exploitation bibliothèques [45, 2, 105, 16, 66]. Nous comparons ceux-ci en fonction de leur impact sur les performances par rapport à l'isolation qu'ils fournissent, et nous pensons que les technologies de conteneurs offrent le meilleur choix. Le Chapitre 2 décrit Jasmin: un intergiciel pour le développement, le déploiement, l'isolation et l'administration des applications orientées services et à base de composants ciblant les systèmes embarqués. Jasmin est basé sur l'implémentation MIND du modèle de composants Fractal. Il fournit une isolation des applications basées sur la technologie des conteneurs Linux, et il étend l'architecture orientée services aux applications isolées, et il permet une communication rapide et transparente entre les applications isolées. Ensuite, nous invoquons des mécanismes d'isolation basés sur la sécurité de types de langages, par exemple, Singularity [60], SPIN [19], KaffeOS [12]. Nous illustrons le langage Java comme exemple d'un langage statique et sûr, et nous décrivons brièvement les principales parties de la machine virtuelle Java. Nous discutons également des techniques pour exécuter plusieurs applications isolées dans une seule machine virtuelle Java, et nous pensons que le partitionnement des applications [79] est inappropriée pour les applications domotiques qui ont besoin de communiquer facilement et rapidement.

[1] Un modèle à composants.
[2] Une implémentation de modèle à composants.

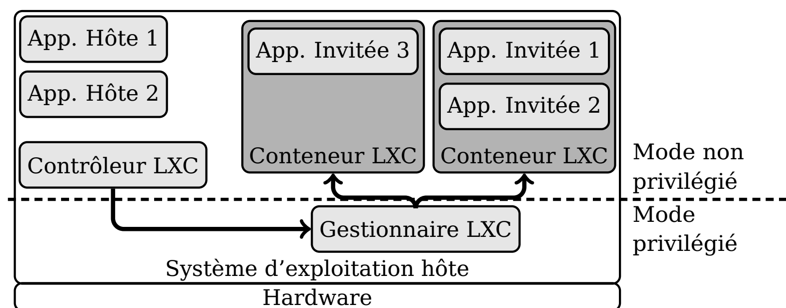


Figure 1.3: Structure des conteneurs Linux (LXC).

La plupart des approches d'isolation ne sont pas parfaits, laissant place aux conflits de partage de ressources. Dans notre quête pour résoudre ces conflits, nous avons besoin d'outils pour les découvrir en premier lieu. En d'autres termes, nous devons surveiller ces conflits de partage. Ainsi, nous commençons par discuter de la notion de «conflit de ressources», puis nous exposons les différentes approches de surveillance dans différents environnements logiciels [84, 71, 41, 100, 52, 99, 86, 46, 83], et nous décrivons les fondamentaux points de vue complémentaires en comptabilité des ressources. Enfin, nous détaillons une source typique de conflits de mémoire en machines virtuelles Java multi-locataires, en discutant de ses implications majeures et des exemples d'efforts de référence [47, 65, 87, 22, 90] pour y faire face. La contribution présentée au Chapitre 4 propose un mécanisme de surveillance de mémoire qui est conscient de la conception à base de composants de la plate-forme de maison intelligente, et qui fournit des données de surveillance qui sont précises et pertinentes à ce modèle.

3. Jasmin

Nous avons présenté Jasmin, un intergiciel de maison intelligente ouvert et robuste qui héberge les applications fournissant des services destinés à l'utilisateur final. Les services offerts par l'intergiciel Jasmin ouvrent des voies à des modèles d'affaires nouveaux et attrayants tels que «Entreprise à Entreprise au Consommateur» (B2B2C), par l'ouverture de la plate-forme de la maison intelligente à tout fournisseur de services qui vise à exposer ses services directement à l'intérieur de la maison de l'utilisateur final.

Jasmin suit le paradigme de l'architecture orienté services [103] et permet un déploiement facile et dynamique des applications en automatisant la plupart des étapes de déploiement (voir Figure 1.4). Jasmin exécute des applications basées sur le cadriciel MIND implémentant le modèle de composants Fractal qui pousse les développeurs à produire des services proprement conçus. Ceci permet un développement et un déploiement de services rapide et facile, et contribue ainsi à faire de Jasmin une plate-forme attrayante pour de nombreux fournisseurs de services.

Les risques de sécurité et de robustesse introduites par cette ouverture sont résolus par Jasmin via l'utilisation des conteneurs d'isolation. Jasmin propose des conteneurs d'isolation

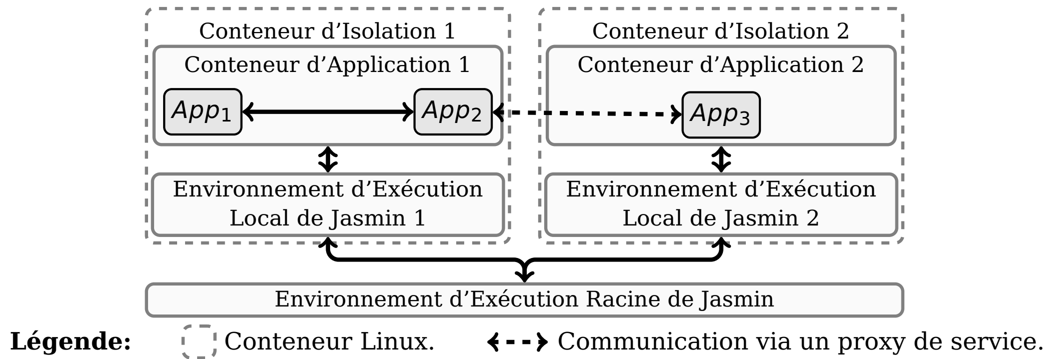


Figure 1.4: Architecture distribuée de Jasmin.

dans de multiples niveaux sélectionnables et convenables pour de différentes exigences d'applications et niveaux de confiance. Il implémente son plus haut niveau d'isolation en se basant sur les conteneurs Linux, qui offrent des garanties élevées d'isolation à un faible coût en performance.

Afin de développer des services riches, les fournisseurs de services doivent pouvoir invoquer facilement des services existants. Cela implique que les applications isolées doivent pouvoir communiquer. Jasmin se charge de la complexité d'invocation de services distants en chargeant automatiquement les proxys d'interfaces et en effectuant la sérialisation et le transfert de données en fonction des besoins. Cela permet aux consommateurs de services d'invoquer facilement et de façon transparente les services locaux et distants.

Jasmin exécute les applications sur une couche d'abstraction du système d'exploitation (OSAL). Cela permet une portabilité plus facile des applications aux différents systèmes d'exploitation, et facilite le portage de Jasmin lui-même aux autres systèmes. C'est surtout un grand pas pour maîtriser l'hétérogénéité de la maison intelligente.

