Service Coroner: A Diagnostic Tool for Locating OSGi Stale References

Kiev Gama and Didier Donsez
University of Grenoble, LIG laboratory, ADELE team
{firstname.lastname}@imag.fr

Abstract

The OSGi Services Platform provides a framework for the dynamic deployment of Java-based applications. It allows to install, to activate, to update and to uninstall application modules without the need to restart the host Java Virtual Machine. However, the mishandling of such OSGi dynamics may result in a problem described in the OSGi specification as Stale References, which happen when services from uninstalled modules are still referenced by active code. It may lead to inconsistencies in application's behavior, state and memory. Currently, there are no tools available to address this issue. This paper presents a diagnostics tool named ServiceCoroner that detects such problems. It helps developers and administrators diagnose OSGi applications running either in production or test environments. We have validated this tool on two open source applications that run on OSGi: a JavaEE application server and a multi-protocol instant messenger application. The results of the experiments show stale references in those applications.

1. Introduction

The OSGi [1] framework introduced the concept of *module system* that was missing in the Java platform. This concept tackles the problem of the "Classpath hell" encountered when applications are deployed (installed or updated) on production sites (i.e., end-user PCs or host servers). Applications can take advantage of a "hot deploy" capability, where modules (called *bundles* in OSGi nomenclature) can be added, updated or removed without restarting the JVM. In addition, each bundle is provided with its own class loader. This provides various advantages including the possibility of having independent versions of the same class, and allowing the unloading of classes when a module is updated or uninstalled.

Moreover, the OSGi specification applies service-oriented architecture principles to Java application design. The concept of service is important to decouple the application modules in order to dynamically substitute or update individual modules without affecting the entire system.

The OSGi has proved to be successful in embedded systems. Its adoption in the software industry is continuously growing as more applications tend to take advantage of its pluggable architecture. One of the well known cases is the Eclipse project [2] that since version 3.0 has migrated to the OSGi platform. A new trend in desktop applications [2] and server middlewares [3] (i.e., JOnAS, Weblogic, WebSphere) shows the growing adoption of OSGi as a modular layer for either commercial or open-source Java-based products.

Programming a modularized application that targets OSGi is a task that is apparently easy but developers must be aware of some particularities. It is critical to correctly handle framework events such as arrival (registration) and departure (unregistration) of services and bundles. Most of the time, application developers that are not experienced with OSGi dynamics provide code that may retain service references even when the providing bundles are gone. The OSGi R4 specification refers to this problem as *stale references*. Identifying stale references is not easy since current Java diagnostic tools do not handle this OSGi specific problem.

This paper presents the results of an experiment made with a custom built tool which allows the analysis and detection of stale references. The objective of the tool is not to solve the stale references problem, but to identify it and help developers and administrators of OSGi based applications ensure that bundles provide behavior well suited to OSGi dynamics. The tool was validated with the analysis of two open source applications constructed on top of OSGi: JOnAS 5.0.1 [4] and SIP Communicator [5], both of which have presented stale references after the update of some bundles.

The remainder of this paper is structured in the following order: Section 2 details OSGi and the stale references problem. Section 3 describes our tool, ServiceCoroner, and the techniques used to develop it. In section 4, the analysis of two OSGi applications is detailed. Section 5 presents related work, and at last conclusions and future work are presented in section 6.

2. OSGi and the Stale References Problem

The OSGi framework provides each bundle with a *BundleContext* object that gives access to the underlying framework. A bundle can register and retrieve services through the BundleContext. In order to retrieve a service instance in OSGi it is first necessary to know its *ServiceReference*, which is service metadata that informs what bundle provides the service, what are the service properties, and so forth. The BundleContext provides a getServiceReference method that takes the name of the service interface and an optional property filter.

A bundle that provides a service may choose to directly provide the service instance[1] or to provide a ServiceFactory that will be responsible of creating service instances. The ServiceFactory makes it possible to provide an individual service instance per client bundle. That is, if two bundles request the same service they will get different instances of the very same type.

After being loaded by OSGi, each bundle will have an individual class loader to load resources (e.g., images, text files) and classes provided by the bundle. This, combined with a class import and export policy, gives a certain level of isolation between bundles. This mechanism provides an additional namespace level making it possible to have multiple versions of classes with identical absolute names but provided by different bundles (note that this is not related with the ServiceFactory described before).

