A Practical Approach for Finding Stale References in a Dynamic Service Platform

Kiev Gama and Didier Donsez

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

Abstract. The OSGi™ Service Platform is becoming the de facto standard for modularized Java applications. The market of OSGi based COTS components is continuously growing. OSGi specific problems make it harder to validate such components. The absence of separate object spaces to isolate components may lead to inconsistencies when they are stopped. The platform cannot ensure that objects from a stopped component will no longer be referenced by active code (a problem referred by OSGi specification as stale references) leading to memory retention and inconsistencies (e.g., utilization of invalid cached data) that can introduce faults in the system. This paper classifies different patterns of stale references detailing them and presents techniques based on Aspect Oriented Programming for runtime detection of such problems. We also present a fail-stop mechanism on services to avoid propagation of incorrect results. These techniques have proven to be effective in a tool implementation that validated our study.

Keywords: OSGi, stale references, dynamic services, memory leaks, runtime diagnostics, component validation.

1 Introduction

The OSGi service platform [1] is a framework targeting the Java platform, providing a dynamic environment for the deployment of services and modules (referred as *bundles* in OSGi terminology). The OSGi architecture provides a hot deployment feature by allowing modules to be dynamically added, updated or completely removed during application execution without the need to restart the JVM. OSGi is being used in a myriad of applications (e.g., desktop and server computers, home gateways, automobiles) and is becoming the de facto standard for modularized Java applications [2] [3] [4] [5]. A milestone of OSGi's acceptance in software industry is its adoption in the Eclipse Platform [6].

Although the OSGi platform has evolved and matured in several aspects, its runtime environment does not enforce the isolation of bundles. A certain level of isolation by means of class loaders is provided by the OSGi platform, but bundles are not truly isolated from each other under a memory perspective. There are no separate object spaces between bundles that would guarantee a safe and complete removal of a bundle from the platform. Bundles may freely exchange objects, but there is no mechanism to enforce that an object will not be referenced when its bundle stops.

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 246–261, 2008.

Even with events notifying the departure of services and bundles, the current OSGi programming model is not trivial to follow and the handling of such events is error prone. Due to bundle programming flaws, object instances may be kept by a consumer bundle after the provider bundle stops. The usage of such objects leads to memory retention preventing the classes from stopped bundles to be unloaded from memory. Faulty components can be introduced in the system due to propagation of incorrect results (e.g., old or invalid cached data) that may result from calls to those stale objects.

The OSGi specification briefly describes this issue and refers to it as *Stale References*. Avoiding it is a matter of good programming practices since the environment cannot control or inspect it. Although there are mechanisms to minimize the occurrence of this problem, it is not possible to assure that every possibility of stale reference is being taken care of. This problem is difficult to detect in existing diagnostic applications (e.g., Eclipse TPTP, Netbeans profiler, Borland Optimizeit) because it is a consequence of particularities in the OSGi dynamic environment.

The market of OSGi based Commercial-Off-The-Shelf (COTS) components is rapidly growing [2]. Under the perspective of the OSGi dynamicity aspects that we have presented, existing tools or testing suites cannot guarantee or evaluate that OSGi based COTS components can be safely introduced in an OSGi platform without bringing any problems such as stale references upon OSGi life cycle events.

This paper proposes techniques that enable such type of validation for the OSGi environment. We go deeper in the stale references problem by classifying and detailing different patterns of stale references. We propose and validate diagnosis techniques that rely on Aspect Oriented Programming [7] to change OSGi framework implementations enabling them to provide information to detect those patterns during application runtime. We found that a static analysis approach may impose several constraints and it is not suitable to a dynamic environment such as OSGi. We also transparently introduce a fail-stop approach on calls to stale services to avoid the propagation of incorrect results.

Our detection techniques make possible to identify and visualize stale references, achieving an OSGi specific inspection feature that is not yet available in existing diagnostic tools. By identifying such problems it is possible to provide information that can help correcting bundle source code, allowing developers to guarantee the quality of their OSGi targeted applications and components.

All the techniques explained here were validated with the development of a diagnostic tool [8] that can be used to inspect OSGi targeted applications and components. The analysis of four open source OSGi based applications presented stale references after simulating life cycle (update, stop, uninstall) events.

The remaining sections of this paper are organized as follows: section 2 gives an overview of the dynamics in OSGi and its implications; section 3 details different patterns of stale references; section 4 explains the techniques for runtime detection of those patterns; section 5 presents the results of an experiment with 4 open source application as a part of the validation of our work; section 6 talks about related work; and, at last, section 7 presents the future work and conclusion.