L'évaluation de Jasmin montre que celui-ci respecte la nature embarquée des appareils de la maison. En fait, Jasmin non seulement dispose d'une faible utilisation des ressources en termes de CPU et de mémoire et d'espace disque, mais aussi induit un coût très faible pour les applications qu'il héberge.

4. Incinerator

OSGi est de plus en plus utilisé dans l'environnement de la maison intelligente comme un cadriciel pour héberger des applications orientées services délivrés par plusieurs tiers. Cela rend les références obsolètes une menace croissante au cadriciel et aux applications en cours d'exécution. Nous présentons Incinerator, qui aborde le problème de références obsolètes en OSGi et les fuites de mémoire qu'elles provoquent, en étendant le ramasse miettes de la machine virtuelle Java pour tenir compte des informations d'état des applications.

Nous abordons les problèmes soulevés par les références obsolètes en OSGi au niveau de

la machine virtuelle Java en proposant Incinerator [9] une extension aux ramasse miettes consciente d'OSGi. L'approche d'Incinerator est d'intégrer la détection de référence obsolètes dans la phase de collecte des miettes. Cette approche induit un faible coût puisque la phase de collecte de miettes traverse déjà tous les objets vivants, et la vérification de l'obsolescence d'une référence nécessite quelques opérations. Cette approche est également indépendante de l'algorithme spécifique de ramasse miettes, car il ne nécessite que la modification de la fonction qui balaye les références et les objets contenus à l'intérieur d'un objet donné. Lorsque Incinerator trouve une référence, il vérifie si l'objet référencé appartient à une application désinstallée ou à une version précédente d'une application mise à jour. Dans ce cas, la référence est identifiée comme obsolète et Incinerator la définit à `null`. Par conséquence, aucun objet obsolète ne reste accessible à la fin de la collecte et la mémoire associée est récupérée par le ramasse miettes.

Incinerator détecte plus de références obsolètes que le détecteur de référence obsolètes existant, Service Coroner. En fait, alors que Service Coroner ne fait que détecter les références obsolètes, Incinerator les élimine aussi en les définissant à `null`. Cela permet au ramasse miettes de récupérer les objets obsolètes référencés. En effet, nous avons constaté que les références obsolètes peuvent provoquer des fuites de mémoire importantes, telles que la fuite de mémoire 6 mégaoctets sur chaque mise à jour du «bundle» `HTTP-Server` causée par la défaillance de référence obsolète que nous avons découvert dans Knopflerfish. La prévention des fuites de mémoire augmente la disponibilité de la machine virtuelle Java, ce qui est une mesure importante dans les passerelles domotiques.

Incinerator est essentiellement indépendant d'une implémentation spécifique d'OSGi et, en effet, seulement 10 lignes doivent être modifiées dans le cadriciel OSGi Knopflerfish afin d'intégrer Incinerator. La surcharge du CPU induite par Incinerator est toujours inférieur à 1,2% sur les applications de la suite de tests DaCapo sur un ordinateur haut de gamme, et moins de 3,3% sur un ordinateur bas de gamme. Ce dernier résultat montre que Incinerator est utilisable dans les systèmes de domotiques ayant une puissance de calcul limitée.

Remarques de Compatibilité

Incinerator modifie le comportement de la machine virtuelle Java, parce qu'il annule les références qu'il trouve obsolètes. Nous avons étudié les problèmes de compatibilité qui pourraient découler de ce changement. Il convient de noter qu'une application correctement écrite qui libère ses références obsolètes, quand une application est désinstallée ou mise à jour, n'est jamais affectée par Incinerator. Quand une application ne libère pas une référence obsolète, notre choix de conception est de minimiser l'impact de l'annulation, tout en assurant que la mémoire est libérée. Trois situations peuvent se produire, selon la façon dont la référence obsolète est utilisée:

- Si la référence obsolète n'est jamais utilisée, alors l'annuler n'a aucun impact sur les autres applications.

- Si la référence obsolète est accessible uniquement dans le cadre d'une opération de nettoyage, c'est-à-dire, dans la méthode `finalize()`, alors Incinerator exécute d'abord cette opération de nettoyage afin d'éviter d'autres types de fuites.

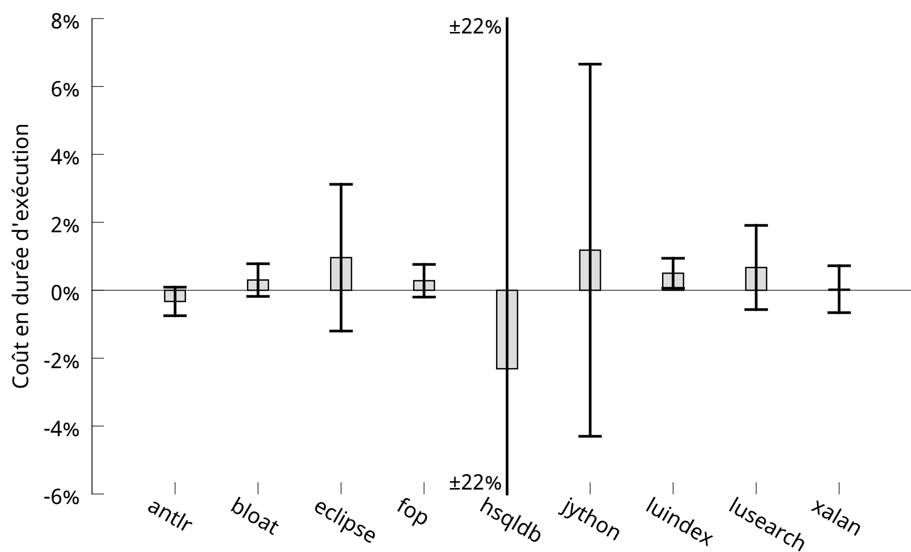


Figure 1.5: Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 entre J3 et Incinerator executées sur un ordinateur bas de gamme. `hsqldb` présente un écart-type de 22%, tronqué ici pour la clarté.

