Incinerator - Eliminating Stale References in Dynamic OSGi Applications

Koutheir Attouchi, Gaël Thomas, André Bottaro, Julia L. Lawall, Gilles Muller

Incinerator – Eliminating Stale References in Dynamic OSGi Applications

Koutheir Attouchi, Gaël Thomas, André Bottaro, Julia Lawall, Gilles Muller

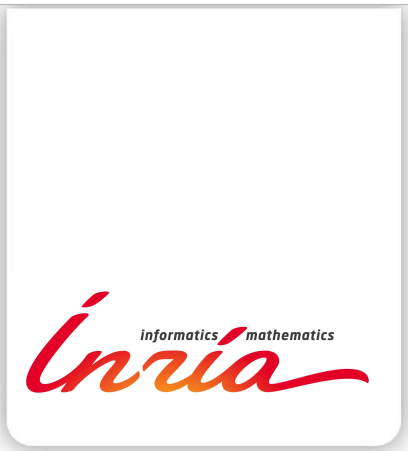
RESEARCH REPORT

N° 8485

February 2014

Project-Team REGAL

ISSN 0249-6399 ISRN INRIA/RR--8485--FR+ENG



Incinerator – Eliminating Stale References in Dynamic OSGi Applications

Koutheir Attouchi*, Gaël Thomas†, André Bottaro*, Julia Lawall†, Gilles Muller†

Project-Team REGAL

Research Report n° 8485 --- February 2014 --- 19 pages

Abstract: In the context of smart homes, the OSGi middleware is emerging as a standard to execute applications that collaborate together to render services. However, an application update in OSGi can introduce stale references, i.e., references to an outdated version of the application. A stale reference leads to a memory leak and to an inconsistency between the outdated version of the application and the new one. To avoid stale references, we propose Incinerator, a Java virtual machine extension that not only detects, but also eliminates stale references at runtime. Incinerator mainly runs when the garbage collector scans the object graph, so as to find stale references and set them to `null`. We have used Incinerator to detect a stale reference in the Knopflerfish OSGi framework implementation. Incinerator has a low overhead of at most 3.3% on average on the applications of the DaCapo benchmark suite. This shows that Incinerator is reasonable for use in production environments.

Key-words: smart home, OSGi, JVM, garbage collector

* Orange Labs, Grenoble, France.
† REGAL team, Laboratoire Informatique de Paris 6, France.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Incinerator – Élimination des références obsolètes dans les applications OSGi dynamiques.

Résumé :

Dans le contexte des maisons intelligentes, l'intergiciel OSGi émerge en tant que standard pour implémenter des services sous la forme d'applications communicantes. Une mise-à-jour d'application OSGi peut introduire des références obsolètes, c'est-à-dire des références à une version dépassée de l'application. Une référence obsolète crée une fuite mémoire et une inconsistance entre la version dépassée de l'application et la nouvelle version. Dans cet article, nous proposons Incinerator, une extension à la machine virtuelle Java qui détecte les références obsolètes et les élimine lors de l'exécution en les remplaçant par `null`. Incinerator s'exécute lorsque le ramasse miettes balaye le graphe d'objets. Grace à Incinerator, nous avons détecté une référence obsolète dans Knopflerfish, une des implémentations les plus connues d'OSGi. Incinerator induit un faible surcoût à l'exéctution d'environ 3,3% pour les applications de la suite de tests DaCapo. Ceci montre que l'utilisation d'Incinerator est possible en environnement de production.

Mots-clés : maison intelligente, OSGi, JVM, ramasse miette

1 Introduction

The last few years have seen the massive deployment of smart devices and services in the context of smart homes. As promoted by the operators and affiliate companies of the Home Gateway Initiative [21], multiple untrusted applications collaborate to render services. Collaboration and the mulplicity of the applications induce two requirements for the underlying framework: updating a single application should not imply a complete reboot of the system, and communication between applications that frequently exchange data has to be efficient. The OSGi framework [22] addresses both requirements. First, it deploys an application as a bundle, which can be uninstalled or updated individually. Second, it runs all bundles in a single address space, thus avoiding expensive remote procedure calls between bundles, where parameters have to be marshaled and unmarshaled [6].

Unfortunately, running all bundles in a single address space make OSGi prone to inconsistencies and memory leaks, as stated by the OSGi specification [22] and as recently demonstrated in practice by Gama et al. [9]. 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. Such an outdated reference is said to be *stale*. Using a stale reference, one may execute outdated code which can conflict with the updated bundle. We have observed that such conflicts can compromise the robustness of the system or may induce unexpected behavior for the devices controlled by the bundle. Furthermore, stale references prevent the garbage collector from reclaiming the bundle's class definitions and generated code, which amounts to a memory leak. In a system that has to be up and available for a very long time, repeated memory leaks eventually lead to random and unexpected allocation failures. Overall, the consequences of stale references may range from user annoyance, for entertainment services, to life critical issues for healthcare and security-related home services.