2 OSGi Dynamics and Implications

The OSGi framework provides a straightforward service platform for the deployment of modules and services. The deployment unit in OSGi is called bundle, which is an ordinary compressed jar file with classes and resources. The jar file manifest contains OSGi specific attributes describing the bundle. A bundle can be dynamically loaded or unloaded on the OSGi framework and may optionally provide or consume services, which can be any Java object. Applications can take advantage of the dynamic loading feature to update software components without the need to stop the application. For example, a production system may have a bundle updated with a new version due to minor bugs fixed or other types of improvements.

Bundles can access the OSGi framework through a *BundleContext* object which becomes available in the bundle's activation process. Through that object they can register and retrieve services. In OSGi, services are ordinary Java objects that are registered into the framework service registry under a given interface name. The basic process to retrieve a service instance consists in two steps: it is necessary to ask the BundleContext for the desired interface, resulting in a *ServiceReference* object which holds metadata of a service. The next step is to use the BundleContext again to retrieve the service instance that corresponds to that ServiceReference object.

Upon service registration, modification or unregistration—either explicit or implicit when the defining bundle is stopped— the framework notifies the subscribers of the *ServiceListener* interface. Therefore, it is possible for service consumers to know when services become available (registered) or unavailable (unregistered).

Any OSGi targeted code should be written considering the arrival and departure of bundles and services. The code must release references appropriately upon such events. Service consumers must be aware that a service departure means that a service instance or its ServiceReference must not be used anymore. Any usage of the unregistered object may lead to inconsistency.

2.1 Bundles Isolation through Class Loaders

Whenever a bundle is loaded —either during startup or later during runtime— it is provided with its own class loader. Classes and resources from a bundle should be only loaded through its class loader. This individual class loader mechanism permits to unload from memory all classes provided by a given bundle when it is stopped.

The OSGi framework provides a basic level of isolation between bundles by means of that class loading mechanism. A bundle may choose which packages will be visible to other bundles by defining in its manifest an attribute with a list of exported packages. Only classes from exported packages (specified in the bundle manifest) may be instantiated by other bundles, which also need to explicitly specify in their manifest what packages they import. Whenever a bundle tries to reference a type, its class loader will enforce if the visibility rules are followed. Other mechanism that can be seen also as an isolation enforcement is the utilization of optional framework security permissions (AdminPermission, PackagePermission, and ServicePermission) which can provide a fine grained control to grant authority to other bundles perform certain actions, for example to retrieve a given service instance.

2.2 Isolation Limitations

Although there is some isolation level between bundles, this mechanism cannot ensure complete or safe removal of bundles from memory. During bundle active time objects can be exchanged freely between bundles. For instance, a service may receive a parameter object that comes from other bundle. If the bundle that provided the parameter object is stopped there is no guarantee that the service will stop referencing the object it received as parameter, even if the bundle of origin of that object uninstalled from the framework.

There is no security enforced communication channel (e.g., communication via proxy objects) that can be closed upon bundle departure, nor a protection domain (i.e, individual object spaces in memory) that enforces communication restrictions or other forms of application isolation.

The OSGi platform does not provide a true means of isolation between bundles. It mostly relies in a set of good programming practices to avoid the misreferencing of objects after bundles are stopped.

2.3 Stale References

The OSGi specification, release four, defines in the section 5.4 a stale reference as

> *"a reference to a Java object that belongs to the class loader of a bundle that is stopped or is associated with a service object that is unregistered"*

The utilization of such objects after the provider bundle being stopped leads to inconsistencies such as (1) incoherent operation results (e.g., stale services returning old data from stale caches) or erroneous behaviour due to the stale object's context (e.g.., network connections, binary streams) be released or de-initialized; (2) garbage collection obstruction of the retained object, its class loader, and the class loader's loaded types, leading to a memory leak.

Utilizing a ServiceTracker or an OSGi component model helps to minimize the occurrence of stale references. The ServiceTracker is a utility class in the OSGi framework for providing a transparent means for locating services but it is error prone since service consumers may not release the consumed service instances appropriately. OSGi Declarative Services (part of the OSGi R4 compendium specification), Service Binder [9], iPOJO [10] and Spring Dynamic Modules [11] are OSGi component models that provide the transparent handling of services arrival and departure. However, their usage would not avoid all possible types of stale references. Other patterns of stale references which are detailed in the next session may not be avoided by such mechanisms.

2.3.1 Propagation of Incorrect Results

The usage of an unregistered service may lead to inconsistent method calls. If a bundle unregisters a service, it is likely that the service needs to be disposed; therefore it may release internal resources (open file streams and database connections, etc) and calls on that object would produce erroneous behaviour. Exceptions may be raised

(e.g., access to a method that internally would try to use a closed connection) when methods of stale references are used. However if such method calls do not fail but produce incorrect results, there is a worst scenario where faulty components are introduced into the system with risks to propagate inconsistencies throughout the whole application. This can happen due to the stale object's internal state being invalid or stale (e.g., old cached data), which compromises the accuracy of operations involving that object. Such types of faults are harder to detect since the system would hide these issues and continue to work apparently without any problem.