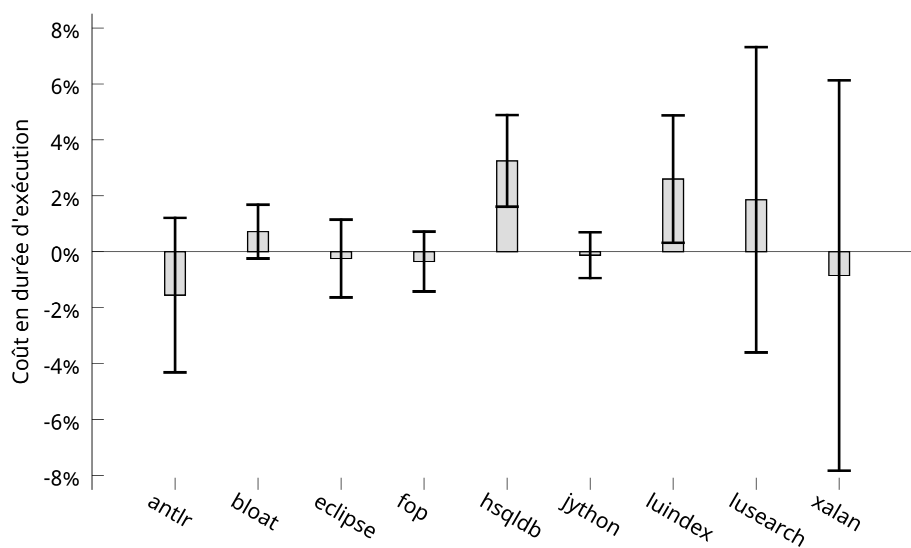
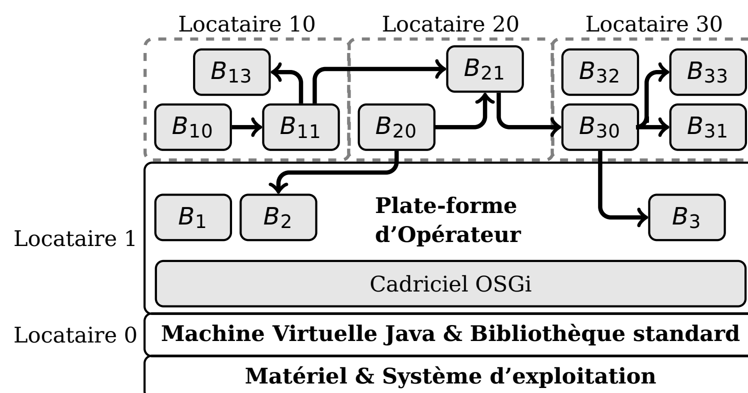


Figure 1.6: Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 entre J3 et Incinerator executées sur un ordinateur haut de gamme.

- Si la référence obsolète est utilisée ailleurs, que ce soit pour accéder ou se synchroniser sur l'objet obsolète, l'application qui détient la référence obsolète est défaillante puisque l'utilisation de l'objet obsolète conduirait à des opérations potentiellement conflictuelles. Puisque la référence a été annulée, une telle application défaillante reçoit une `NullPointerException`, ce qui aide les développeurs à identifier la défaillance, en la rendant visible. Si un fil d'exécution est bloqué en attente d'une synchronisation sur la référence obsolète, Incinerator débloque aussi le fil d'exécution, afin d'éviter les interblocages ou les fuites du fil d'exécution et de ses objets accessibles.

5. Surveillance de Mémoire en OSGi

Nous présentons un système de surveillance de mémoire conscient d'OSGi [10] et qui est principalement transparent aux développeurs d'applications, et qui permet la collaboration entre les locataires distincts partageant le même environnement d'exécution OSGi. Le système surveille les appels entre locataires et fournit, sur demande, des captures instantanés de statistiques d'utilisation de la mémoire pour les différents locataires.



Légende: B_1,...,B_{33} sont des «bundles» OSGi.

Figure 1.7: Environnement d'exécution multi-locataire basé sur OSGi.

Nous présentons un système de surveillance qui surveille l'utilisation de mémoire au niveau de granularité des applications, sans nécessiter l'isolation des locataires distincts. Notre système est beaucoup moins intrusif que les systèmes et les méthodes existantes, et il n'assume pas des locataires fiables. Le système de surveillance est modulaire et principalement indépendant des implémentations du cadriciel OSGi et du ramasse miettes.

Pour résoudre le problème de la comptabilisation des ressources au cours des interactions entre les locataires, le système de surveillance a des règles implicites prédéfinis de comptabilité des ressources qui décrivent correctement la plupart des interactions entre les locataires. Pour les interactions qui ne sont pas correctement comptabilisées, le système permet de spécifier des règles comptables explicites sous la forme de fichiers de configuration simples chargés par le système de surveillance au démarrage de la machine virtuelle Java. Notre prototype nécessite que les règles de comptabilité des ressources restent constants

pendant la durée de vie de la machine virtuelle Java. Lors de l'exécution, le système de surveillance applique des règles implicites et explicites pour tenir compte correctement de la mémoire utilisée par les applications dans les variables locales, les classes chargées, et les objets créés. En pratique, la plupart des applications n'ont pas besoin d'écrire des règles comptables parce que les règles implicites gèrent leurs interactions correctement. Les applications qui doivent écrire des règles comptables explicites sont principalement celles qui génèrent l'activité asynchrone, telles que celles qui publient des événements. Les exemples incluent le cadriciel OSGi, et les applications exposant les données de capteurs à la maison en temps réel. Cependant, la plupart des applications consomment services séquentiellement et fournissent des services uniquement sur demande.

Basé sur la suite de tests DaCapo, nous avons montré que la surcharge de notre système était au dessous de 46% pour des applications Java concrètes. Cette surcharge est acceptable en développement et en test, et elle est tolérable dans les applications à rythme lent qui sont fréquentes dans la maison intelligente. Notre enquête approfondie sur le coût en performances a montré les aspects spécifiques de surveillance qui ont causé la majorité du coût, et les types d'applications qui souffriront le plus de la surveillance de mémoire.

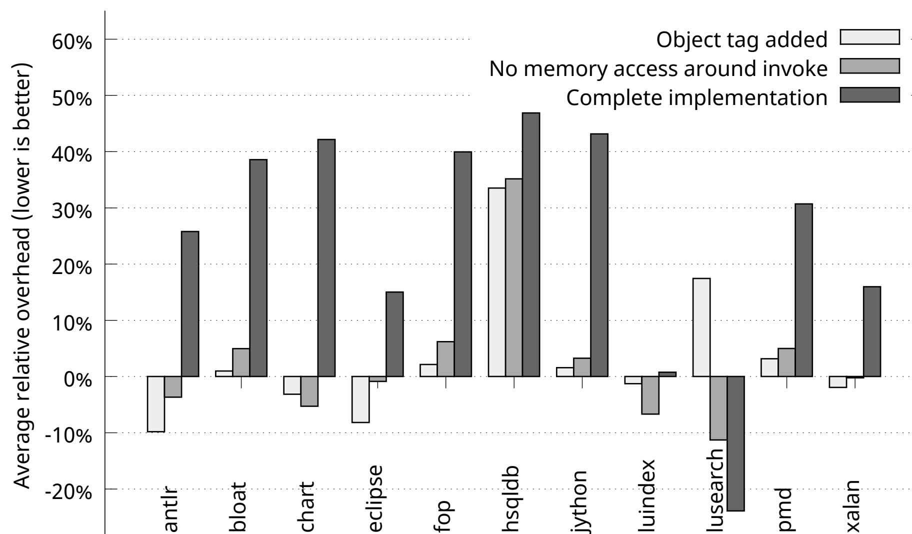


Figure 1.8: Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 en s'exécutant sur des implémentations partielles du système de surveillance memoire, comparées à la machine virtuelle Java d'origine (J3/VMKit). Moyenne de 10 exécutions.

