Towards Dynamic Component Isolation in a Service Oriented Platform

Kiev Gama and Didier Donsez

University of Grenoble, LIG, ADELE team
kiev.gama@imag.fr, didier.donsez@imag.fr

Abstract. When dealing with dynamic component environments such as the OSGi Service Platform, where components can come from different sources and may be known only during runtime, evaluating third party components trustworthiness at runtime is difficult. The traditional namespace based isolation and the security mechanisms provided in the Java platform (the base platform for OSGi) can restrict the access of such components but can not provide fault isolation. In this paper we present a dynamic component isolation approach for the OSGi platform, based on a recently standardized Java mechanism. When an untrusted component is activated during runtime, it is isolated in a fault contained environment but it can still collaborate with the application. If it is observed that the untrusted code does not bring any threat to the application, at runtime it can be dynamically promoted to the safe environment. Tests have been performed in a controlled environment where misbehaving components hosted in the sandbox were not able to disturb the main application.

1 Introduction

In Component Based Software Development (CBSD) one may not know in advance the impacts (e.g. runtime incompatibilities, errors leading to application crashes) of integrating third party components into an application. During development components can be tested (e.g. unit testing) as individual blackbox entities but component vendors may face combinatorial explosions when trying to validate their products against possible system configurations, and these combinations still grow if components can still be integrated after deployment of the system [39]. This is exactly the case of dynamic platforms where one may not predict which components are going to be deployed during application execution.

The OSGi Service Platform [29] is a component framework for the Java Platform, and is an example of such type of dynamic platform where components can be deployed, started, stopped or updated during runtime without stopping application execution. In dynamic environments as OSGi it is a frequent scenario having dependencies to service *interfaces* known at compile time but during runtime having the corresponding service *implementations* provided by possibly untrusted components dynamically deployed. The usage of OSGi in software industry has gained a strong momentum after the Eclipse Platform became one of its main adopters [10]. A large COTS market around OSGi is emerging [30] where third party components are becoming available increasingly, but defining their quality and trustworthiness is not a precise task. COTS quality models do exist, but they are difficult to be used due to the large quantity of attributes to be measured and the lack of information provided by component vendors [20]. The *reliability* characteristic (maturity, recoverability and fault tolerance as sub-characteristics) of those models is indirectly one of the attributes of component *trustworthiness*, which can be defined as measured and perceived dependability (a combination of reliability, safety, robustness, availability and security) [34].

As previously mentioned, fault isolation is an issue closely related to the reliability characteristic and can make composites stronger. It is indeed an essential theme in CBSD, since the strength of a composition is defined by its weakest component [39]. Since fault is a concept that has a very wide scope, we consider the concept from [23], which says that faults are the cause of errors (deviations from correct state) which may lead to system failures, thus being a threat to dependability. A detailed analysis [31] on component vulnerabilities in Java Service Oriented Platforms shows that some of them are caused by the lack of CPU and memory isolation between components, which is fundamental for fault isolation. OSGi uses class loader based namespace isolation, giving a sort of pseudo-isolation between components. However, namespace isolation is not robust enough for a multiple component vendor scenario where one cannot assure that such third party code behaves correctly. As all components and objects coexist in the same memory space without any mechanism that ensures object domains or other elaborate ways of isolation, components may introduce faults in applications:

- Inconsistencies and silent errors in the system when dynamicity is mishandled by components [7], caused by different factors such as incorrectly refactored applications and components.
- A component crash (e.g. stack overflow, out of memory exception) may bring the whole application down.

The objectives of this paper are: to provide a review on standard isolation mechanisms in the Java platform; and to present a component isolation approach for OSGi, based in one of the analyzed Java isolation mechanisms. The proposed mechanism allows untrusted components to execute in a fault contained sandbox that allows clean termination of components preventing any harm to the core (trusted) components environment enhancing applications' robustness and reliability. Our solution is based on a standardized Java Specification Request (JSR) that addresses isolation. However, by adding such isolation barriers, we are aware that a component communication overhead will be introduced. The intention is not to isolate each and every component since it would annihilate one of OSGi's main advantages, which is the fast communication between services and components.