Avoiding stale references is challenging for the OSGi developer. From the Java programming language point of view, a stale reference is 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. To manually detect stale references, the developer must track all inter-bundle references in the source code and carefully check that these references are always kept consistent with all bundle updates. 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.

In this paper, we address the problems raised by OSGi stale references at the Java virtual machine level by proposing a bundle-aware garbage collector, named Incinerator. 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 requires 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 a consequence, no stale object remains reachable at the end of the collection and the associated memory is reclaimed by the garbage collector.

Incinerator changes the behavior of the Java virtual machine, because it nullifies references that are found to be stale. We investigate 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 is never affected by Incinerator. When a bundle does not release

some stale reference, our design choice is to minimize the impact of nullification while ensuring that the memory is released. Three situations can occur: (i) the stale reference is never used; nullifying it has no impact on the other bundles, (ii) the stale reference is only accessed as part of a cleanup operation, *i.e.*, a finalize method; Incinerator executes this cleanup operation in order to avoid other kinds of leaks, (iii) the stale reference is used elsewhere, either an access to or a synchronization on the stale object; 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 developer track down the bug, by making it visible.[1] If a thread is blocked while waiting for a synchronization on the stale reference, Incinerator also unblocks the thread, in order to prevent leaking of the thread and its reachable objects.

We have prototyped Incinerator in J3, a Java virtual machine based on VMKit [10]. Incinerator modifies the MMTk ''Mark-Sweep'' garbage collector [3] included in VMKit. Implementing Incinerator required the addition of 650 lines of code to J3. We have used Incinerator together with Knopflerfish 3.5.0 [15], one of the main OSGi implementations. Preparing Knopflerfish for Incinerator required modifying only 10 lines of code, to notify the Java virtual machine when a bundle is installed and uninstalled.

We have evaluated the impact of Incinerator both in terms of the increase in robustness and the performance penalty with the following experiments:

- We have designed a test suite of nine micro-benchmarks that cover the possible sources of memory leaks caused by stale references in OSGi. Incinerator identifies the stale references in all cases and prevents memory leaks, while they occur with a standard JVM.
- We have evaluated the overhead incurred by Incinerator using the DaCapo benchmark suite [4] which covers a wide range of application behaviors. The average overhead remains below 1.2% on a high-end desktop machine and below 3.3% on a smart-home PC, which shows that Incinerator is reasonable for production environments.
- We have used Incinerator to find a bug in a legacy OSGi bundle, the widely used `HTTP-Server` bundle of Knopflerfish. We sent a bug report and a patch, which have been accepted by the Knopflerfish maintainers.

The rest of the paper is organized as follows. Section 2 describes Java class loaders and the problem of memory leaks. Section 3 presents the design and implementation of Incinerator. Section 4 evaluates the benefits of Incinerator. Section 5 presents an overview of related work, and Section 6 concludes.

2 Background

In this section, we first describe the behavior of Java class loaders. Then, we present how they are used to implement OSGi bundles and how uninstalling or updating a bundle may lead to a stale reference.

2.1 Java class loaders

A Java virtual machine [16] (JVM) executes bytecode instructions that belong to a Java class. A class is loaded individually on-demand using a *Java class loader* which is a Java object whose

[1] It would be helpful to throw an exception that is specific to stale reference access, but this is not yet supported by our implementation.

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.

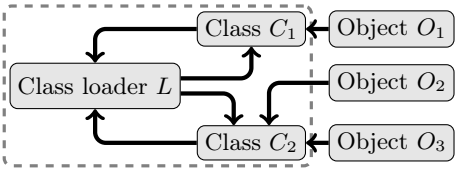


Figure 1 – References graph between class loaders, classes, and objects.

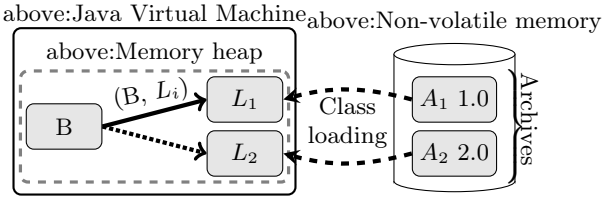


Figure 2 – OSGi bundle update. Each archive A_i is loaded using a separate class loader L_i. Updating bundle B from version 1.0 to version 2.0 implies creating L_2, then loading A_2 using L_2, and then associating B with L_2 instead of L_1.

purpose is to return a Java class given a fully qualified class name. As shown in Figure 1, 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 has loaded.