6. Conclusion

Tout au long de cette thèse, nous avons décrit notre travail sur la gestion des conflits de partage de ressources dans l'environnement de la maison intelligente. Nous avons commencé notre quête par décrire l'environnement matériel de la maison intelligente, afin de synthétiser

ses principales propriétés, à savoir, l'ouverture, dynamicité, le développement rapide, l'hétérogénéité, la distribution et les aspects embarqués. Nous avons étudié chaque propriété pour révéler les défis qui en découlent, puis nous nous sommes concentrés sur chacun de ces défis, à la recherche de travaux existants qui y font face. Compte tenu des propriétés particulières de la maison intelligente, nous avons soutenu l'argument que la passerelle domotique doit intégrer une plate-forme et des applications à base de composants [38] et orientée services [103]. Cela nous a conduit à considérer des modèles de composants tels que Fractal [29] et OSGi [127], et les plates-formes orientées services tels que OSGi Declarative Services [126].

Afin de gérer les conflits de partage de ressources, il faut les découvrir, les prévenir autant que possible, et les résoudre lorsque la prévention est hors de portée. Notre première contribution, appelé Jasmin [4], est un effort pour éviter les conflits de partage de ressources dans la passerelle domotique. Jasmin est un intergiciel pour le développement, le déploiement, l'isolation et l'administration des applications orientées services et à base de composants ciblant les systèmes embarqués. Le prototype de Jasmin est basée sur l'implémentation MIND [92] du modèle de composants Fractal. Jasmin propose plusieurs niveaux d'isolation des applications, de la faible la séparation conceptuelle, à la forte isolation basée sur la technologie des conteneurs Linux [77] qui offrent un excellent compromis entre isolation et coût en performances. Jasmin implémente une architecture orientée service et l'étend aux applications isolées en définissant un coordinateur de registre de service qui permet la résolution des dépendances de service indépendamment de l'emplacement du service. De plus, Jasmin permet une communication transparente entre les applications isolées en chargeant automatiquement et liant les proxys de services. Ces proxys de services sont basés sur un mécanisme de communication entre conteneurs implémenté par Jasmin, qui est 60% plus rapide que les communications locales à la machine via une implémentation standard de RPC [118] sous Linux. Dans le cadre de l'évaluation, nous décrivons l'effort nécessaire pour porter une application multimédia existante à l'intergiciel Jasmin. Nous montrons que l'effort de portage est relativement faible, et qu'il peut être réalisé progressivement. Nous montrons également que les performances des applications Jasmin sont presque identiques aux performances des applications existantes.

Compte tenu des propriétés de la maison intelligente, nous avons proposé l'exécution d'une machine virtuelle Java multi-locataire dans la passerelle de la maison intelligente, qui accueillerait plusieurs applications de la maison intelligente. S'agissant d'un système s'exécutant pour de longues durées, et à capacité matérielle contrainte, la passerelle domotique est vulnérable aux références obsolètes, qui provoquent des fuites de mémoire considérables, augmentant par conséquent les risques de conflits de mémoire dans la passerelle et les incohérences d'état dans les appareils contrôlés par la passerelle. Notre deuxième contribution, appelée Incinerator [9], résout le problème de références obsolètes en Java en détectant et en éliminant celles-ci, avec moins de 4% de surcharge CPU. Par rapport à l'outil de référence en détection de références obsolètes, Service Coroner [47], Incinerator détecte plus de références obsolètes, et va plus loin en les éliminant. Incinerator nous a également permis de contribuer à la communauté en découvrant et en résolvant un bug de référence obsolète [8] dans Knopflerfish [81], une implémentation à source ouverte d'OSGi.

En raison de la mémoire limitée disponible dans la passerelle domotique, les conflits de partage de mémoire sont un enjeu important dans la passerelle, en particulier lors de

l'exécution d'une machine virtuelle Java multi-locataire. Le problème de la surveillance mémoire apparaît naturellement dans ce contexte, ce qui est abordé par notre troisième contribution: surveillance mémoire dans les plates-formes OSGi. Nous proposons un système de surveillance conscient d'OSGi [10] et majoritairement transparent pour les développeurs d'applications, et qui permet la collaboration entre les différents fournisseurs de services qui partagent le même environnement d'exécution OSGi. Le système surveille les appels entre applications et fournit, à la demande, des captures instantanés de statistiques d'utilisation de la mémoire pour les différents fournisseurs de services, à un coût inférieur à 46%. Les données rapportées sont suffisamment précises parce que le système bascule entre la comptabilité directe et indirecte en fonction du contexte. Cette commutation est basée sur un algorithme qui reconnaît la plupart des contextes d'appels et comptabilise précisément les ressources consommées au cours de l'appel. Les décisions de l'algorithme peuvent être substituées par l'administrateur de la plate-forme par l'intermédiaire d'un ensemble de règles simples qui sont introduits dans le système de surveillance. Les règles sont écrites dans un langage simple et versatile spécifique au domaine et défini pour l'affaire.

Perspectives

Nous croyons que la gestion des conflits de ressources dans la maison intelligente doit être la responsabilité de plusieurs couches logicielles et approches. Compte tenu des propriétés de la maison intelligente, nous pensons que la sécurité doit être discutée au cours des premiers stades de développement de plates-formes et des applications ciblées à la domotique, comme la sécurité a souvent des répercussions sur l'ensemble du système. La sécurité est une décision conceptuelle, pas une option.

Les approches d'isolation très proches doivent être analysées plus loin, tels que les conteneurs [77, 95, 106, 70, 121], exokernels [45, 2, 105, 16, 66], bacs-à-sable [132, 53, 135], etc. Vu qu'il y a relativement peu d'applications domotiques, même les solutions non rétrocompatibles peuvent être tentées, telles que les systèmes d'exploitation et intergiciels basés sur la sécurité du langage de programmation [60, 12, 19].

Afin de résoudre les conflits de ressources et réguler l'utilisation des ressources, on pourrait tenter de migrer des applications entre les différents appareils dans la la maison intelligente [44]. Cela permet non seulement une utilisation équilibrée des ressources, mais aussi une réduction de la consommation électrique lorsque très peu de ressources sont nécessaires, en migrant les applications dans la passerelle et en "hibernant" les autres appareils.