The rest of this paper is organized as follows: section 2 details the standard isolation approaches in the Java Platform; section 3 analyzes the usage of those mechanisms for component isolation in different Java editions; section 4 highlights OSGi's isolation limitations; section 5 presents our proposed model and its implementation; section 6 describes related work; and finally, section 7 concludes and presents our perspectives.

2 Standard Isolation Mechanisms in Java

In this section we explore the standard isolation mechanisms that we have identified in the Java platform: namespace-based isolation, OS-based isolation and a relatively recent approach based on a sort of domain isolation.

2.1 Namespace-based Isolation

As explained in [24], the class loader mechanism in Java provides the ability to dynamically load classes during application execution enabling features such as lazy loading; unloading of classes; multiple namespaces; and user extensibility through user defined class loading policies. These multiple namespaces are the standard form for achieving isolation in Java, where a class type is uniquely determined by the combination of class name and class loader. To better illustrate namespaces with class loaders, consider that two class loaders A and B co-existing in the same running application can load different versions of a foo.Bar class. Each class loader can apparently provide instances of the same class but in fact the provided foo.Bar objects are of different classes. By considering a fully qualified name notation to differentiate each class, as the one used in [24], we have something like <foo.Bar, A> and <foo.Bar, B> which visibly do not correspond to the same class.

The basic loading mechanism is based on a delegation principle inside a hierarchy. Before loading a given class, a child class loader asks its parent for that class. If the immediate parent can not find the class, this delegation continues until the top of the hierarchy. The hierarchy of class loaders defines that children can "see" the classes loaded by their parent, but not the contrary. Following that principle, sibling class loaders also can not share class definitions. However, this mechanism isolates code in different namespaces but does not ensure object instances living in isolated address spaces. Thus, namespaces do not bring the necessary robustness because faults in code residing in a class loader can affect other parts of the application.

2.2 OS-based Isolation

This type of isolation is enforced by Operating System protection boundaries (e.g. processes in separated memory spaces). In Java this can be done with a combination of techniques by breaking a single application into multiple pieces running on different VMs (i.e. different processes) allowing application to be located in separate address spaces managed by the OS. Such type of isolation enables fault containment, thus a crash in a component would not bring the whole system down. However, using separate address spaces requires using relatively expensive inter-process communication in order to allow collaboration between the isolated components. In the case of Java it can be achieved either through sockets or higher level protocols such as RMI-IIOP. A significant disadvantage of such approach is exactly such type of cross-boundary communication overhead, as well as the memory footprint for each VM instance. Also, in the case of a component bringing a part of the application down, the restart of the crashed part would need to wait for the whole bootstrap of the

VM and the component container/runtime. This solution may be resource consuming, especially in small devices, but in server application cases such as [22] the decision of isolating several web applications in different VMs had an acceptable performance overhead that was taken into account in their analysis.

2.3 Domain Isolation

The JSR 121 [15] is a relatively recent standardization effort for application isolation in Java. It defines the notion of Isolate, a first class representation of a strong isolation container with an API to control their lifecycle. The model proposed by the Isolate API does not specify how Isolates should be implemented. The strategy is implementation specific and could range, for example, from a per-isolate operating system process (e.g. using a standalone JVM) approach, to all-isolates in one process (i.e. same JVM) approach. The latter is used in the reference implementation provided by Sunlabs in the Multitasking Virtual Machine (MVM) [4], which realizes Isolates using a multitasking approach. The MVM allows several Java applications to run in the same OS process, where each isolate is a logical instance of the JVM, with logically separated heaps, and no objects that can be directly shared. A basic set of resources, like runtime classes and shared libraries, is shared by all isolates but applications run in complete isolation. In case of an application failure, only that application is impacted, not the JVM. Other applications are completely shielded from that application failure. Besides isolation, other advantages are the low memory footprint for multiple applications in the same VM and quick application startup.

The isolation achieved with Isolates is completely transparent. Legacy Java applications can be executed in Isolates without needing any additional changes. However, applications can be aware of the existence of Isolates and explicitly use the API. Although isolated, Java applications can achieve collaboration through previously existing mechanisms such as sockets and Remote Method Invocation (RMI), or through *Links*, which are part of the Isolate API. They provide a low-level layer for communication through basic data types such as byte arrays, buffers, serialized objects and sockets. The usage of isolates can make applications more robust by adding fault containment and clean application termination, serving also as a basis for enabling other features such as the Resource Consumption Management API [19].