A service failure mechanism, as presented in [12], currently is not enforced by the platform. A fail-stop strategy would be able to make the faults more explicit when using stale references. If any calls to stale references would result in a crash (an exception thrown) there would be no propagation of incorrect results, and bugs would be evident.

2.3.2 OSGi Specific Memory Leaks

While the previous problem may sometimes be identified due to exceptions thrown, memory retention is rather difficult to be seen. In addition, the retention of class loaders impedes OSGi to dynamically unload the classes from a stopped bundle.

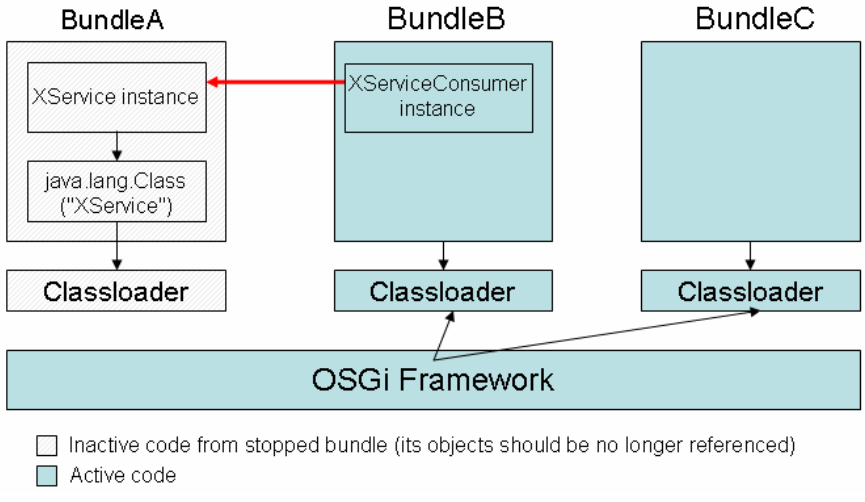

Fig. 1. The arrow from BundleB to BundleA illustrates a stale reference that prevents the appropriate unloading of BundleA from memory

According to the Java Language Specification [13], a class or interface reification (a java.lang.Class instance) may be unloaded if and only if its defining class loader may be reclaimed by the garbage collector. As long as an object from a stopped bundle is reachable (Figure 1) we will have a reference to that object's type as well, which references the bundle class loader which keeps all loaded types. Consequently, the classes can never be unloaded due to the presence of stale references.

3 Patterns of Stale References

As stated previously, the framework cannot guarantee that the objects provided by a bundle will no longer be referenced when the bundle stops. Neither the OSGi framework itself nor the mechanisms mentioned in section 2.3 can completely avoid stale references. In the current OSGi specification, the framework needs to share responsibilities with bundles. The bundle side is error prone as it depends on good programming practices to correctly handle the departure of services and bundles.

The correct handling mentioned previously will handle only a few patterns of stale references. We have classified three main patterns: (1) Stale services; (2) forwarded objects and (3) active threads from stopped bundles.

3.1 Stale Services

Stale services are a pattern of stale references that can be found when an unregistered service is still being referenced by active bundles. We considered that there are two levels of service referencing: reference to a service instance and reference to a ServiceReference instance. The former is the service object itself and the latter is a framework metadata object which is necessary to get a service instance. We kept references to ServiceReference instances as a simple case, but we classified a specialization of the reference to service instances as two possibilities: services from stopped bundles and services from active bundles. Therefore, we present the concept of stale services as three variations:

- Reference to an unregistered instance of a service whose bundle is still active (has not stopped);
- Reference to an unregistered instance of a service from a stopped bundled (update or uninstallation would lead to stopping the bundle as well);
- References to unregistered org.osgi.framework.ServiceReference objects.

The first case can happen during the active life-time of a bundle which may unregister a service due to an internal bundle change, for example. If after unregistration the service instance is retained by service consumers from other bundles we have a case of stale reference. In this case, the service can propagate incorrect results and it will also be prevented to be garbage collected.

The second pattern is rather similar to the first one, but now the propagation of errors is more likely because the bundle has been stopped and may have suffered some de-initialization code. In addition, the bundle class loader and classes would be prevented to be unloaded from memory.