Jasmin expose un service d'administration pour les agents administratifs sur qui sont actuellement exclusivement locaux. L'administration à distance devient une exigence forte dans la maison intelligente. Différents agents d'administration à distance pourraient être construites, implémentant le protocole CWMP appelé aussi TR-069 [27], et UPnP-DM [129]. Actuellement, l'implémentation du conteneur d'isolation dépend du noyau Linux. Cependant, l'architecture distribuée de Jasmin est adaptée pour supporter d'autres conteneurs des systèmes d'exploitation, tels que FreeBSD Jails [106] et Solaris Zones [95]. Les développeurs doivent être en mesure de contrôler ou aider la résolution de dépendances effectuée par Jasmin. Cela pourrait être fait en ajoutant des propriétés et des filtres pour les interfaces

fournies ou requises. Les fonctionnalités d'extensibilité de la chaîne d'outils MIND sont appropriés pour ajouter ce support. Cela pourrait également devenir très pratique pour éviter les boucles de dépendances circulaires entre applications.

Le prototype d'Incinerator définit les références obsolètes selon le modèle de composant OSGi. Le portage de Incinerator au modèle de composant de OSGi Declarative Services devrait aider à détecter et à éliminer les références obsolètes pour les composants de OSGi Declarative Services, par conséquent rendant le modèle de composant plus robuste.

L'information pertinente communiquée par le système de surveillance peut devenir une entrée précise à un gestionnaire de ressources autonome[46, 83], permettant à ce dernier de détecter automatiquement les conflits de ressources, comme un premier pas vers leur résolution. Nous cherchons de moyens pour rendre dynamiques la liste des règles comptables tout en gardant le coût acceptable. Rendre les règles explicites encore plus génériques, par exemple à base d'expressions régulières, permettrait la factorisation de nombreuses règles explicites, mais peut réduire les performances de correspondance par rapport aux règles, nous sommes donc encore incertains si plus d'expressivité vaudrait la perte de performance. Nous voudrons également surveiller d'autres ressources pertinentes, telles que, l'utilisation du processeur, l'utilisation du disque et de bande passante réseau.

Bibliography

[1] B. Alpern, C. R. Attanasio, J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. E. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. doi: 10.1147/sj.391.0211.

[2] T. E. Anderson. The case for application-specific operating systems. Technical report, Division of Computer Science, University of California, Berkeley, 1993. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/6023.html`.

[3] M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polakovic, M. Poulhiès, J. Pulou, S. Seyvoz, J. Tous, and T. Watteyne. Think: View-based support of non-functional properties in embedded systems. In *Proceedings of the 2009 International Conference on Embedded Software and Systems*, ICESS'09, pages 147–156, May 2009. doi: 10.1109/ICESS.2009.30. URL `http://think.ow2.org/`.

[4] M. Anne, K. Attouchi, D. H. de Villeneuve, and J. Pulou. Jasmin: an alternative for secure modularity inside the digital home. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE'12, pages 145–150. ACM, 2012.

[5] D. N. Antonioli and M. Pilz. Analysis of the Java class file format. Technical report, University of Zurich, Apr. 1998.

[6] Arduino Project. Arduino Yún, May 2014. URL `http://arduino.cc/en/Main/ArduinoBoardYun`. [Online; accessed 21 May 2014].

[7] ARM. Procedure Call Standard for the ARM Architecture. Technical report, ARM, Nov. 2012. URL `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf`.

[8] K. Attouchi. gatespace.org / bugs / #175 HTTP-Server bundle causes a memory leak when uninstalled, Mar. 2013. URL `http://sourceforge.net/p/gatespace/bugs/175/`. [Online; accessed 17 March 2014].

[9] K. Attouchi, G. Thomas, A. Bottaro, J. L. Lawall, and G. Muller. Incinerator - Eliminating stale references in dynamic OSGi applications. Rapport de recherche RR-8485, INRIA, Feb. 2014. URL `http://hal.inria.fr/hal-00952327`.

[10] K. Attouchi, G. Thomas, A. Bottaro, and G. Muller. Memory Monitoring in a Multi-tenant OSGi Execution Environment. In *Proceedings of the 17th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE'14, Lille, France, July 2014. ACM.

[11] A. Azeez, S. Perera, D. Gamage, R. Linton, P. Siriwardana, D. Leelaratne, S. Weerawarana, and P. Fremantle. Multi-tenant SOA middleware for cloud computing. In *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 458–465, July 2010. doi: 10.1109/CLOUD.2010.50.

[12] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, OSDI'00, pages 23–23. USENIX, 2000.

[13] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, PLDI'98, pages 258–268. ACM, 1998.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003. SOSP'03.

[15] V. Bauche. JSR 361: Java ME Embedded Profile, Apr. 2014. URL `https://www.jcp.org/en/jsr/detail?id=361`. [Online; accessed 09 April 2014].

[16] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'13, pages 239–252, New York, NY, USA, 2013. ACM. URL `http://doi.acm.org/10.1145/2465351.2465375`.

[17] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile software development, May 2014. URL `http://agilemanifesto.org/`. [Online; accessed 23 May 2014].

[18] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–6, Anaheim, CA, USA, Feb. 2005. ATEC'05.

[19] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, SOSP'95, pages 267–283, New York, NY, USA, 1995. ACM. URL `http://doi.acm.org/10.1145/224056.224077`.

[20] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE'04, pages 137–146. IEEE, 2004.

[21] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'06, pages 169–190. ACM, 2006. URL http://www.dacapobench.org/benchmarks.html.

[22] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *Proceedings of the 23th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'08, pages 109–126, New York, NY, USA, 2008. ACM. URL http://doi.acm.org/10.1145/1449764.1449774.

[23] G. Bonnardel, A. Bottaro, S. Dimov, E. Grigorov, and A. Rinquin. OSGi RFC 200 resource management. Request for comments, OSGi Alliance, Dec. 2013.

[24] J. Bonwick and B. Moore. *ZFS: The Last Word In File Systems*. Sun Microsystems, Inc., Dec. 2007. URL http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf. [Online; accessed 17 March 2014].

[25] D. Bornstein. Dalvik vm internals, May 2008. URL http://sites.google.com/site/io/dalvik-vm-internals/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf. [Online; accessed 24 April 2014].

[26] A. Bottaro and F. Rivard. OSGi ME - an OSGi profile for embedded devices. In *OSGi Community Event 2010*, London, UK, Sept. 2010.

[27] Broadband Forum. Broadband Forum - CWMP, 2014. URL http://www.broadband-forum.org/cwmp.php. [Online; accessed 19 March 2014].

[28] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model. Technical report, INRIA and France Telecom R&D, Feb. 2004. URL http://fractal.ow2.org/specification/.