2.2. The Stale References Problem

Although OSGi provides individual class loaders per bundle, bundles are not completely isolated from each other. One bundle may use a service that is provided by another, and consequently originated from a different class loader. Whenever a bundle becomes unavailable all other bundles that use its classes must release all references to objects provided by the departed bundle. This procedure is necessary to ensure that the garbage collection will take place correctly and that no bundle will utilize inconsistent services.

Figures 1 through 3 illustrate a common scenario of life cycle events that result in a stale reference being kept. In Figure 1 normal interaction between the service provider and the service requester can be seen on Bundles 1 and 2, respectively. Bundle 2 consumes a service whose implementation is provided by an instance of the class A from Bundle 1.

If Bundle 1 is stopped, the framework will automatically notify the unregistering of all services provided by that bundle. In our scenario, Bundle 2 is supposed to release references to the service instance of class A upon the unregistration of A's corresponding service reference. This results in having no references towards objects of Bundle 1, permitting these objects to be collected by the JVM.

However, Figure 2 shows that this procedure was not performed by Bundle 2, which keeps holding a reference to a service instance from a stopped bundle (Bundle 1). It shows that Bundle 2 did not handle the departure of the service reference that provides the A service instance. Bundle 2 is still able to perform method calls on the object A that possibly can lead to inconsistent information or to an inconsistent state since such method calls are being made on an object from a stopped bundle. For example, network connections or file streams could have been closed by the stopped bundle, causing errors on the object A.

The retention of the service instance will prevent the A object of being garbage collected. As a consequence, A's class loader and all class types (java.lang.Class) it has loaded will also hang in memory.

In the continuation of the described scenario, the Bundle 1 has been updated during runtime. A new class loader has been assigned to that bundle, and a new instance of A (now from a different class loader) has been registered. Figure 3 shows that Bundle 2 kept using the stale reference and did not take into consideration the arrival of the new service provided by the updated Bundle 1.

In the described scenario the objects from old Bundle 1 (the service instance A v1.0, its class loader and all java.lang.Class instances loaded by that class loader) will only be eligible to garbage collection if Bundle 2 is stopped. However, if Bundle 2 is restarted the same situation is likely to happen again upon the same sequence of events.

This is not only limited to bundle updates. If, for example, a bundle provider of a given service instead of updated is just stopped or uninstalled, it can never be released from memory if a similar situation as in Figure 1 happens.

[1] For the sake of clarity this paper refers to service reference as the ServiceReference object, and service instance as an instance provided by the BundleContext via the getService(ServiceReference) method.