The latter case of stale service (references to unregistered ServiceReference objects) does not prevent the unloading of bundle classes because there would be no reference to a bundle object, since the ServiceReference object is provided by the framework. Because of that, one may argue that this pattern does not fit the stale reference definition. However, this case has been added to our patterns because it may bring faults to the application and also characterizes the mishandling of service unregistration. When a ServiceReference is unregistered, subsequent calls to the framework using that ServiceReference object would return a null value, leading to a NullPointerException upon any method call attempt on the resulting value.

3.2 Forwarded Objects

Bundles may freely exchange messages between them by means of service method calls. Ordinary objects may be passed as method parameters across bundle boundaries without restriction. Also, there is no restriction for a service to retain an object received as a method parameter or to forward that object reference to objects from other bundles. If the bundle that provides that forwarded object is stopped, the same memory retention problem as the stale service pattern would happen. The same also applies when objects are registered in server object repositories (e.g., MBean server, RMI registry) and are not appropriately unregistered when bundles are stopped.

We have identified two variations of the forwarded objects pattern:

- Forwarding of ordinary (non-service) objects
- Forwarding of services as ordinary objects

Figure 2 shows an example of the forwarding of an ordinary object. Consider that the code on that example runs on an object from Bundle X, and foo.BarService is provided by an object from Bundle Y. Bundle X calls a method on a service from Bundle Y and sends a parameter, which is a local ordinary (non-service) object from Bundle X. That parameter will be retained as an attribute in the Bundle Y service. If Bundle X is stopped, uninstalled or updated, the object that was sent to Bundle Y's service will fit in the regular case of stale reference: impossibility to garbage collect the referenced object (localObj) and to unload the classes previously provided by Bundle X's class loader.

```
//Code on a BundleX retrieves a service from a BundleY
ServiceReference ref =
ctx.getServiceReference("foo.BarService");
BarService bar = (BarService)ctx.getService(ref);

//LocalObject is created in (and provided by) BundleX
LocalObject localObj = new LocalObject();

//service from BundleY will hold an object from BundleX
bar.setAttribute("anAttribute", localObj);
```

Fig. 2. Forwarding of an ordinary object

The second type of forwarded object pattern is detailed in Figure 3. It shows that the Bundle X uses a service instance from Bundle Z and forwards that instance to a service from a third bundle (Bundle Y). Bundle Y now references an object from Bundle Z without knowing that it is a service. Although at that time the foo.BarService service holds an instance of xyz.AService, most likely it would ignore the unregistration of xyz.AService, since the setAttribute method semantics does not expect a service. If Bundle Z (the provider of the xyz.AService "attribute service") is ever stopped, the foo.BarService will not release the reference to the xyz.AService

object. Bundle Y would point to a stale reference that prevents the unloading of classes from Bundle Z.

A significant difference between referencing an ordinary (non-service) object from a stopped bundle and referencing a service instance from a stopped bundle is the absence of framework events to notify the departure of ordinary objects. But if a forwarded service is treated as an ordinary object, notifications of service unregistration are ignored and do not help.

```
//Code on a BundleX retrieves a service from a BundleY
ServiceReference ref =
ctx.getServiceReference("foo.BarService");
BarService bar = (BarService)ctx.getService(ref);

//Code on a BundleX retrieves a service from BundleZ
ServiceReference anotherRef =
ctx.getServiceReference("xyz.AService");
AService servObj = (AService)ctx.getService(anotherRef);

//service from bundleY holds a service as an attribute
bar.setAttribute("anAttribute", servObj);
```

Fig. 3. Forwarding of a service instance as an ordinary object

3.3 Active Threads from Stopped Bundles

According to the OSGi specification, when a bundle is stopped it has to immediately stop all of its executing threads. Since there is no isolated bundle space in memory, the framework cannot cancel a bundle's set of executing threads. So, it must rely on good OSGi programming practices leaving that responsibility to the bundle developer.

Table 1. Summary of stale references

Referred object	**Memory Retention (bundle objects and class loader)**	**Incorrect Results**
Unregistered Service instance (Stopped bundle)	Yes	Yes
Unregistered Service instance (Active bundle)	Yes (but no class loader retention)	Yes
Unregistered `ServiceReference` instance	No	Yes (NullPointerException)
Active Thread (stopped bundle)	Yes	Yes
Forwarded object (stopped bundle)	Yes	Yes

If the thread is not stopped in such cases, the same stale reference issue is found: an object (the Runnable object) from a stopped bundle is still reachable in memory, preventing garbage collection of its class loader (the bundle class loader) and the loaded types of that bundle.