The garbage collector can collect an object when it becomes unreachable [14]. Collecting a Java class C_i implies reclaiming the bytecode of the methods of C_i, their generated machine code, and associated information e.g., C_i's fully qualified name, hierarchy, and fields. A class loader L that has loaded classes $C_1, ..., C_N$ can only be collected when the graph of objects $\{L, C_1, ..., C_N\}$ becomes unreachable, which implies that, for each C_i, all objects of C_i are unreachable.

2.2 Bundles and stale references

A *bundle* is a Java object created by the OSGi framework from an *archive*, which represents an OSGi deployment unit, holding Java bytecode and OSGi-specific metadata including the symbolic name, vendor and version. A bundle is identified by a *bundle ID* that is unique during the lifetime of the OSGi framework. Each bundle has its own class loader and contains a reference to this class loader.

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. The class loader is subsequently considered to be *stale*, and referencing or accessing it is regarded as an invalid operation.

Making the class loader stale makes all the classes loaded by the class loader stale, and all objects of those classes. Accordingly, a reference is said to be *stale* if it references the class loader of the uninstalled bundle, a class loaded by this class loader, or an object that instantiates one of these classes.

When a bundle is *updated*, OSGi first uninstalls it, making the previous class loader stale, and then creates a new class loader that loads the new bundle archive. Finally, OSGi updates the bundle so that it refers to the new class loader. The update process is illustrated in Figure 2.

3 Incinerator design and implementation

The goal of Incinerator is to eliminate stale references by setting them to `null`. For this, 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 have chosen to design Incinerator by extending it. 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. Finally, we introduce the most important implementation details.

3.1 Stale class loaders

As described in Section 2.2, a reference is stale when the referenced object is an instantiation of a class that belongs to a stale class loader. However, in Java, it is not possible to distinguish a stale class loader, that happens to be reachable by some object in the system, from a non-stale one. To make explicit the notion of class loader staleness, we add a *stale* flag to the internal representation of a class loader in the JVM. This flag is cleared when the class loader is created and is set when the associated bundle is uninstalled or updated by OSGi. Since the flag is internal to the JVM, we also introduce a native Java method to make the flag-setting operation accessible to the OSGi framework.

3.2 Stale references and synchronization

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 JVMs allocate a monitor structure that is separate from the object and is managed explicitly [2]. 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 remanining blocked threads.

3.3 Stale references and finalization

In Java, a `finalize()` method defines clean up code that is executed exactly once by the garbage collector before the memory associated with the object is reclaimed. If a `finalize()` method accesses a stale reference that was nullified by Incinerator, then it encounters a `NullPointerException` and is 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. Indeed, when the bundle is uninstalled or updated, its finalizable objects become unreachable. These finalizable objects often use intra-bundle references, which are thus stale, and nullifying these references prevents the execution of the `finalize()` methods.

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, *i.e.,* objects that implement a `finalize()` method for which the `finalize()` method has not yet been called. The garbage collector does not reclaim the memory of an unreachable finalizable object at this time because the object can become reachable again during the execution of its `finalize()` method, e.g., by storing a reference to itself in a reachable object or in a static variable. Instead, during the second phase, the garbage collector marks 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 executed only once. Finally, after the second phase, the garbage collector executes the `finalize()` method. 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 to be unreachable.

Incinerator is designed with the goal of preventing resource leaks. For this, Incinerator tries to allow `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.

For a finalizable object that is reachable at the end of a collection, 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 any 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 a collection cycle. Incinerator uses the following three-step algorithm, illustrated in Figure 3:

1. The first step is performed when the garbage collector initially scans the heap. At this time, Incinerator does not know which objects are finalizable and unreachable. For each stale reference scanned, Incinerator (i) saves the location of the reference in an internal list, and (ii) aborts the scanning of the referenced object. As a consequence, the stale object and its sub-graph are not marked as reachable.

2. The second step is performed when the garbage collector marks as reachable each unreachable finalizable object and its reachable sub-graph. Incinerator leverages the scanning of these objects performed by the garbage collector. For each stale reference scanned, Incinerator

removes from the internal list all the locations that reference the same stale referenced object. Moreover, in this case, Incinerator does not abort the scanning of the stale reference, and consequently lets the garbage collector scan its reachable sub-graph. After the garbage collector has marked all the unreachable finalizable objects, the internal list of Incinerator only contains locations of references that are both (i) stale, and (ii) not reachable from an unreachable finalizable object.

3. The third step is performed at the end of the collection cycle. Incinerator nullifies all of the remaining stale reference locations in its internal list. At the same time, it handles the problem of monitors discussed in Section 3.2.