3 Component Isolation in the Java Platform

Given the standardized mechanisms, we provide in this section a brief analysis of component isolation using such mechanisms in the Java Platform and the respective approaches for component collaboration across isolation boundaries. From the Java standard and enterprise editions we describe the isolation in Applets and Enterprise Java Beans (EJB), respectively; in the Java micro edition (ME) we show isolation in two application models: Midlets and the Xlets, from the CLDC[1] and the CDC[2], respectively; and finally we see the isolation approaches in Java Card Applets.

3.1 Applets

The isolation achieved with class loaders combined with security policies is fundamental for guaranteeing a sandbox where applets have restricted visibility of other applications and controlled access to system resources, enforced by security verifications. This ensures that untrusted code (the applets) does not cause harm (e.g. accessing and damaging the file system) to the underlying system. The namespace based isolation through different class loaders guarantees that if a web page loads applets from different locations they do not have access to each other.

Applets are present in Java since the initial versions, when composition models were rudimentary and in the case of applets it could be done by placing the applets in the same web page [39] and letting them communicate via the AppletContext object. This can be possible only in the case of applets from the same code base, that is, the same directory on the server. Such rudimentary composition can not be possible when applets come from different locations.

3.2 Enterprise Java Beans

Isolation of EJBs is usually done in two flavors: either through class loaders namespaces or by isolating components in different JVMs. In the former case, isolation fits in the class loading delegation principle previously described. Although there is no fixed structure for class loaders in Java EE, each vendor has its own implementation that follows the same principles. Fig. 1, based on an illustration from [1], illustrates a class loader hierarchy in Java EE.

Class loaders in grey, on the top of the hierarchy on Fig. 1, are the standard Java class loaders provided by the platform. The other class loaders represent a general Java EE class loading scheme. Every Enterprise Application Archive (EAR) will have its own class loader that will provide each application with its own namespace [1]. All EJBs of the EAR will be loaded by the same class loader, thus sharing the same namespace. Each Web Application Archive (WAR) is deployed with its own class loader and will not have class visibility to other sibling application.

The whole EJB model was conceived with distribution in mind, thus remote communication is supported by the component container. The infrastructure for EJB communication is based in a message based IPC approach that uses the RMI-IIOP protocol. Thus EJBs can also be isolated by separating them in different VMs. A crash in one component would not directly affect components hosted in other VMs. However, this choice leads to problems such as scalability and memory footprint. The cost of isolating components in separate VMs hosting heavyweight runtimes such as EJB containers would be expensive in terms of resources; communication overhead and coordination. An experimental isolation approach [14], which is detailed on the

[1] Connected Limited Device Configuration. http://java.sun.com/products/cldc/

[2] Connected Device Configuration. http://java.sun.com/javame/technology/cdc/index.jsp

related work section, takes advantage of isolates for providing different levels of isolation in J2EE servers.

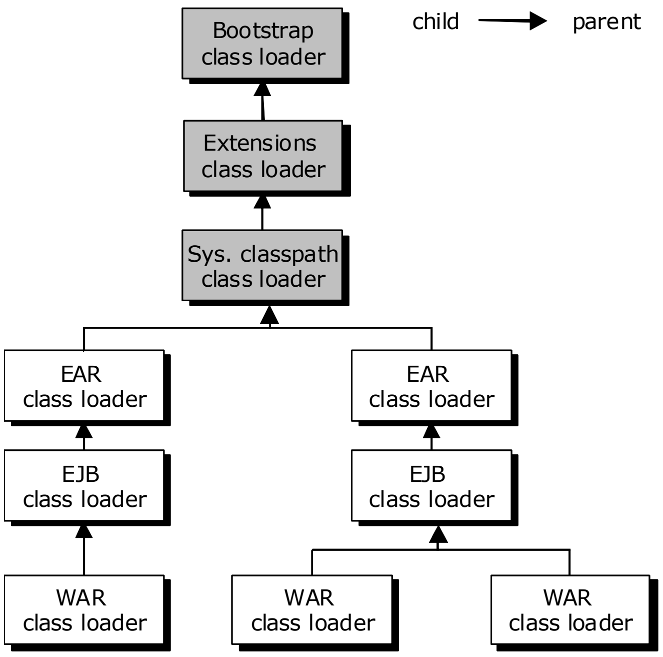

Fig. 1. Classloader delegation hierarchy in a Java EE Server.