4 Detection Techniques

Information to track object references and diagnose stale references is not present in implementations OSGi of the framework. Several reasons have led us to think that changing the source code of OSGi implementation to add that information would not be adequate. It would be needed to inspect the registration and retrieval of services, class loader creation, etc. The custom code to track such objects would be scattered all over the OSGi framework implementation code. It is clear that a solution which customizes a given OSGi implementation would compromise the portability to other OSGi implementations. In addition, other problems such as tracking the creation of threads would concern bundles but not the framework. This would imply in changing bundle code as well, which we most likely don't have access in all applications.

The whole situation led us to choose the application of Aspect Oriented Programming (AOP) [7] techniques. Instead of adding a cross-cutting concern to the code of OSGi implementations, we left the tracking code as separate aspects. AOP would enable to weave those aspects into different OSGi implementations. The process would be the same for all of them: each implementation would have its bytecode changed resulting in a composed implementation capable of providing information to identify stale references.

The reference tracking techniques presented here rely on a special type of reference provided by the Java programming language, called *weak reference.* Weak references are different than ordinary (strong) references. They do not prevent a referred object to be reclaimed from memory and are able to tell if an object has been garbage collected.

4.1 Point Cut Definitions

AOP introduces the concept of *joint points*, which are well defined points in the program flow (e.g., method call, constructor call). *Point cuts* are elements that pick one or more specific join points in the program flow. We have defined two different sets of point cuts. One was responsible for aspects that would be applied to the framework, for example tracking service registration and retrieval, bundle start up, class loader creation, etc. The other set of join points was responsible for the aspects on bundles, which so far were limited to the creation and start up of threads.

The code that is injected into point cuts during the weaving process is called *advice* in AOP terminology. The portions of code defined in the advices are executed during method interception. In the techniques that we have developed and tested, the advices contained the calls to the code that enabled the tracking of objects.

4.2 Detection of Stale Services

With AOP, service registration can be intercepted and each ServiceReference object tracked with weak references. Our technique consists also in track the garbage collection of each instance provided by a ServiceReference. Multiple service instances can be served by the same ServiceReference when the service provider is a *ServiceFactory*, which can provide one service instance per bundle.

In order to verify the existence of stale services, it is necessary to analyze tracking information relative to unregistered ServiceReference objects. There are two straightforward manners to know the existence of stale services. One is checking if the unregistered ServiceReference object has not been garbage collected, and the other is to verify if all service instances of each unregistered ServiceReference have been garbage collected. The former would characterize the pattern of a reference to an unregistered ServiceReference. The latter identifies the pattern of a reference to an unregistered service instance.

4.3 Detection of Active Threads from Stopped Bundles

The detection of thread creation and its start up in bundle code is necessary in order to have more information about them. Instead of weaving the framework, this approach implies in weaving the bundles. Two options are possible: static weaving or dynamic (runtime) weaving. The same aspects are reusable in both approaches.

The static weaving is easier to perform but adds the step of externally weaving the bundles before loading them into the platform. The dynamic approach is more flexible but adds the overhead of weaving while loading the bundles in runtime. It is also necessary to add code in the framework, by AOP as well, to intercept the loading of bundles and dynamically weave them.

The information on thread point cuts allows establishing a bundle-thread relation that can be stored for later inspection. Running threads that are in the bundle-thread map can have their metadata inspected (e.g. the class loader of the bundle that started the thread) and compare it with logged information of the bundle that started the thread. It is possible to identify if the bundle that started the thread has been update, stopped or uninstalled.

4.4 Identifying Forwarded Objects

Identification of forwarded objects was found to be more difficult and depends on the inspection of dumps of memory, as the one provided by tools such as jmap which comes with the Java 6 SDK. It is necessary to inspect a memory dump and verify if there are reachable objects whose class loader belongs to a stopped bundle. Jhat is a tool also available in the Java 6 SDK which allows performing queries o memory dumps. Its API can be integrated into applications that can programmatically perform queries on memory.

Establishing a relation between runtime information and memory dump information is difficult. An object's id in the heap is a sort of JVM private information that is not available to the runtime objects via a Java API. User intervention constructing ad-hoc queries has proven to be more precise some times. This happened due to the fact that automated inspection extracted runtime information of private attributes by

means of reflection and compared it with results from queries on memory dumps. The results most of the times would return a list of suspects that would need to go through a manual inspection by the user.

5 Validations and Experiments

The techniques to detect the patterns of stale references presented here were developed, tested and validated. We have developed a diagnostic tool called Service Coroner [8] which examines the "dead" objects from stopped bundles. Our work comprises the implementation of the aspects to track the code, the classes to perform the queries, the tool that visualizes the problems and a fail-stop mechanism to avoid calls on stale services. The latter was developed as a side experiment that we detail in the end of this session.

Aspects were developed and weaved with AspectJ [14] and each technique was initially validated by bundles that were intentionally developed with errors that would present stale reference problems. A series of life cycle events (stop, update or uninstall) would lead to stale references that were diagnosed by the tool.