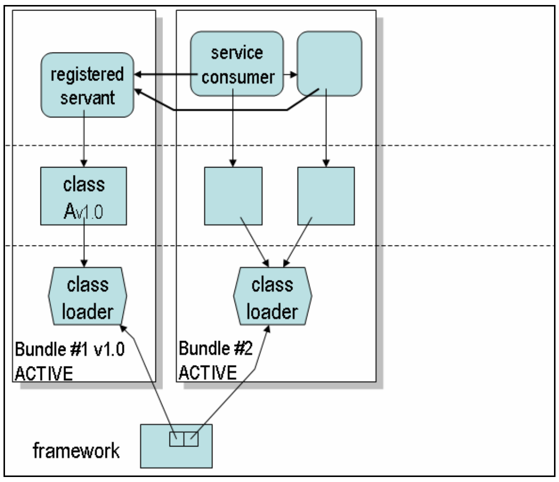


Figure 1. Normal scenario

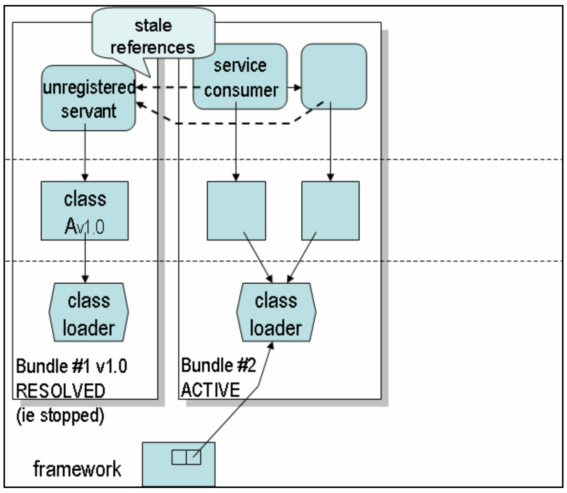


Figure 2. Reference is now stale

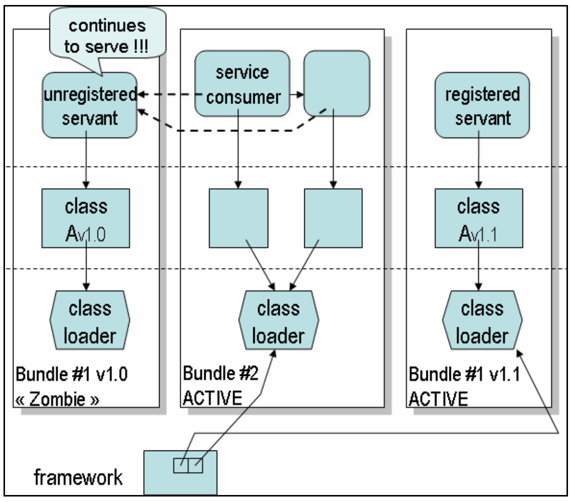


Figure 3. New bundle is started but not used

Bundles isolation is compromised by the bad handling of service departure. The OSGi specification indicates practices to minimize the problems but they still are error prone.

An alternative to avoid such problems is to utilize mechanisms that provide transparent handling of service location, like the ServiceTracker. However, in that approach, clients can still keep references to service instances no matter how these references were retrieved. In addition to transparent location handling, other mechanisms like OSGi declarative services, Service Binder [14], iPOJO [6] and Spring Dynamic Modules [15] deal seamlessly with the releasing of service instances and service references upon service unregistration.

These last two options are the ideal way to provide a more complete handling of service dynamics, consequently avoiding stale references. On the other hand, stale references can still be found in other patterns. In this other scenario, for example, if a bundle X retrieves a reference to a service from bundle Y. Then bundle X makes a method call on an object from a third bundle Z passing that reference ahead as a method parameter. This makes harder to prevent a possible stale reference. That third bundle may keep the reference if bundle Y (the provider) becomes unavailable. Most likely, in this scenario only bundle X is concerned with the service departure. Therefore, reference forwarding to other bundles would not be solved by the previously mentioned mechanisms.

3. The ServiceCoroner diagnostic tool

The ServiceCoroner tool relies on weaved OSGi implementations that enable to examine OSGi-targeted applications during runtime and to diagnose stale references. It enables developers and administrators diagnose OSGi applications running either in production or test environments.

Instead of changing the source code of any existing OSGi implementation we have chosen to use Aspect Oriented Programming (AOP) [7] techniques because they allow keeping our tracking code separated from actual OSGi implementation code. Since one of AOP's principles is separation of concerns, we keep the tracking code as a concern that does not "pollute" the OSGi code. By keeping that separation we have just a minimal effort of weaving each OSGi framework implementation to achieve tool portability across them.

A few frequent patterns of stale references to be analyzed in OSGi have been identified: retention of service instances and service references from stopped bundles; services that retain an ordinary object reference (received as a parameter in a normal method call) from a client bundle that has been stopped; unfinished threads created by bundles that are stopped.

Currently, the tool deals only the first presented pattern. Work is already in progress to also identify the threads pattern but results are not precise yet. The next subsections describe in more detail our diagnostic tool.

3.1. Method interception with AOP

The Service Coroner tracks each service reference object and the service instances that bundles provide. Upon services unregistration it is possible to verify if the services that are supposed to be garbage collected have really being removed from memory.

Since the current OSGi implementations do not provide enough information to track its services in a manner that stale references can be identified, the implementation of this tool adds that functionality to the platform using automated mechanisms, eliminating the need to manually modify the OSGi implementation. This has been achieved using AspectJ [8] to provide interception of method calls in the OSGi framework code. AspectJ provides an AOP extension to the Java platform.