[29] 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 (SP&E)*, 36(11-12):1257–1284, 2006. Special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[30] M. Condry, U. Gall, and P. Delisle. Open Service Gateway Architecture Overview. In *The 25th Annual Conference of the IEEE on Industrial Electronics Society, 1999*, volume 2 of *IECON'99*, pages 735–742. IEEE, 1999.

[31] O. Consortium. Cecilia framework, Apr. 2009. URL http://fractal.ow2.org/cecilia-site/current/. [Online; accessed 17 March 2014].

[32] O. Consortium. Fractal ADL, Dec. 2012. URL http://fractal.ow2.org/fractaladl/. [Online; accessed 17 March 2014].

[33] O. Consortium. Fractal - julia, Dec. 2012. URL `http://fractal.ow2.org/julia/`. [Online; accessed 27 March 2014].

[34] M. E. Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, Oct. 1958. URL `http://doi.acm.org/10.1145/368924.368928`.

[35] J. Corbet. Btrfs: Getting started, Dec. 2013. URL `http://lwn.net/Articles/577218/`. [Online; accessed 17 March 2014].

[36] J. Courtney. JSR 36: Connected Device Configuration, Dec. 2005. URL `https://www.jcp.org/en/jsr/detail?id=36`. [Online; accessed 09 April 2014].

[37] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, Sept. 1981.

[38] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5): 593–615, Sept. 2011. doi: 10.1109/TSE.2010.83.

[39] D. Curtis, C. Stone, and M. Bradley. IIOP: OMG's Internet Inter-ORB Protocol – A Brief Description, June 2012. URL `http://www.omg.org/library/iiop4.html`. [Online; accessed 23 May 2014].

[40] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'01, pages 125–138. ACM, 2001.

[41] G. Czajkowski and T. von Eicken. Jres: a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'98, pages 21–35. ACM, 1998.

[42] P. J. Denning. Before memory was virtual. *In the Beginning: Personal Recollections of Software Pioneers*, 1996.

[43] P. J. Denning. Origin of virtual machines and other virtualities. *IEEE Annals of the History of Computing*, 23(3):73, 2001.

[44] R. Druilhe. *L'Efficience Énergétique des Services dans les Systèmes Répartis Hétérogènes et Dynamiques : Application à la Maison Numérique [Energy Efficient Services in Distributed, Heterogeneous and Dynamic Environments: Application to the Digital Home].* PhD thesis, Université Lille 1 - Sciences et technologies, Dec. 2013. URL `http://tel.archives-ouvertes.fr/tel-00915321`.

[45] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, SOSP'95, pages 251–266, New York, NY, USA, 1995. ACM. URL `http://doi.acm.org/10.1145/224056.224076`.

[46] J. Ferreira, J. Leitão, and L. Rodrigues. A-OSGi: A framework to support the construction of autonomic OSGi-based applications. In A. Vasilakos, R. Beraldi, R. Friedman, and M. Mamei, editors, *Autonomic Computing and Communications Systems*, volume 23 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 1–16. Springer Berlin Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-11482-3_1`.

[47] K. Gama and D. Donsez. Service coroner: A diagnostic tool for locating OSGi stale references. In *Software Engineering and Advanced Applications, 2008. 34th Euromicro Conference*, SEAA'08, pages 108–115. IEEE, 2008.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, Nov. 1994.

[49] F. R. Güntsch. *Logischer Entwurf eines digitalen Rechnergerätes mit mehreren asynchron laufenden Trommeln und automatischem Schnellspeicherbetrieb [Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation]*. Technische Universität Berlin, May 1956. [Doctoral dissertation, D 83].

[50] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *IEEE/IFIP International Conference on Dependable Systems & Networks, 2009*, DSN'09, pages 544–553. IEEE, 2009.

[51] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, VEE'10, pages 51–62. ACM, 2010.

[52] B. Goetz. Java theory and practice: Instrumenting applications with JMX, Sept. 2006. URL `http://www.ibm.com/developerworks/java/library/j-jtp09196/index.html`. [Online; accessed 16 April 2014].

[53] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, volume 6 of *SSYM'96*, 1996.

[54] L. Gong. A software architecture for open service gateways. *IEEE Internet Computing*, 5(1):64–70, Jan. 2001. doi: 10.1109/4236.895144.

[55] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} language specification*. Addison-Wesley, 3rd edition, 2005.

[56] G. T. Heineman and W. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, USA, 2001.

[57] R. Hipp. Sqlite, 2014. URL `http://www.sqlite.org/`. [Online; accessed 17 March 2014].

[58] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In L. Grunske, R. Reussner, and F. Plasil, editors, *Proceedings of the 13th international conference on Component-Based Software Engineering*, volume 6092 of *CBSE'10*, pages 21–36. Lars Grunske and Ralf Reussner and Frantisek Plasil, Prague, Czech Republic, 2010.

[59] Home Gateway Initiative. Requirements for software modularity on the home gateway, version 1.0. Technical report, Home Gateway Initiative, June 2011.

[60] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007*, EuroSys'07, pages 341–354. ACM, 2007.

[61] E. Jeseen. Origin of the virtual memory concept. *IEEE Annals of the History of Computing*, pages 71–73, Oct. 2004.

[62] G. Johnson and M. Dawson. Introduction to Java multitenancy, May 2014. URL `http://www.ibm.com/developerworks/java/library/j-multitenant-java/`. [Online; accessed 15 May 2014].

[63] M. T. Jones. Anatomy of linux dynamic libraries, Aug. 2008. URL `http://www.ibm.com/developerworks/library/l-dynamic-libraries/`. [Online; accessed 24 April 2014].

[64] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[65] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL'07, pages 31–38, New York, NY, USA, 2007. ACM. URL `http://doi.acm.org/10.1145/1190216.1190224`.

[66] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, volume 31 of *SOSP'97*, pages 52–65. ACM, 1997.

[67] S. Kächele and F. J. Hauck. Component-based scalability for cloud applications. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP'13, pages 19–24. ACM, 2013. URL `http://doi.acm.org/10.1145/2460756.2460760`.

[68] M. Kessis, P. Déchamboux, C. Roncancio, T. Coupaye, and A. Lefebvre. Towards a flexible middleware for autonomous integrated management applications. In *Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, Bucharest, Aug. 2006. ICCGI'06.

[69] T. Kilburn, D. B. Edwards, M. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, Apr. 1962.

[70] K. Kolyshkin. Virtualization in linux. *(Unpublished in a journal)*, pages 1–5, Sept. 2006.

[71] J. Koshy. pmcstat – performance measurement with performance monitoring hardware, Sept. 2008. URL `http://www.freebsd.org/cgi/man.cgi?query=pmcstat&manpath=FreeBSD+10.0-RELEASE`. [Online; accessed 16 April 2014].

[72] P. Kriens and B. Hargrave. Listeners Considered Harmful: The "Whiteboard" Pattern. Technical report, The OSGi Alliance, Aug. 2004.