3.3 Midlets

The Mobile Information Device Profile (MIDP) for CLDC introduces the concept of MIDlets, which are managed applications with a life cycle similar to Java applets. MIDP has been conceived to execute in constrained devices where Java MIDlets would usually execute one at a time with security constraints concerning aspects such as visibility of types restricted to the same MIDlet suite. The MIDlet security model enforces that each MIDlet suite must run in isolation, but concerns were mostly related to type visibility since initial versions of MIDP executed one MIDlet at a time. Starting from MIDP 3.0 [18], parallel execution of MIDlets is specified. The Inter-MIDlet Communication (IMC) protocol, similar to sockets, would allow two MIDlets to establish communication by means of a channel, enabling communication and possibly some rudimentary composition such as in the case of applets. Although communication is possible using the IMC protocol, runtime isolation is enforced: MIDlets must not be able to have access to the variables or memory from each other, having their own executing spaces. They will always run in different contexts when executed. The isolation concept also applies to the usage of shared libraries (LIBlets), with the set of classes and resources of a LIBlet behaving (e.g. per-context static variable value) as if they were packaged with the MIDlet.

Multitasking is already being used as a way for isolating applications with less memory footprint [36] in CLDC devices. The Sqwak Virtual Machine [38] is another multitasking VM targeting the CLDC but for the Information Module Profile [16], which consists on a profile for devices without graphical display capabilities. Both of those approaches are based on Isolates, which appears to be the next generation standard for multitasking in upcoming JVMs.

3.4 Xlets

The Personal Basis Profile (PBP) [17] for CDC provides an application model based on Xlets which are managed applications originally defined for the Java TV API. The Xlet application model resembles Java applets and MIDlets, providing also small applications with life cycle (init, start, pause, destroy). PBP provides a means of communication between Xlets with the Inter-Xlet Communication (IXC) mechanism, which uses a subset of Java RMI. Xlets executing in the same virtual machine are able to exchange objects across class loader boundaries. Although such communication takes place in the same VM, it relies on RMI proxies. An Xlet can register an object in the IXC registry. Other Xlets running on the same VM can perform a lookup in the registry to retrieve the object that is bound to the queried name. The result is a dynamically generated stub that implements the same remote interface of that object. Since code is running in different class-loaders, the class definitions are not shared. The client Xlet must have the same interface type of the requested object packaged with its application so it can correctly retrieve the corresponding stub instances.

The approach above described fits in the initial CDC VM monolithic versions that provide class loader based namespace isolation. The CDC Application Management System (AMS) [37] introduces process-based application management, where all Java applications run in native processes coordinated by the AMS. The IXC would continue working as the communication mechanism between Xlets, but in a more robust environment with fault isolation and clean application removal.

3.5 Java Card Applets

In the Java Card platform, applications are called applets (*card applets* for disambiguation). A firewall mechanism isolates card applets from each other by mean of contexts, which are separate protected object spaces. It enforces security constraints and provides a secure environment where card applets may not access each other's functionality unless explicitly specified through shareable interfaces (SI). These contexts provide a sort of object domain in terms of data visibility, but do not provide fault isolation. The separate spaces and security mechanisms do not prevent an unhandled fault from halting the VM, as described in its specification: “As the Java Card virtual machine is single-threaded, uncaught exceptions or errors will cause the virtual machine to halt”. Thus, a misprogrammed card applet that provokes a `StackOverflowError`, for example, affects all loaded applets. This applies also to the most recent Java Card specification (v. 3.0), which in addition to context isolation also provides *code isolation* in the connected edition. This type of isolation is implemented using the traditional class loader delegation hierarchy that provides separate class namespaces. Communication between card applets is still through SIs, but with the class loader hierarchy principle implies that the SI implementations be loaded by a higher level class loader so they can be visible to all card applets, which are all potential invokers of the shared object.

3.6 Summary

The predominant way for component isolation in Java is by means of class loaders, which allow separate namespaces that give less robust isolation. However a trend towards multitasking in the embedded market is observed as a means to enhance isolation. The utilization of Isolates allow programmatic access to an API for starting and controlling the execution of an application container that transparently provides strong isolation, enabling fault containment and a much more robust isolation mechanism than the one provided by class loaders. EJBs components can take advantage of isolation either with namespaces or in separate VMs, since these components where conceived for a distributed model where inter-VM communication is natural, but choosing to host individual EJBs in separate VMs leads to a rather complicated problem that would compromise scalability. Table 1 presents a summary of some isolation characteristics for each analyzed type of component approach.