The aspects are created targeting interfaces of the OSGi API. The bytecode weaving mechanism changes some classes (which implement the target interfaces) of the target OSGi platform by adding the desired diagnostic aspects. Strategic *joinpoints* in the OSGi API were defined in order to identify where the interception of certain method calls should be done. A build process first compiles the ServiceCoroner, and then performs the weaving of the aspects into a target OSGi implementation adding calls to the ServiceCoroner API.

The tracking of garbage collection of objects was possible by utilizing Java Weak References, which are a special type of object reference that is treated differently by the Garbage Collector, allowing the application code to know if an object allocation has been reclaimed or not. The aspects weaved into the framework provided a way to inject calls to the ServiceCoroner inside the framework code. Among other tasks, those calls would create weak references to track service instances and service references.

3.2. Diagnostic Process

The process of identifying stale references and their potential causers can basically be done using two different strategies: active diagnosis or passive diagnosis.

The active process consists in forcing bundle life cycle events (e.g., stop, update, etc.) either by interacting with an OSGi command line tool or by using ad-hoc code on the ServiceCoroner scripting console. Life cycle events may be applied to bundles randomly chosen or to a given range of bundles. If the code does not handle service dynamics appropriately, stale references can be easily found. This approach usually leads to faster results than passive diagnosis. Such empirical usage of this approach eventually leads to more refined test cases.

The passive process requires a longer observation of service arrivals,departures and life cycle events. The objective of this process is to observe the life cycle changes without directly interfering. Normal interaction is done in the application. Life cycle events result from the actual administration tasks such as the deployment or update of modules consistent with the application usage. If necessary, a production environment could have its deployed OSGi framework temporarily replaced by the ServiceCoroner weaved version in order to provide an actual diagnosis and find possible service retention bottlenecks.

The drawback of the first approach is that one can not be sure that a simulation of the application's real behavior is being done. The second approach may take longer to get significant results but it can provide more accuracy related to the actual behavior of the OSGi application. Moreover, while the active process is only suitable for a test environment the passive approach can perform diagnosis either on a test or on a production environment.

3.3. Analyzing Results

By inspecting all information utilizing the tool's GUI, it is easy to identify stale references when a service reference that has been unregistered still appears in the list with its number of active servants greater than zero or with the garbage collected status as false. A built-in query is utilized by the GUI to show a list of all stale references found.

The ServiceCoroner may show inconsistent information immediately after a bundle has been stopped. Since it depends on weak references, the tool would have to wait until the next garbage collection takes place in order to display consistent information. Garbage collection (GC) may be called through the tool's GUI which simply performs a call to Java's System.gc() method. This will not necessarily imply in an immediate execution of the GC as it may vary in each JVM implementation. However, in the environment of our experiment, manual calls to the GC via our GUI presented a fast response. We could verify that by doing the following steps: (1) stop a range of bundles; (2) verify the number of stale references; (3) perform the call to perform GC; (4) verify again the number of stale references. The number of stale references in step 4 was always smaller than in step 2.

By using our tool it is also possible to identify the potential retainers of service instances. This feature is not yet automatic since the resultant data is just an intermediary result that still needs to be analyzed by the user. Currently it lists all referrers to all instances of a given type. For example, if the referrers of ServiceX from Figure 2 are analyzed, the result set would display not only the objects from Bundle B that refer to the particular ServiceX instance but also all the objects that refer to any ServiceX instance, including the instances from Service X' of Bundle A'. In the current version the queries on memory are based on the type rather than the single object.

In order to perform such detailed inspection on the memory allocation tree, the ServiceCoroner tool utilizes the jhat and jmap tools provided in the Java 6 SDK. Jmap allows making dumps of memory with information about heap details. Jhat is another tool that is able to read jmap memory dump files and perform queries on it.

The object referrer queries are done on an instance of a jhat engine running on the same JVM as the ServiceCoroner. Query results are transformed into ServiceCoroner API objects.

3.4. Portability and performance

The weaving process was validated on the three main OSGi R4 implementations: Apache Felix [9] version 1.0, Knopflerfish [10] version 2.0.1 and Equinox [11] version 3.2.0. Initial tests on Equinox v. 3.3.0 were not successful to its default utilization of signed jars. Since the bytecode weaving changes the affected class files, the corresponding class hashes stored in the jar manifest would become invalid. During framework startup, this resulted in a validation error while attempting to load the framework jar file. The process worked in Equinox v. 3.3.0 only if we removed all security information from the jar and then performed the weaving.