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 the 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 flag to the class loader to indicate whether its stale objects has already survived a collection cycle. Incinerator only delays the nullification of the stale references to the stale objects during the first collection and not during the second one.

3.4 Implementation

The Incinerator prototype is based on the J3/VMKit [10] experimental JVM. Implementing Incinerator requires 650 lines of C++, and modifying approximately 20 lines in the JVM (the scan and the termination functions of the garbage collector, and the lock acquire function of the monitors). This suggests that Incinerator should be relatively easy to port to a different JVM.

3.4.1 JVM changes.

We have seen that 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 presented in the previous section. Incinerator also 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, 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 if it is certain that all stale references have been eliminated, i.e., if Incinerator did not find any stale references that are reachable from a finalizable object. Incinerator checks this flag at the beginning of a collection and appropriatly install the original or the Incincerator scan function. It is also checked at the end of a collection to know whether stale reference locations have to be nullified.

3.4.2 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

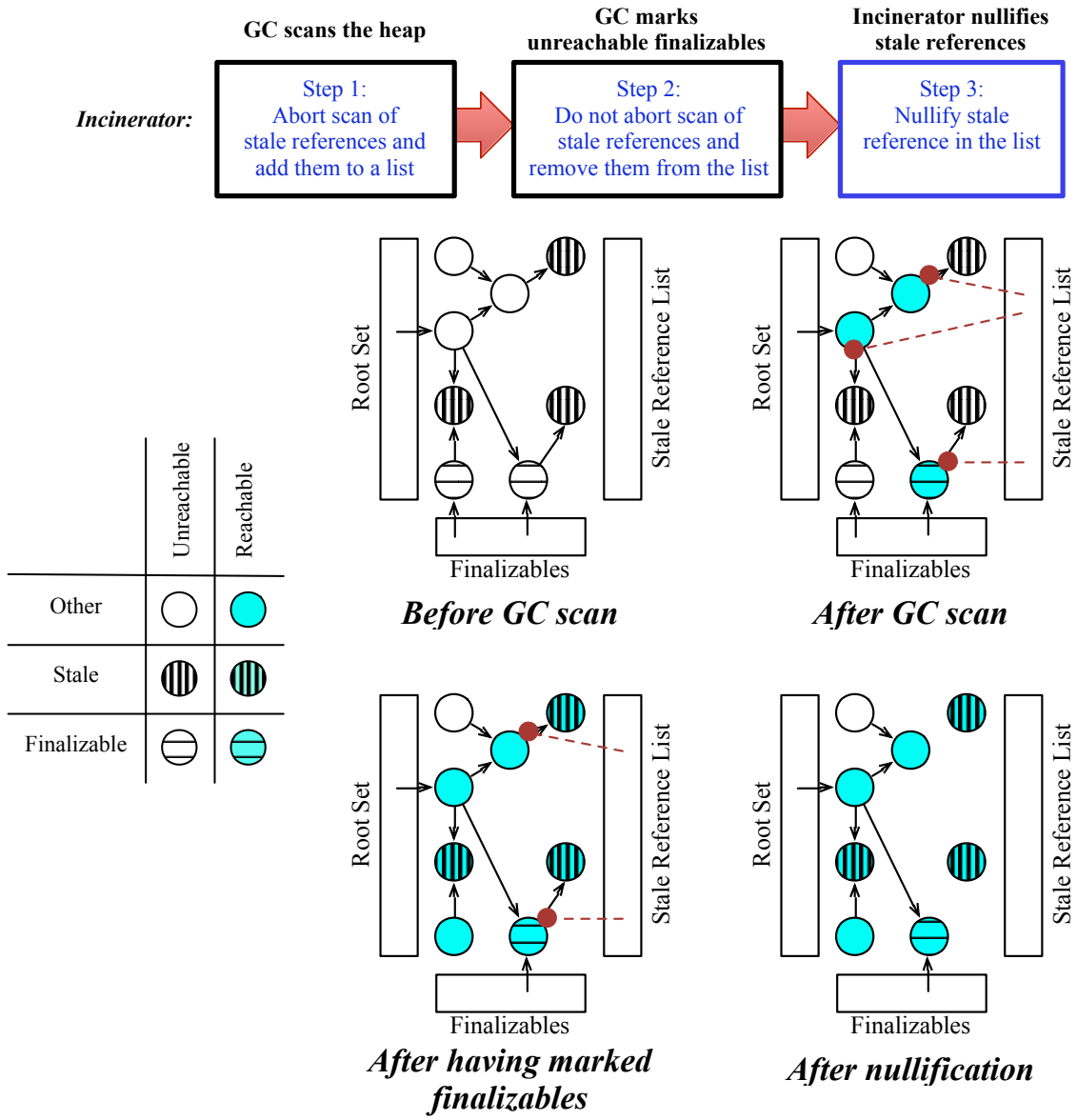


Figure 3 – Incinerator algorithm.