[73] C. Larsson and C. Gray. Challenges of resource management in an OSGi environment. In *OSGi Community Event 2011*, Darmstadt, Germany, Sept. 2011.

[74] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, CGO'04, Palo Alto, California, Mar. 2004.

[75] J. Löwy. *COM and .NET Component Services*. O'Reilly Media, 1st edition, Oct. 2001.

[76] T. Lindholm and F. Yellin. *The Java^{TM} virtual machine specification*. Addison-Wesley, 2nd edition, 1999.

[77] LXC Community. LXC - Linux Containers, 2014. URL `https://linuxcontainers.org/`. [Online; accessed 17 March 2014].

[78] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz. *Reference Model for Service Oriented Architecture 1.0 – OASIS Standard*. OASIS Open, Oct. 2006. URL `http://docs.oasis-open.org/soa-rm/v1.0/`.

[79] S. Majoul, M. Richard-Foy, A. Agirre, A. Péandrez, and A. Kung. Enforcing trust in home automation platforms. In *IEEE Conference on Emerging Technologies and Factory Automation, 2010*, ETFA'10, pages 1–6, Sept. 2010. doi: 10.1109/ETFA.2010.5641362.

[80] Makewave. Certified OSGi technology and services from the source of Knopflerfish | Home | Makewave, 2014. URL `http://www.makewave.com/`. [Online; accessed 21 March 2014].

[81] Makewave AB Corp. Knopflerfish OSGi - open source OSGi service platform. OSGi R5, R4 and R3, 2014. URL `http://www.knopflerfish.org/`. [Online; accessed 17 March 2014].

[82] D. Marples and P. Kriens. The Open Services Gateway Initiative: An Introductory Overview. *IEEE Communications Magazine*, 39(12):110–114, Dec. 2001. URL `http://dx.doi.org/10.1109/35.968820`.

[83] Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi. Adaptive monitoring of end-user OSGi-based home boxes. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, CBSE'12, pages 157–166. ACM, 2012.

[84] P. Menage, P. Jackson, and C. Lameter. Control Groups, Feb. 2014. URL `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`. [Online; accessed 16 April 2014].

[85] Microsoft, Inc. Dynamic-link libraries, Apr. 2014. URL `http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589`. [Online; accessed 24 April 2014].

[86] T. Miettinen, D. Pakkala, and M. Hongisto. A method for the resource monitoring of osgi-based software components. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, SEAA'08, pages 100–107, Washington, DC, USA, 2008. IEEE Computer Society.

[87] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In L. Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40531-3. URL `http://dx.doi.org/10.1007/978-3-540-45070-2_16`.

[88] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'09, pages 265–276. ACM, 2009.

[89] P. A. Nelson. A comparison of PASCAL intermediate languages. In *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*, SIGPLAN'79, pages 208–213, New York, NY, USA, 1979. ACM. URL `http://doi.acm.org/10.1145/800229.806971`.

[90] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th international symposium on Memory Management*, ISMM'07, pages 15–30, New York, NY, USA, 2007. ACM. URL `http://doi.acm.org/10.1145/1296907.1296912`.

[91] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.3 – Part 2: CORBA Interoperability*. Object Management Group, Nov. 2012. URL `http://www.omg.org/spec/CORBA/3.3/Interoperability/PDF`.

[92] ObjectWeb 2 Consortium. The MIND project, Jan. 2013. URL `http://mind.ow2.org/`. [Online; accessed 17 March 2014].

[93] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of android dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES'12, pages 115–124, New York, NY, USA, 2012. ACM. URL `http://doi.acm.org/10.1145/2388936.2388956`.

[94] R. W. O'Neill. Experience using a time-shared multi-programming system with dynamic

address relocation hardware. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 611–621. ACM, Apr. 1967.

[95] Oracle Corp. *Consolidating Applications with Oracle Solaris Containers*. Oracle Corp., May 2010.

[96] Oracle Corp. VirtualBox user manual, Mar. 2011. URL `http://www.virtualbox.org/manual/UserManual.html`.

[97] Oracle Corp. Secure coding guidelines for the Java programming language, version 4.0, 2012. URL `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`. [Online; accessed 17 March 2014].

[98] Oracle Corp. Java Security Architecture, Mar. 2014. URL `http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc5.html`. [Online; accessed 05 May 2014].

[99] Oracle, Inc. JVM Tool Interface - Version 1.2, June 2013. URL `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`. [Online; accessed 16 April 2014].

[100] Oracle, Inc. VisualVM - All-in-one Java Troubleshooting Tool, Mar. 2014. URL `http://visualvm.java.net/`. [Online; accessed 16 April 2014].

[101] K. Palacz. JSR 121: Application isolation API specification, June 2006. URL `https://jcp.org/en/jsr/detail?id=121`. [Online; accessed 17 March 2014].

[102] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynas. Incommunicado: efficient communication for isolates. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'02, pages 262–274. ACM, 2002.

[103] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, Nov. 2007. doi: 10.1109/MC.2007.400.

[104] P. Parrend and S. Frénot. Classification of component vulnerabilities in java service oriented programming (SOP) platforms. *Component-Based Software Engineering*, pages 80–96, 2008.

[105] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, number 3 in ASPLOS'11, pages 291–304, New York, NY, USA, 2011. ACM. URL `http://doi.acm.org/10.1145/1950365.1950399`.

[106] T. F. D. Project. *FreeBSD Handbook*. The FreeBSD Documentation Project, Feb. 2011.

[107] ProSyst. ProSyst - Your Partner for Open Standards based Middleware, OSGi, 2014. URL `http://www.prosyst.com/`. [Online; accessed 21 March 2014].

[108] Raspberry Pi Foundation. Raspberry Pi, May 2014. URL `http://www.raspberrypi.org/`. [Online; accessed 21 May 2014].

[109] H. Reiser. Trees in the Reiser4 filesystem, Dec. 2002. URL `http://www.linuxjournal.com/article/6267`. [Online; accessed 17 March 2014].

[110] R. Riggs. JSR 271: Mobile Information Device Profile 3, Nov. 2009. URL `https://www.jcp.org/en/jsr/detail?id=271`. [Online; accessed 09 April 2014].

[111] M. Rosenblum. The reincarnation of virtual machines. *Queue*, 2:34–40, July 2004. URL `http://doi.acm.org/10.1145/1016998.1017000`.

[112] Y. Royon and S. Frénot. Multiservice home gateways: Business model, execution environment, management infrastructure. *IEEE Communications Magazine*, 45(10):122–128, 2007.

[113] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *IEEE Security and Privacy*, 8(2):35–44, 2010.

[114] A. Shvets, G. Frey, and M. Pavlova. Observer design pattern, Apr. 2014. URL `http://sourcemaking.com/design_patterns/observer`. [Online; accessed 28 April 2014].