The ServiceCoroner was implemented independent of any particular OSGi implementation. The coupling exists only to the framework API, which is common to all implementations. Thus, the construction of the tool was achieved without needing to change or recompile the source code of any OSGi implementation. Only the bytecode needed to be changed by applying weaving techniques provided by AspectJ.

Even though no performance changes were measured, the cost of method interception appears to have a non significant impact. The interception is made on some interactions of the bundles with their BundleContext object, which are not very frequent.

An optional tool feature that allows a fine grained analysis of object referrers depends on the jhat and jmap experimental tools that are shipped as part of Sun's JDK 6. The usage of those tools is then limited to the usage of that JDK version. The usage of jhat has memory usage implications as it allocates a large amount of memory to load the memory dumps. Exhaustive utilization of that feature leads to OutOfMemory errors.

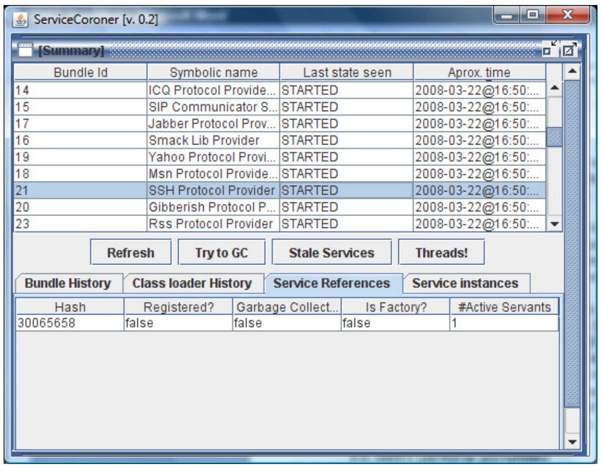
3.5. Graphical User Tools

ServiceCoroner provides two graphical user tools to help the developer to inspect a running OSGi platform.

The first one is displayed upon execution sharing the same JVM of the inspected application, as shown in figure 4. The tool is able to provide information per bundle information regarding class loaders, service references and service instances.

In addition to detailed service references information, the GUI provides a scripting console to write and execute ad-hoc code directly into the platform. This is possible through an OSGi BundleContext instance that is provided as a built-in variable available to the console.

However, if a server needs to be tested, the built-in GUI is not appropriate. A remotely enabled tool is more adequate to such task.

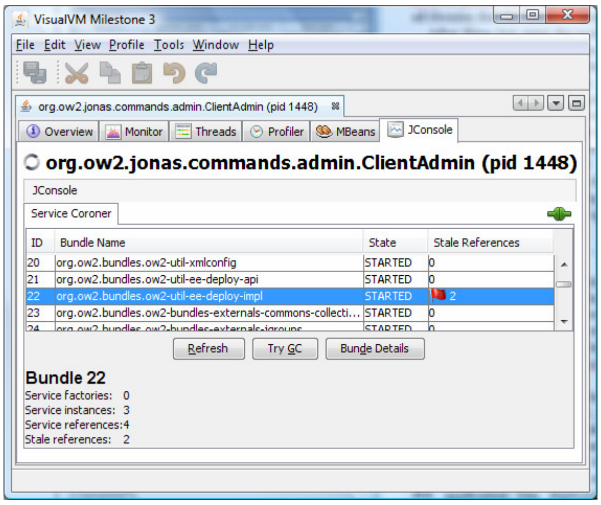


Bundle Id	Symbolic name	Last state seen	Aprox. time
14	ICQ Protocol Provide...	STARTED	2008-03-22@16:50:...
15	SIP Communicator S...	STARTED	2008-03-22@16:50:...
17	Jabber Protocol Prov...	STARTED	2008-03-22@16:50:...
16	Smack Lib Provider	STARTED	2008-03-22@16:50:...
19	Yahoo Protocol Provi...	STARTED	2008-03-22@16:50:...
18	Msn Protocol Provide...	STARTED	2008-03-22@16:50:...
21	SSH Protocol Provider	STARTED	2008-03-22@16:50:...
20	Gibberish Protocol P...	STARTED	2008-03-22@16:50:...
23	Rss Protocol Provider	STARTED	2008-03-22@16:50:...