The diagnostic tool and the results of an initial experiment are presented in [8]. We have extended that experiment by adding two other open source applications and also analyzing stale threads. The tool is able to inspect OSGi applications and diagnose the patterns presented in this paper.

5.1 Portability Across OSGi Implementations

Althought the process of weaving an OSGi implementation may be seen as intrusive due to the changes it performs in the bytecode, the techniques that we have developed as separate aspects where easy to be applied to different OSGi implementations. As part of the validation, we have achieved to weave the diagnostics aspects into the three main implementations of the OSGi specification, Release 4: Equinox [15], Felix [16] and Knopflerfish [17]. All of the weaved platforms were successfully tested with our bundles that present the stale references patterns.

From a source code point of view there was no need to change any of the implementations. The process of aspect weaving was the same on all of the three platforms, and consisted on a simple build process that basically compiles the Service Coroner tool, the aspects and then weaves the aspects into the OSGi implementation.

5.2 Experiment on Open Source OSGi Applications

We have validated the diagnostic tool in an application scenario where errors would not be intentional like in our test bundles. Four open source applications constructed on top of OSGi were inspected with the Service Coroner tool: JOnAS[1] 5.0.1 [18], SIP Communicator Alpha 3 [19], Newton 1.2.3 [20] and Apache Sling [21]. JOnAS is a JEE application server; SIP Communicator is a multi-protocol instant messenger application; Newton is a distributed component framework that provides an

[1] We have also inspected Apache Geronimo and Glassfish V3 JEE servers, however analyzing them would not bring significant results since they do not use the OSGi service layer.

implementation of the Service Component Architecture (SCA) standard [22]; and Sling is a web framework that uses a Java Content Repository. All applications are of significant size, especially JOnAS, whose core is about 400 000 lines of code but comes to over 1 500 000 when the other components are taken into account.

Table 2 presents an overview of the experiment that was run on a Sun HotSpot JVM 1.6.0u4. All OSGi implementations utilized have been previously weaved with the aspects that we have developed. The line IV of table 2 shows that JOnAS, SIP Communicator and Sling are partially developed with component models for the OSGi Platform: iPOJO, Service Binder and Declarative Services, respectively. Nevertheless, Newton which provides an implementation of SCA has not been developed with a component model.

Table 2. Overview of the experiment. Lines VIII to XI present the results.

I	Application	**JOnAS**	**SIP Comm.**	**Newton**	**Sling**
II	Version	5.0.1	Alpha 3	1.2.3	2.0 incubator snapshot
III	OSGi Impl.	Felix 1.0	Felix 1.0	Equinox 3.3.0	Felix 1.0
IV	Bundles using Component Models	20 iPOJO [10]	6 Service Binder [9]	0[2]	18 Declarative Services [1]
V	Lines of Code	Over 1 500 000	Aprox. 120 000	Aprox. 85 000	Over 125 000
VI	Total Bundles	86	53	90	41
VII	Initial No. of Service Refs.	82	30	142	105
VIII	No. of Bundles w/ Stale Svcs.	4	17	25	2
IX	No. of Stale Services Found	7	19	58	3
X	No. of Stale Threads	2	4	0	0
XI	Stale Services Ratio (IX/VII)	8.5 %	63 %	40.8%	2.8%

The tool was capable of executing scripts that could simulate life cycle events (update, start, stop, uninstall). A script executed by the tool simulated the update of components during runtime by performing calls on the update method of bundles that provide services (except for bundles related to the OSGi framework or component models). We used a standard 10 seconds interval between each bundle life cycle method call. With Newton and Sling we had to adapt the script because of exceptions being raised during bundle update. Instead of the update method, we performed a call to the stop and start methods with the standard interval between each call.

[2] Actually the whole Newton implementation is an SCA constructed on top of OSGi, but its bundles did not use an OSGi component model like the other analyzed applications did.

5.3 Fail-Stop Calls on Stale Services

A crash-only principle, as provided in [12], could be adapted to services in the OSGi environment. We have implemented this fail-stop approach to avoid the propagation of incorrect results when calling methods on stale services. Any method call on stale services would throw an exception. Actually such calls were being done through a proxy object dynamically generated.

We have added another point cut to intercept the calls of the getService method in the BundleContext. Whenever a service instance was requested, the result would be a proxy object that wrapped the service instance. The proxy would receive the calls and delegate them to the actual service. Upon service unregistration, the proxy object had its state invalidated. Subsequent calls to the invalidated proxy would throw a runtime exception. Proxies were cached to avoid creating multiple proxies for the same service instance if it was requested multiple times.