Table 1. Isolation overview on the Java Platform

Component/Application Model	Isolation mechanism	Fault Containment	Collaboration
Java Applets	Namespace	No	Direct access
Local EJB	Namespace	No	Direct access
Remote EJB	OS based	Yes	RMI-IIOP
Midlets (MIDP 3)	Domain based[3]	Yes	Socket-like
Xlets[4]	OS based	Yes	IXC (RMI)
JavaCard V.2.x Applets	Domain-like[5]	No	Direct access
JavaCard V.3 Applets, Connected edition	Namespace	No	Direct access

4 Isolation in OSGi

The OSGi framework is a dynamic service platform for the construction of modular Java applications, allowing the installation, uninstallation, update and startup of components and services with no application reboot. OSGi components are called bundles, which are the platform's unit of deployment consisting of jar files with

[3] If utilizing the CLDC MVM

[4] Considering the utilization of the CDC AMS

[5] There is no traditional classloading in JavaCard v2.2.2, but its isolation model through firewalls does not provide fault isolation, as in JSR121 domain isolation

custom manifest attributes for defining, for example, information about versioning, package (type) dependencies and provided packages. Optionally bundles may provide services, which are published in a central registry that can be queried by service consumers, allowing component collaboration with loose coupling through service interfaces. After service binding, the consumer code directly references the servant object, without any proxies

The isolation level in OSGi is in fact a sort of enhanced namespace isolation by means of individual class loader instances provided for each bundle. The class loading mechanism follows some policies for loading types, basically considering the information provided by the Import-Package and Export-Package manifest attributes. Instead of a child to parent visibility in a tree hierarchy, the class loading in OSGi is rather a graph (Fig. 2), where sibling class loaders may provide classes between them.

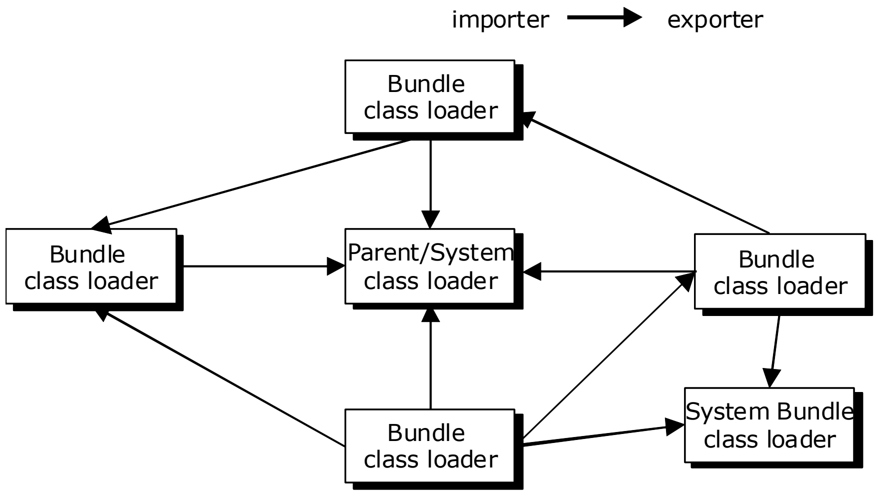

Fig. 2. Illustration of the classloading graph in OSGi.

Misprogrammed components are a concern since component developers that target the OSGi platform need to be aware of the dynamics in the environment, needing to appropriately release references to departed objects. When a service becomes unregistered, the referrer code is notified and must release the reference to the corresponding service. This unregistration may be due to a service implementation reacting (e.g. unregistering itself) to application or environment changes or due to the service providing bundle going to a process of deactivation (stopped, updated or uninstalled). Such type of dynamic events may happen frequently in such type of platform, and may incur problems known as stale references [7]. Static validation at load time would be a possibility for checking the occurrence of such problems, but that process is too costly, especially if the target platform runs on a resource limited device.