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 only requires 10 lines of additional Java code.

3.4.3 Modifications to the Just-In-Time compiler.

The Java language specification [12] 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 possibility of an infinite loop when using Incinerator, because Incinerator can nullify the reference to the object containing the monitor, causing the monitor release operation itself to raise a `NullPointerException`, which triggers again the execution of the same exception handler.

To avoid this issue, we have modified the Just-In-Time compiler so that the code generated for a lock release 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 ensures that the reference given to a synchronized block is never touched between an acquire and a release instruction. As a consequence, the argument of the release instruction can never be `null` as, if it were, the preceding acquire instruction would have thrown the `NullPointerException`. However, our workaround is incompatible with the Java specification and could change the behavior of programs written directly in Java byte code or generated in an ad-hoc fashion.

4 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 nine micro-benchmarks that we have designed to cover the possible sources of stale references. This test suite is used to compare the behavior of Incinerator with Service Coroner [9], a tool that instruments the OSGI reference managing calls and that detects stale references by analyzing the object references from 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 [15]. Finally, we study the performance overhead of Incinerator by running the DaCapo 2006 benchmark, which is representative of standard Java applications.

We have executed all of the benchmarks on two computers, both of which run Debian 6.0 with a Linux 2.6.32-i386 kernel: (i) a low-end computer with a 927 Mhz Intel Pentium III processor, 248 megabytes of RAM and 4 gigabytes of swap space, which has performance comparable to that of a typical system in a smart home environment, (ii) a high-end computer having two 2.66 Ghz Intel Xeon X5650 6 core processors, 12 gigabytes of RAM and no swap space. In the former case, the swap space is necessary, because J3 requires at least 1 gigabyte of address space to run the DaCapo benchmark. Service Coroner was executed using Knopflerfish 3.5.0 on top of Sun JVM 6.

4.1 Stale reference micro-benchmarks

In order to assess the scope of the stale reference problem, we have designed a test suite of nine 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 4 illustrates the bundle configuration used in our scenario descriptions. We consider three bundles A, B, C, and two objects. X is an object created by A and Y an object created by B. The OSGi framework manages a reference to the object X but not to the object Y. Stale references will appear in the faulty bundle C.

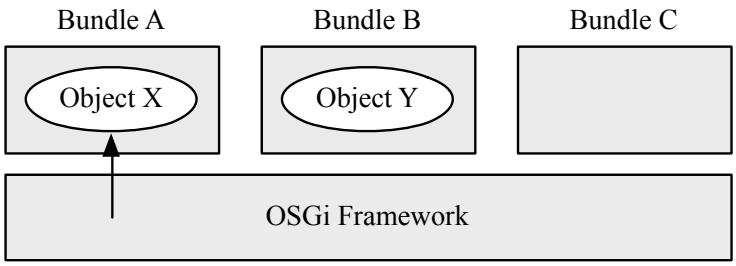


Figure 4 – Bundle configuration used in micro-benchmarks scenarios.

4.1.1 OSGi Visibility.

OSGi visibility refers to whether the reference is managed by the OSGi framework which means that a bundle must call the OSGi framework to obtain a reference to the object. By instrumenting the OSGi calls, one could keep track of the references given to bundles. Service Coroner uses this technique for detecting stale references. We have developed two scenarios illustrating whether the reference is visible to OSGi. In **Scenario 1**, the bundle C retains a reference to X which is managed by OSGi. The reference is turned stale by uninstalling A. In **Scenario 2**, the bundle C retains a reference to Y which is not managed by OSGi. The reference is turned stale by uninstalling B.

4.1.2 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 have designed four scenarios to check that Incinerator finds stale references in all possible kinds of locations. A reference to Y is retained by the bundle C, respectively, in a local variable (**Scenario 3**), in a global variable (**Scenario 4**), in an object field (**Scenario 5**), and in a thread-local variable (**Scenario 6**). In all scenarios, the reference is made stale by uninstalling B.

4.1.3 Synchronization.

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 has woken up. **Scenario 7** illustrates this situation by having two threads created by C synchronizing on Y. The references to Y are turned stale by uninstalling B.