The experiment presented previously did not utilize the fail-stop services. We have successfully tested it in a controlled environment where we developed all bundles deployed in the framework. Other adjustments would be necessary to make our implementation more robust and usable in other scenarios. This strategy could be taken further to minimize the impact of stale services, the strategies to handle such exceptions would allow the auto correction of applications that upon such crashes could react trying to retrieve a valid service or aborting the operation if no valid instance of the service is found.

5.4 Limitations and Drawbacks

Some drawbacks have been found regarding the implementation of the techniques presented here. The first one is regarding the OSGi optional security layer when using digitally signed jars files. Since we have utilized bytecode weaving, the resulted jar file will be different from the original one. Thus, the loading of the changed framework jar file will imply in security errors that will impede the start up of the OSGi platform. This could be found with Equinox [14] version 3.3.2 which provides the digitally signed jars feature, a security feature whose objective is to ensure that jars contents are not modified. In order to utilize our tool, such security option would have to be disabled. We have achieved to turn that off on Equinox by removing all information about security on the manifest and the jar file.

The second drawback was found when doing inspections of memory dumps using the jhat API integrated to our tool. The process of reading memory dumps consumes a large amount of memory and occasionally would lead to out-of -memory errors. An alternative would be using such tools as a parallel auxiliary tool instead of trying to integrate it with the running application.

Although we have removed the propagation of incorrect results produced by stale services and made their utilization explicit by throwing exceptions, generally the proxy solution of our fail-stop approach has two limitations. It does not completely solve the memory retention problem. Upon service unregistration the proxy can free the reference to the actual service, but the service class loader (and all java.lang.Class objects it has loaded) would still hang in memory.

6 Related Work

Our work addresses a problem which is a consequence of code isolation limitations in a specific Java-based middleware for services and components. The same issues apply in environments with similar modularity approaches based on the concepts of OSGi, such as the upcoming Java Module System [23]. Thus theses techniques could be adapted to detect the same problem when that system becomes available.

We focus on the dynamic diagnosis of OSGi applications, evaluating OSGi specific problems during runtime. There are other mechanisms partially addressing this problem in OSGi and in other platforms as well. OSGi component models [9], [10] and [11] provide mechanisms that automate service location and handle service departure but do not avoid all patterns of stale references, as previously mentioned.

A formal model was built on [24] for OSGi verification. By doing that formal analysis they were able to check and identify stale references problems. However their solution was coupled to a specific OSGi implementation (Knopflerfish) and constrained by the limitation of the environment that was used for formal verification. Only applications with a maximum of 10 000 lines of code could be analyzed. They proposed three different solutions to avoid stale references. On each solution the services would have to extend from a default service superclass that provides a lock object. All solutions would depend on synchronization on that object in order to acquire a lock to access the service.

A service failure approach [12] presents a fail-stop solution to handle faults in the composition of services in SOA environments where consumers of a service must anticipate that any service provider will crash from time to time. Another work [25] presents, like ours, a proxy-based service solution to deal with fault tolerance. However, their approach to is different and does not prevent the stale service from being called. Their proxy implementation is responsible for dynamically locating the best service implementation, and in case of faults it tries to locate another service.

Concerning isolation mechanisms, other environments such as .NET [26] have concepts like application domains which resemble lightweight processes isolated from one another and can even be terminated without interfering in the other domains execution. Communication across application domains is done in an RPC fashion and objects are sent via marshalling. Application domains can be dynamically loaded but have limitations in being unloading.

In Java, an effort on JSR 121 [27] provides an environment where applications can be isolated from each other by means of *Isolates*, which are application units which resemble lightweight processes. Applications are isolated in different object spaces but they can share some resources like runtime libraries. Communication between isolates can be done through Java RMI (remote method invocation) based mechanisms which imply in marshalling objects across contexts.

7 Conclusions and Future Work

The OSGi service platform is a dynamic environment for modules (bundles) and services, but it still does not provide a completely isolated environment where services and bundles may be transparently removed during runtime without the risk of

having their objects still being referenced by active code. Memory instrumentations tools currently available (e.g., Eclipse TPTP, Netbeans profiler, Borland Optimizeit) do not consider such particularities of the OSGi framework such as bundle life cycle. The problem of *stale references* described in the OSGi specification may happen if misprogrammed bundles do not handle correctly services unregistration and bundles unavailability. The utilization of stale references introduces memory leaks and faulty components into the system due to the propagation of incorrect results (e.g., a stale service that provides invalid cached data).