Hash	Registered?	Garbage Collect...	Is Factory?	#Active Servants
30065658	false	false	false	1

Figure 4. ServiceCoroner standalone GUI

A second type of GUI enables the diagnostic on a remote OSGi platform. The ServiceCoroner registers a JMX [12] manageable bean (MBean), the tool allows to diagnose a remotely deployed application that utilizes the ServiceCoroner tool and a properly weaved OSGi framework. A custom JConsole plugin has been developed to display ServiceCoroner information provided by its custom MBean in the standard JConsole 6 (shipped with Java 6) and the VisualVM [13]. Figure 5 show the ServiceCoroner plugin running inside of the VisualVM.


VisualVM Milestone 3
File Edit View Profile Tools Window Help
org.ow2.jonas.commands.admin.ClientAdmin (pid 1448)
Overview
Monitor
Threads
Profiler
MBeans
JConsole
org.ow2.jonas.commands.admin.ClientAdmin (pid 1448)
JConsole
Service Coroner
Refresh
Try GC
Bunde Details

ID	Bundle Name	State	Stale References
20	org.ow2.bundles.ow2-util-xmlconfig	STARTED	0
21	org.ow2.bundles.ow2-util-ee-deploy-api	STARTED	0
22	org.ow2.bundles.ow2-util-ee-deploy-impl	STARTED	🟥 2
23	org.ow2.bundles.ow2-bundles-externals-commons-collecti...	STARTED	0
24	org.ow2.bundles.ow2-bundles-externals-jgroups	STARTED	0

Bundle 22
Service factories: 0
Service instances: 3
Service references:4
Stale references: 2

Figure 5. ServiceCoroner plugin in the Visual VM

The functionality currently provided is rather limited in terms of details and provided information when compared with the default GUI that is started by the ServiceCoroner, but permits tools such as JConsole and VisualVM to display such information

4. Validation and Analysis

The experiment performed an analysis in two applications: JOnAS and SIP Communicator.

JOnAS (Java Open Application Server) [4] is an open source implementation of the JEE specification, provided by the OW2 consortium. The version 5 of JOnAS is a bundlization of more than 1.500.000 lines of code (LOC) of the previous version. Technical services and API are packaged as OSGi bundles [3]. Most of them exchange OSGi services (unlike Geronimo which uses only the module layer). Part of the OSGi services is provided through iPOJO [6] components.

SIP Communicator [5] is a multi protocol audio/video Internet phone and instant messenger tool. Protocols plugins are delivered as separated bundles. The version 1.0 alpha 3 contains approximately 120.000 LOC packaged in 53 bundles.

Both applications use the Apache Felix [9] implementation of OSGi. JOnAS and SIP Communicator have been chosen since they are free open source applications that use OSGi's service layer. As the current state of the tool presented here focuses on the bad utilization of services, the ServiceCoroner tool would not bring useful information in applications such as Geronimo which uses only the module layer and Eclipse IDE which uses mainly its own concept of extension points.

The active process previously described was applied on the two analyses. In order to run the desired application with the tool, the OSGi implementation version used by each application was replaced by a bytecode weaved version.

During execution time, the code that intercepted some OSGi methods would add tracking information to allow the discovery of stale references.

Table 1. Experiment results

I	Application	JOnAS	SIP Communicator
II	Version	5.0.1	Alpha 3
III	OSGi implem.	Apache Felix 1.0	
IV	JVM	Sun HotSpot JVM 1.6.0u4	
V	Lines of code	More than 1.500.000[2]	120.000
VI	Total Bundles	86	53
VII	No. of Bundles with Stale Refs.	4	17
VIII	Initial number of Serv. Refs.	82	30
IX	No. of Stale Refs. found	7	19
X	Ratio (IX/VIII)	8,5 %	63 %

A custom script for the experiment was run in the ServiceCoroner scripting console. The script is responsible for calling the update method in an interval of bundles that provide services. After executing the script the tool could analyze which bundles had unregistered service references and service instances kept by other bundles. The data was ensured to be

[2] JOnAS code base is only 400KLOC according to ohloh.net. However, JOnAS had many runtime dependencies to other FLOSS projects (e.g. TomCat, Jetty, EasyBeans, Axis, JacORB, Medor, Joram). As a comparison, similar JEE servers such as Glassfish and JBoss contain 3MLOC and 1.2MLOC, respectively.

collected after the garbage collection, according to the process described in section 3.3.