5 Dynamic Component Isolation

The OSGi specification tries to be as simple and lightweight as possible. Its direct object referencing brings the advantage of fast communication but it may impose problems if components do not handle correctly the inherent dynamicity of that platform. Other sorts of misbehavior, especially from untrusted third party components may also introduce faults, but such cases do not necessarily concern malicious components since a fault may happen due to lack of proper testing, or integration issues, for example. Security policies and class loading provide a limited level of isolation but no robustness.

Software-based isolation [40] introduces the concept of *sandboxes* for isolating in the process level untrusted modules and for providing fault containment, which is seen as a strategy for preventing error propagation across defined boundaries [27]. The term *sandboxing* has gained a wider sense throughout the years and now is often used to generally describe similar isolation techniques for preventing the underlying system to be harmed.

Therefore, in order to achieve such containment we needed to establish boundaries for separating components. Although there are custom VMs that provide object isolation in the Java platform through non-standard mechanisms, we wanted a solution focusing on technology that is standardized. We have chosen Java Isolates as our isolation boundaries for a few reasons: Isolates come from an official specification (JSR-121); its concepts are a trend for isolation and multitasking approaches that have been already tested with success in CLDC VMs; it continues to serve as an enabler for other features such as the ongoing effort of the resource consumption API [19] for Java.

Our component isolation mechanism for the OSGi platform tries to increase application robustness and dependability, in such a way that we can provide a sandbox where untrusted components are put in quarantine in a separate container where they can execute without harming (either intentionally or unintentionally) the application. In case a component misbehaves, or becomes stale, the sandbox can be restarted and the component can be safely terminated without needing to bring down the whole application. This type of isolation fills the well-isolated pre-condition for microrebooting [3] (individual rebooting of fine-grain application components). However, even if components are designed independently they are meant to collaborate as a part of a framework [28]. We have a means for isolating components but they still need to collaborate. Communication across boundaries is also provided in our approach, which is detailed in the next sub-sessions.

The implementation of the isolation solution described here was done in OpenSolaris with Sunlabs' Multitasking Virtual Machine[6] (MVM). We have patched the Apache Felix[7] v. 1.4.0 OSGi implementation for enabling the isolation solution using the JSR121 (Isolate API).

[6] http://mvm.dev.java.net
[7] http://felix.apache.org

5.1 Isolation Mechanism

The solution provided in [8] uses services as the grain of isolation. The principle is also applied here, but in a coarser grain. While that solution focuses on service isolation via proxies in the same VM, the one described here focuses on separating components in isolated domains. The semantics is the same, as generalized in the meta-model from Fig. 3.

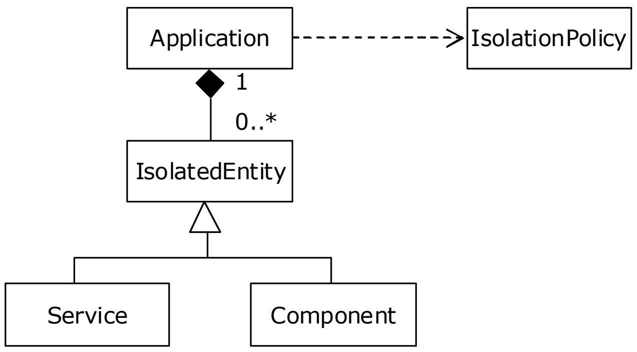

Fig. 3. Meta-model with the general isolation approach used either with services or components

At runtime, when a given entity (i.e. a service or a component) is about to be activated (retrieved in the case of a service, and started in the case of a component) then the system verifies if that entity must be isolated, and proceeds with the isolation process if necessary.

Since OSGi consists of a dynamic platform, our isolation model also needs to be dynamic. The sandboxing is done selectively, and at runtime, only for untrusted components which are then grouped in the sandbox. Faults are quarantined in that isolated container, and can not interfere in the main environment. The mechanism also allows the component to be promoted from the quarantine to the main platform during application runtime, currently upon human decision based on application behavior observation. For the dynamic component isolation, upon installation of an OSGi bundle, the customized framework performs a policy verification against the bundle jar file in order to know if it must be isolated or not. Implementations for the policy verification may vary: it can be done based on signed jar files, CRC verification, etc.