This paper presents different patterns of stale references, techniques to diagnose them and a fail-stop mechanism to minimize inconsistent results due to the utilization of stale services. The runtime diagnosis techniques presented here were implemented and validated in a tool called Service Coroner, and were effectively tested against four open source applications. Our detection techniques provide a solution that is portable across different OSGi implementations, without needing to change their corresponding source codes. We rely on AOP to keep the tracking code as separate aspects that can be weaved into different OSGi implementations. Weak references were used to identify which tracked objects have been garbage collected or not.

In a COTS market that targets OSGi application it would be necessary to somehow measure the quality of the components. For example, if they are able to be updated in the system without leaving any weak references or if they would not provoke such problems in the system.

The diagnostics tool that is part of our work addresses OSGi specific issues not covered by currently available tools. Our techniques have proven that it is completely feasible to analyze large OSGi applications and components during runtime, allowing to detect the presence of implementation flaws that lead to stale references. We were able to evaluate if the applications' components are ready to handle some dynamic characteristics of the OSGi platform like being able to cope with module updates.

The initial fail-stop mechanism that we provided invalidates any method call on stale services, avoiding the propagation of incorrect results and facilitating to know where stale services are being used in the application. Some improvements need to be done in that mechanism in order to run it in any type of OSGi application.

In our future work, we also plan to provide a more automated test approach by wrapping the script execution on unit tests. A wider range of OSGi based applications should be tested. It would also be important to adapt the presented techniques for providing the runtime inspection of the Eclipse platform's extension points (although constructed on top of OSGi, Eclipse has its own dynamic plugin mechanism).

References

[1] OSGi Alliance, `http://www.osgi.org`

[2] OSGi Alliance. About the OSGi Service Platform, Technical Whitepaper Revision 4.1, `http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf`

[3] Delapp, S.: Industry Use of OSGi Continues to Increase (retrieved April 9, 2008), `http://www.infoq.com/news/OSGi-Use-Increases`

[4] Chappel, D.: Universal Middleware: What's Happening With OSGi and Why You Should Care (retrieved April 9, 2008), `http://soa.sys-con.com/read/492519_3.htm`

[5] Desertot, M., Donsez, D., Lalanda, P.: A Dynamic Service-Oriented Implementation for Java EE Servers. In: 3th IEEE International Conference on Service Computing, Chicago, USA, pp. 159–166 (2006)
[6] Gruber, O., Hargrave, B.J., McAffer, J., Rapicault, P., Watson, T.: The Eclipse 3.0 platform: Adopting OSGi technology. IBM Systems Journal 44(2), 289–300 (2005)
[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
[8] Gama, K., Donsez, D.: Service Coroner: A Diagnostic Tool for Locating OSGi Stale References. In: Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications, Parma, Italy (2008)
[9] Cervantes, H., Hall, R.S.: Automating Service Dependency Management in a Service-Oriented Component Model. In: Proceedings of the 6th International Workshop on Component-Based Software Engineering, Portland, USA (2003)
[10] Escoffier, C., Hall, R.S., Lalanda, P.: iPOJO: An extensible service-oriented component framework. In: IEEE International Conference on Service Computing, Salt Lake City, USA, pp. 474–481 (2007)
[11] Spring Dynamic Modules for OSGi™ Service Platforms, http://www.springframework.org/osgi
[12] Hobbs, C., Becha, H., Amyot, D.: Failure Semantics in a SOA Environment. In: 3rd Int. MCeTech Conference on eTechnologies, pp. 116–121. IEEE Computer Society, Montréal (2008)
[13] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn., pp. 330–331. Addison-Wesley, Reading (2005)
[14] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)
[15] Equinox Framework, http://www.eclipse.org/equinox/framework
[16] Apache Felix, http://felix.apache.org
[17] Knopflerfish OSGi, http://www.knopflerfish.org
[18] JOnAS Open Source Java EE Application Server, http://jonas.objectweb.org
[19] SIP Communicator, http://www.sip-communicator.org
[20] Newton Framework, http://newton.codecauldron.org/
[21] Apache Sling, http://incubator.apache.org/sling/
[22] Service Component Architecture Specifications – Open SOA Collaboration, http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
[23] JSR 277: Java Module System, http://jcp.org/en/jsr/detail?id=277
[24] Chen, Z., Fickas, S.: Do No Harm: Model Checking eHome Applications. In: Proceedings of the 29th Intl. Conference on Software Engineering Workshops, Minneapolis, USA (2007)
[25] Ahn, H., Oh, H., Sung, C.O.: Towards Reliable OSGi Framework and Applications. In: Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France, pp. 1456–1461 (2006)
[26] Escoffier, C., Donsez, D., Hall, R.S.: Developing an OSGi-like service platform for. NET. In: Consumer Communications and Networking Conference, Las Vegas, USA, pp. 213–217 (2006)
[27] JSR 121: Application Isolation API Specification, http://jcp.org/en/jsr/detail?id=121