4.1.4 Finalization.

Incinerator tries to allow `finalize()` methods to run without introducing null pointer exceptions by not nullifying stale references reachable by a finalizable object. To check the possible cases, we have defined two scenarios. In **Scenario 8**, the bundle C retains a reference to Y and Y is finalizable with a `finalize()` method that does not access memory. The object Y is made stale by uninstalling B. In **Scenario 9**, the bundle C retains a reference to Y and Y is finalizable

	Scenario 1	Scenarios 2 – 9
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	

Figure 5 – Micro-benchmark execution with standard JVMs, Service Coroner and Incinerator.

with a `finalize()` method that uses an object Y' belonging to B. Y and Y' are made stale by uninstalling B.

4.1.5 Results and conclusions.

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

Figure 5 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 1**, thanks to the instrumentation of the OSGi calls transmiting references to the 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 2** to **9**. Finally, Incinerator detects and eliminates all the stale references illustrated in the nine scenarios. In particular, Incinerator handles correctly the case of the stale references used for synchronization (**Scenario 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`, the holder of the lock when it tries to re-acquire the lock and the blocked thread in the lock-acquiring method. Incinerator also handles correctly the two cases of stale references used by a finalizable object (**Scenarios 8** and **9**), 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.

4.2 Bundle conflicts

To demonstrate the risk of physical hazards and data corruption that can be caused by stale references, we have prototyped an alarm application that is representative of a typical smart home system. Figure 6 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 level of gas. The application accesses physical devices via two driver bundles, `SirenDriver`, `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`.

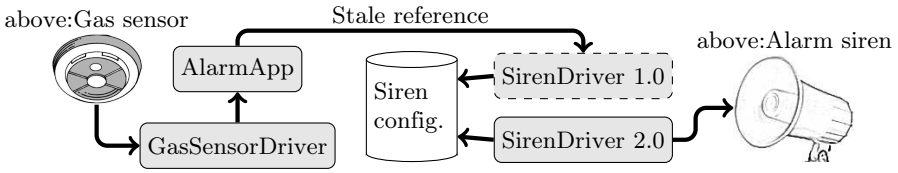


Figure 6 – Hardware and software configuration of the Alarm controller application.

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-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. First, the memory used by the JVM increased. We executed a garbage collection and observed, via the debug logs, that `SirenDriver` 1.0 was not collected, thus leaking memory. Second, 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. Third, after simulating a gas leak to the gas sensor, we observe 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.

4.3 Memory leaks

To investigate the memory leaks caused by stale references in quantitative terms, we repeat **Scenario 1** (Section 4.1) multiple times so as to create a large number of stale references. In this experiment, a bundle C retains a reference to X (see Figure 4). Each time we update A, C both keeps its old reference and obtains a new reference to the new X provided by the new A. The bundle A is a small unit test bundle with 150 lines of Java code distributed over 3 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 `OutOfMemoryException` exceptions on our low-end test machine. Incinerator, however, continues to use the same amount

of memory. These results show that, even stale references into small bundles, such as A, may leak a significant amount of memory.

4.4 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 only requires an old 1.4 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 have 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 they 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 (see Figure 4.4) cause a loss of 6 megabytes of memory on each bundle update. Indeed, the HTTP-Server bundle contains 46 classes, which in all 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 stop. Incinerator thus simultaneously solves the two leaks: the leaked thread is stopped and the memory of the stale bundle is reclaimed.

We sent a bug description and a patch to the developer community.[2] 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.

4.5 Performance benchmarks

In order to measure the performance impact of Incinerator on real Java applications, we ran the DaCapo benchmark suite [4] on J3 and on Incinerator. The DaCapo 2006 benchmark suite includes nine real Java applications[3] stressing many JVM subsystems.

This evaluation assesses the minimal impact of Incinerator, when there are no bundle updates, because the DaCapo 2006 applications do not define bundles. 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 is stale.

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 7 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 8 shows that J3 performs better

[2] http://sourceforge.net/p/gatespace/bugs/175/
[3] http://www.dacapobench.org/benchmarks.html