Both tests described below have been performed on a Sun Hotspot JVM version 1.6 update 4.

Table 1 provides a summary with general information about the results of the experiment performed by the ServiceCoroner tool on the two target applications.

SIP Communicator errors seemed to be more frequent mostly due to an erroneous coding pattern that we have identified. By analyzing memory dumps it could be found that services were being statically referenced by class variables that did not release them upon service departure. Services retention in JOnAS was mostly caused, directly and indirectly, by JMX [12] related bundles.

The tool can quickly identify the objects retained as stale references, but limitations previously mentioned retard the process of detecting the causers of that retention.

5. Related Work

Several approaches exist to tackle the stale reference problems, like component based approaches, formal analysis and isolation.

Dynamic service-oriented component models such as ServiceBinder [14], OSGi R4 Declarative Services, iPOJO [6], and Spring DM [15] provide a component-based approach. These component models ease the development by taking care of listening to service registration/deregistration and automatically handling business logic that gets and releases references to service instances. However this does not guarantee that stale references will be vanished. Bad programming practices like reference forwarding to other bundles may lead to stale references as well.

Another stale references detection approach [16] uses a formal model to perform the analysis. That approach utilizes a third party special JVM that provides an explicit model checker. An OSGi tailored formal model is created (based on the Knopflerfish [10] implementation) and analyzed by the model checker. However, there is a serious limitation in regards to the size of applications that can be analyzed, since they are limited to about 10.000 lines of code and we are interested in large applications, like JOnAS. Besides that, the model is also tied to a specific OSGi implementation (Knopflerfish).

Other application models provide some level of isolation between components. MIDlets can be isolated between them and since MIDP specification 3.0 [17] they can also communicate via low level stream channels (Inter-MIDlet Communication). JavaCard [18] isolation between applications is done via object spaces called contexts. Communication between contexts is possible but security is enforced by an applet firewall. In Java Personal Basis Profile [19], which is largely utilized in Java TV, the Inter-Xlet Communication Model allows components executing on the same JVM to exchange objects across class loaders.

A more elaborate mechanism is provided in the JSR 121 [20]. It presents the concept of isolates. An isolate can be seen as an application unit. Applications are isolated concerning object reachability but they can share resources like runtime libraries. Communication between isolates is possible through RMI mechanisms.

Microsoft provides consolidated isolation solutions in the .NET platform and on Singularity [21]. The former provides the concept of application domain, which works as a lightweight process space that is completely isolated from the other application domains that execute on the same virtual machine. An application domain may be purged from memory without affecting the other. Communication is possible between domains but it implies in a proxy based approach using RPC mechanisms. A research OS written in managed code, called Singularity, provides strong isolation of processes. The Software Isolated Processes (SIP) run on the same address space. This gives the advantage of no process switching and isolation enforced in terms of software. Although isolated from each other, processes can communicate via contract-based channels.

However, such isolation approaches would require a deep refactoring of the API in order to provide strategies for marshalling (e.g.: defining interfaces extending java.rmi.Remote, providing serializable parameters, etc). Therefore it has a serious impact on performance when compared to the direct servant object access used in OSGi technology.

Existing tools such as [22] and [23] provide, among other features, fine grained profiling and memory analysis which enable the detection of memory leaks. However they do not address OSGi specific issues – such as the stale references that we have described– that could be unnoticed by those tools.

Our approach provides a diagnosis based on the observation of a running application. An application can be deployed in a production environment (i.e. in-vivo) and our tool is able to work without disturbing the application, having no impact on the overall performance and the bundles lifecycle while the analysis is passive.

6. Conclusion and Future Work

The dynamicity on the OSGi Services Platform may lead to memory retention problems when developers mishandle the departure of services. This article presents the ServiceCoroner tool and the analysis of the stale references that it performed on two open source applications built on OSGi: JOnAS and SIP Communicator. The tool takes advantage of bytecode weaving techniques to intercept method calls in OSGi implementations, allowing the tracking of service references and service instances.