[115] J. E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[116] T. B. Steel, Jr. A first version of UNCOL. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM'61, pages 371–378, New York, NY, USA, May 1961. ACM. URL `http://doi.acm.org/10.1145/1460690.1460733`.

[117] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communication with changing machines: A proposed solution. *Communications of the ACM*, 1(8):12–18, Aug. 1958. URL `http://doi.acm.org/10.1145/368892.368915`.

[118] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification version 2, June 1988. URL `http://www.ietf.org/rfc/rfc1057.txt`.

[119] Sun Microsystems, Inc. `java.lang.ref` (Java 2 platform SE 5.0), 2010. URL `http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/ref/package-summary.html`. [Online; accessed 17 March 2014].

[120] Sun Microsystems, Inc. and R. Srinivasan. XDR: External data representation standard, Aug. 1995. URL `http://www.ietf.org/rfc/rfc1832.txt`.

[121] SWsoft, Inc. *OpenVZ User's Guide*. SWsoft, Inc., Herndon, VA, USA, Sept. 2005.

[122] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley, Nov. 2002.

[123] A. Taivalsaari. JSR 30: J2ME Connected, Limited Device Configuration, May 2000. URL https://www.jcp.org/en/jsr/detail?id=30. [Online; accessed 09 April 2014].

[124] The bhyve development team. bhyve – The BSD Hypervisor, 2014. URL http://bhyve.org/. [Online; accessed 21 March 2014].

[125] The Embedded Debian Project. Embedded debian project, Jan. 2012. URL http://www.emdebian.org/. [Online; accessed 17 March 2014].

[126] The OSGi Alliance. *OSGi Service Platform: Service Compendium*. The OSGi Alliance, Aug. 2009. URL http://www.osgi.org/download/r4v42/r4.cmpn.pdf. [Release 4, Version 4.2].

[127] The OSGi Alliance. OSGi service platform core specification, release 4, version 4.2, 2009. URL http://www.osgi.org/download/r4v42/r4.core.pdf.

[128] R. Thomas and B. Reddy. Dynamic linking in linux and windows, part one, Nov. 2010. URL http://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one. [Online; accessed 24 April 2014].

[129] UPnP Forum. DeviceManagement:2 UPnP Forum, 2014. URL http://upnp.org/specs/dm/dm2/. [Online; accessed 19 March 2014].

[130] VideoLAN. VideoLAN - libVLC media player, Open Source video framework for every OS, 2014. URL http://www.videolan.org/vlc/libvlc.html. [Online; accessed 17 March 2014].

[131] VMware. Understanding full virtualization, paravirtualization, and hardware assist. Technical report, VMware, Nov. 2007. URL https://www.vmware.com/resources/techresources/1008.

[132] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP'93, pages 203–216, New York, NY, USA, 1993. ACM. URL http://doi.acm.org/10.1145/168619.168635.

[133] J. Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, July 1998. doi: 10.1109/4434.708248.

[134] M. Warkus. *The Official GNOME 2 Developer's Guide*. No Starch Press, Feb. 2004.

[135] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*,

SP'09, pages 79–93, Oakland, California, May 2009. IEEE Computer Society. doi: 10.1109/SP.2009.25.

List of Figures

1.1 Smart home structure. Communications between the "Operator Platform" and App_1,...,App_6 are omitted to simplify the figure. 2
1.2 Java references graph between objects, classes, and class loaders. 7
1.3 A component-based system. 11
1.4 MIND compilation flow. 12
1.5 OSGi bundle update. 15
1.6 Scenario of service discovery and service extinction in the Service-Oriented Architecture. 17
1.7 Publish-Subscribe Design Pattern. 19
1.8 Virtual address translation in processes. 20
1.9 Sandbox structure. 21
1.10 Linux Container (LXC) structure. 22
1.11 Multi-tenant Java-based execution environment 27
1.12 Java object lifetime from different perspectives. 32

2.1 Jasmin application life cycle. 40
2.2 Standalone Jasmin architecture. 41
2.3 Distributed Jasmin architecture. 42
2.4 CPU footprint of running Jasmin Player on different Jasmin execution environments. 50
2.5 Memory footprint of running Jasmin Player on different execution environments. 51

3.1 Incinerator eliminating stale references 55
3.2 Incinerator handling algorithm for finalizable objects. 60
3.3 Stale reference scenarios. 64
3.4 Hardware and software configuration of the Alarm controller application. . . . 66
3.5 Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a low-end computer. `hsqldb` has a standard deviation of 22%, truncated here for clarity. 68
3.6 Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a high-end computer. 69

4.1 Multi-tenant OSGi-based execution environment 72
4.2 Resource accounting during interaction between two tenants 75
4.3 DSL of resource accounting configuration 76
4.4 Sample resource accounting configuration 76
4.5 Accounting decisions made by the accounting algorithm 78
4.6 Tenants declarations in functional tests. 81

4.7 Detailed memory accounting calculations on a call stack of a thread internal to the OSGi framework. 82
4.8 Detailed memory accounting calculations on a thread call stack. 84
4.9 Functional tests components. 85
4.10 Execution overhead of the method call micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the "Zero implementation". Overhead is an average of 10 runs. 86
4.11 Execution overhead of the small objects micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the "Zero implementation". Overhead is an average of 10 runs. 87
4.12 Execution overhead of DaCapo 2006 benchmark applications when run on partial implementations of the monitoring subsystem, compared to the original JVM. Overhead is an average of 10 runs. 88

1.1 Structure domotique. Les communications entre la «Plate-forme d'Operateur» et App_1,...,App_6 sont omis pour simplifier la figure. 99
1.2 Un système à base de composants. 100
1.3 Structure des conteneurs Linux (LXC). 101
1.4 Architecture distribuée de Jasmin. 102
1.5 Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 entre J3 et Incinerator executées sur un ordinateur bas de gamme. `hsqldb` présente un écart-type de 22%, tronqué ici pour la clarté. 104
1.6 Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 entre J3 et Incinerator executées sur un ordinateur haut de gamme. . . . 104
1.7 Environnement d'exécution multi-locataire basé sur OSGi. 105
1.8 Coût moyen en durée d'exécution des applications de la suite de tests DaCapo 2006 en s'exécutant sur des implémentations partielles du système de surveillance memoire, comparées à la machine virtuelle Java d'origine (J3/VMKit). Moyenne de 10 exécutions. 106

List of Tables

1.1 Stale references found by Service Coroner [47]. 33

2.1 Comparison of execution speeds of function calls. 46
2.2 Jasmin disk footprint. 47
2.3 Jasmin memory footprint upon start up. 48

3.1 Micro-benchmark execution with standard JVMs, Service Coroner and Incinerator. 64