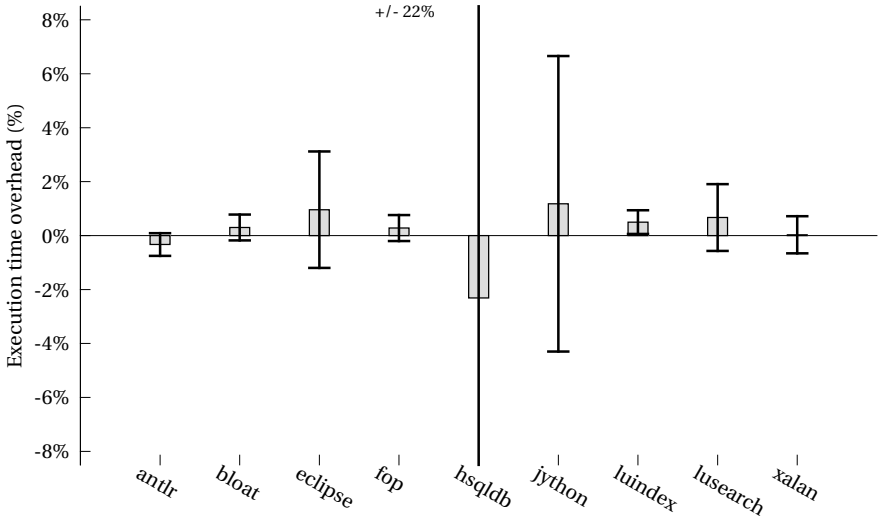


Figure 7 – Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a low-end computer. *hsqldb* has a standard deviation of 22%, truncated here for clarity.

than Incinerator in 4 out of 9 applications, with a worst slowdown of 1.2%. But, as indicated by the standard deviations on Figure 7 and Figure 8, 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 [17].

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

5 Related work

We first compare the management of stale references with the management of weak references already found in Java. We then discuss the *security* and *performance* issues of existing application frameworks and then we illustrate how these frameworks deal with stale references. We show that, rather than *eliminating* stale references, existing frameworks only tend to *avoid* them or to *detect* them.

Weak references. The Java specification defines a *weak reference*[4] as a special 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 nullifies all the remaining weak references and collects the referenced object. Weak references are not adequate in our context because an inter-bundle reference needs to ensure that the referenced bundle stays alive, at least until the bundle is uninstalled or updated. That is, an inter-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.

[4] http://docs.oracle.com/javase/6/docs/api/java/lang/ref/package-summary.html #reachability

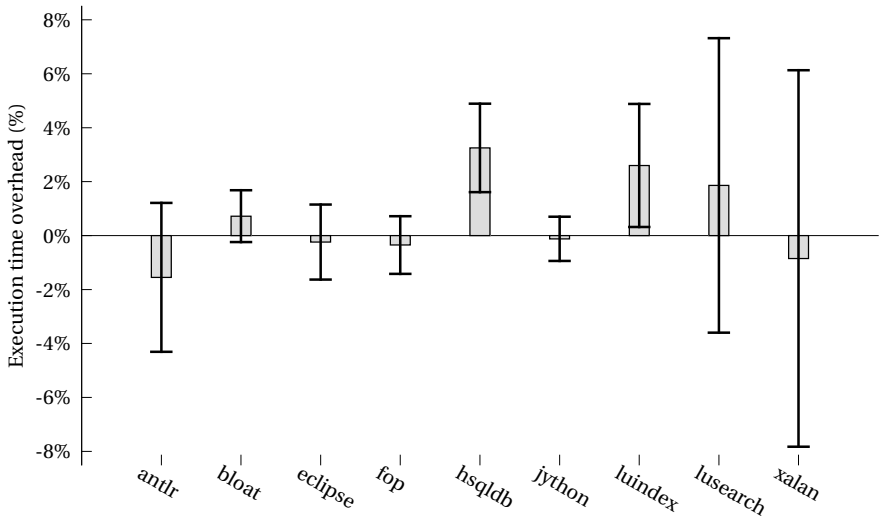


Figure 8 – Average execution time overhead of DaCapo 2006 benchmark applications between J3 and Incinerator when executed on a high-end computer.

Security versus Performance. In environments where multiple applications run simultaneously, the management of data isolation between applications ranges from full isolation to full sharing [19]. Full isolation generally implies strong security guarantees but complicates communication between entities, and degrades performance and ease of use. Existing application frameworks tend to choose to focus on only one of security or performance, at the expense of the other.

The Multitasking Virtual machine [6], Android [20], Singularity [8, 13] and KaffeOS [1] 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, but by preventing process to share objects. In these frameworks, communication between applications involves remote procedure calls. Remote procedure calls require data marshaling, which adds a considerable overhead, especially as compared to direct procedure calls inside a single address space.

OSGi [22] trades security for performance, by running all applications in one memory address space, thus enabling efficient direct method invocation between applications, a.k.a., bundles. The drawbacks of this design is weaker isolation between applications and the possibility that references become stale when bundles are updated or uninstalled. The problem of isolation was studied by Geoffray *et al.* with I-JVM [11] by using resource accounting techniques inspired by JRes [7]. The work on I-JVM shows that a JVM variant can provide an acceptable level isolation between the OSGi bundles, but it does not solve the problem of stale references.