The purpose of the ServiceCoroner is not to fix the stale references problem neither to permanently replace an OSGi framework by a bytecode weaved version in a production environment. The tool aims to help identifying stale references in OSGi based applications. The weaving process adds concerns useful for testing environments before releasing an OSGi based product, and also the possibility to detect flaws in existing applications in order to fix such problems for their later releases.

The ServiceCoroner tool utilization proved to be successful by identifying the studied problem in large applications. It is possible to automatically identify the bundles that are erroneously being referenced. The identification of potential causers of such problems is also possible, but not yet automatic.

Platform portability was achieved by the tool that was easily applied and tested among three different OSGi implementations: Apache Felix, Knopflerfish and Equinox. The build process was applied directly to binary code without any need of source code changes.

The next steps in the tool development would be the identification of other stale references patterns, such as running threads from stopped bundles. Other important feature is to extend the current MBean functionality in order to enhance remote diagnostics and management via JMX. An important improvement on existing functionality is to provide the automatic detection of the referrer bundles that hold references to unregistered service instances.

At last, we focus the Eclipse Platform, which is built on top of the OSGi framework, however it uses OSGi's service layer in a limited way. Eclipse rather uses its own plugin mechanism called extension points, which support dynamic updates of plugins but that implies in a restart of the application to avoid problems such as retention of resources from old plugins. This is a known fact for most developers that utilize the Eclipse IDE. We plan to add to the ServiceCoroner tool the capability to inspect Eclipse's extension registry and look for problems similar to those previously described as stale references patterns.

Acknowledgements

We would like to thank Walter Rudametkin and Johann Bourcier for their analysis, feedback and technical input which helped improving this paper.

7. References

[1] OSGi Services Platform. http://www.osgi.org

[2] O. Gruber et al. "The Eclipse 3.0 platform: Adopting OSGi technology". *IBM Systems Journal* 44(2), 2005, pp 289-300

[3] M. Desertot, D. Donsez and P. Lalanda, "A Dynamic Service-Oriented Implementation for Java EE Servers", *3th IEEE International Conference on Service Computing (SCC'06)* , Chicago, USA, September 2006, pp. 159-166

[4] JOnAS Open Source Java EE Application Server. http://jonas.objectweb.org

[5] SIP Communicator. http://www.sip-communicator.org

[6] C. Escoffier, R.S. Hall and P. Lalanda, "iPOJO: An extensible service-oriented component framework", *IEEE International Conference on Service Computing*, Salt Lake City, USA, 2007

[7] The AspectJ Project. http://www.eclipse.org/aspectj

[8] G. Kiczales et al. "Aspect-Oriented Programming", *European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, Finland, June 1997

[9] Apache Felix. http://felix.apache.org

[10] Knopflerfish OSGi. http://www.knopflerfish.org

[11] Equinox. http://www.eclipse.org/equinox

[12] Java Management Extensions (JMX) Specification, version 2.0. http://jcp.org/en/jsr/detail?id=255

[13] VisualVM. https://visualvm.dev.java.net/

[14] H. Cervantes and R. S. Hall, "Automating Service Dependency Management in a Service-Oriented Component Model", *6th International Workshop on Component-Based Software Engineering*, Portland, USA, 2003

[15] Spring Dynamic Modules for OSGi Service Platforms. http://www.springframework.org/osgi

[16] Z. Chen and S. Fickas, "Do No Harm: Model Checking eHome Applications", *29th International Conference on Software Engineering Workshops*, Minneapolis, USA, 2007

[17] JSR 271: Mobile Information Device Profile 3. http://jcp.org/en/jsr/detail?id=271

[18] Java Card Platform Specification 2.2.2. http://java.sun.com/products/javacard/specs.html

[19] JSR 217: Personal Basis Profile 1.1. http://jcp.org/en/jsr/detail?id=217

[20] JSR 121: Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121

[21] G. Hunt et al: An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005

[22] Netbeans Profiler. http://profiler.netbeans.org/

[23] Eclipse Test & Performance Tools Platform Project. http://www.eclipse.org/tptp/