Incommunicado. Incommunicado [18] proposes an API enabling communication between isolates (e.g., processes) for the Multitasking Virtual machine [6]. It defines a distributed object system that leverages the shared memory between the isolates to optimize the communication cost. Incommunicado defines two kinds of objects: distributed objects called portals, which are sent by reference during a remote call, and other objects, which are deeply copied during a remote call.

Exchanging portal references could lead to stale references. Indeed, an isolate could acquire

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

Figure 9 – Stale references found by Service Coroner.

a reference to a portal allocated by another isolate, preventing this last from being collected. Incommunicado avoids stale references by sending a proxy to a portal instead of its Java reference. The proxy holds a weak reference to the portal, allowing the garbage collector to reclaim the memory of the portal and its isolate, even if the portal is still referenced by another isolate. As already discussed in a previous paragraph, weak references are not adequate in the context of OSGi. Moreover, except for portals, other objects are deeply copied, which drastically increases the cost of inter-process communication.

Furthermore, Incommunicado defers the termination of an isolate if some cross-isolate calls are pending into it. Termination is performed once the calls return, which can take an indefinite amount of time. This can cause Denial of Service (DoS) attacks in the form of calls that never return, preventing the collection of isolates, which causes memory leaks.

Service Coroner. Service Coroner [9] is a profiling tool that detects stale references in OSGi by periodically dumping all of the memory and analyzing the reference usage graph. Such a memory dump induces a high CPU, memory and disk overhead when the dumps occur, which limits the usability of this tool to testing environments. Experiments made by the authors of Service Coroner have shown that stale references exist in several applications, as shown in Figure 9.

However, as presented in Section 4, Service Coroner does not detect all the possible stale references. In OSGi a bundle can register a Java object in a repository. Such an object is then called a service. A services can be seen as an entry point to a bundle. Service Coroner is only able to detect stale services by instrumenting the OSGi method that unregister a service. If a method of the service returns a reference to an object allocated by the bundle of the service, this reference is not visible by OSGi and thus, not by Service Coroner. Moreover, Service Coroner is unable to eliminate stale references because it analyses a dump of the memory, not the memory.

The OSGi ME specification. Developed by IS2T and Orange, the *OSGi ME* specification [5] is another solution to avoid the memory leaks caused by stale references in OSGi. As for Service Coroner, OSGi ME only focuses on stale service references and thus unable to find all the possible stale references.

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 paper, 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.

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.

References

[1] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in java. In *OSDI'00*, pages 23--23. USENIX, 2000.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *PLDI'98*, pages 258--268. ACM, 1998.

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE'04*, pages 137--146. IEEE, 2004.

[4] 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 *OOPSLA'06*, pages 169--190. ACM, 2006.

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

[6] G. Czajkowski and L. Daynès. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA'01*, pages 125--138. ACM, 2001.

[7] G. Czajkowski and T. von Eicken. Jres: a resource accounting interface for java. In *OOPSLA'98*, pages 21--35. ACM, 1998.

[8] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *EuroSys'06*, pages 177--190. ACM, 2006.

[9] K. Gama and D. Donsez. Service Coroner: A diagnostic tool for locating OSGi stale references. In *SEAA 08*, pages 108--115. IEEE, 2008.

[10] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a substrate for managed runtime environments. In *VEE'10*, pages 51--62. ACM, 2010.

[11] 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 *DSN'09*, pages 544--553. IEEE, 2009.

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

[13] 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 *EuroSys'07*, pages 341--354. ACM, 2007.

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

[15] Knopflerfish web page. `http://www.knopflerfish.org/`, 2012.

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

[17] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS'09*, pages 265--276. ACM, 2009.

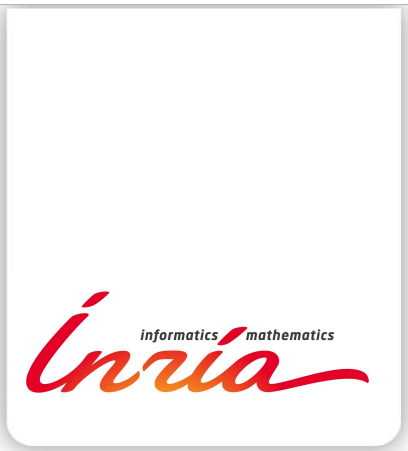
[18] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynès. Incommunicado: efficient communication for isolates. In *OOPSLA'02*, pages 262--274. ACM, 2002.

[19] J. S. Rellermeyer, S.-W. Lee, and M. Kistler. Cloud platforms and embedded computing: the operating systems of the future. In *DAC'13*, pages 1--6. ACM, 2013.

[20] 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.

[21] SWEX Group. Home gateway initiative - requirements for software modularity on the home gateway version 1.0. Technical report, SWEX Group, 2011.

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



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399