Our current policy implementation simply verifies a list of known jar files to see if the deployed jar is known. If the jar does not pass the verification, it is installed in the main platform but marked as “untrusted”. At startup time, when dependencies have been resolved, the platform will not start the untrusted jar in the main platform, but install that untrusted bundle and its dependencies on the sandbox and then perform the start up in the isolated environment, as illustrated in Fig. 4 showing that bundle C depends on types provided by bundle B which is also deployed in the sandbox but not started. As there is no dependency to Bunde A, there is no need to copy it to the sandbox. Thus, there are some rules concerning deployment:

– The main platform will always have all application bundles deployed, but no *untrusted* bundle in the *started* state;
– The sandbox will have untrusted bundles plus their dependencies, but no dependency in the *started* state, unless a dependency is also an untrusted bundle;
– No replicated bundle will be in the started state in both platforms at the same time.

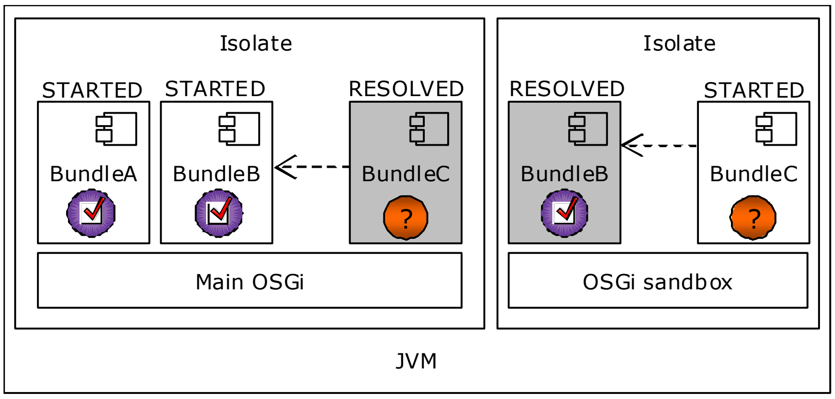

Fig. 4. Current deployment approach for dependency resolving in the sandbox. Untrusted bundles are also deployed in the main platform, but started only in the sandbox.

Although the duplication of components can increase memory footprint, and potentially load the same classes both in the main platform and in the sandbox, there are mechanisms that could reduce the cost of loading the same class representation multiple times in custom application class loaders [5], which is the case in OSGi. By hosting a component in a separate object domain (the sandbox) there will also be performance impact of the communication cost for trespassing the domain barrier if the sandboxed component needs to use services from a component in the main OSGi platform, and vice-versa.

5.2 Components Communication

In the OSGi platform, components establish communication through services which is done directly. In the case of our OSGi isolation mechanism, if an untrusted component needs to use a service provided by a trusted component, or vice-versa, the method calls need to cross the isolation boundaries that separate the service consumer and provider. This is done transparently by dynamic service proxies over a simple communication protocol which we have implemented. There is a two step process for retrieving a service instance in OSGi: query the service registry for a `ServiceReference` object, and then use that object for retrieving the actual service instance. In our implementation, if the requested service reference is not found in the local registry, the query is sent to the isolated platform. In case of a match the requestor gets an `IsolatedServiceReference` object, which is an instance of `ServiceReference`, and then requests the corresponding service which would be a dynamic proxy implementing the requested service interface. The proxy delegates the calls to the isolated platform using our protocol. The usage of service proxies evidently adds an overhead for proxy creation and subsequent method calls on the proxy. The service orientation without the communication overhead is one of the advantages of OSGi over other service oriented platforms. However, the goal of our proposed isolation mechanism is not to completely forbid the OSGi's standard proxyless service referencing. In the mechanism described here, the communication between services co-located in the same container is still done via direct object referencing.

This proof of concept has been implemented using `javax.isolate.Links`, a JSR-121 specific API for communicating between Isolates, but the communication layer has been abstracted in such a way that the details of the communication implementation can be easily changed to another approach. In doing so, the component isolation solution could be ported to non-multitasking VMs, but with the additional cost of a whole JVM footprint (besides its startup). The `Link` usage could be replaced by sockets, or even RMI. The protocol which we have written had a small set of messages for installing untrusted bundles in the sandbox; querying services in the isolated platform; invoking isolated services; sending framework events (Service and Bundle). This initial implementation focused on feasibility before giving any attention to performance. After performing simple microbenchmarks, we have identified that service calls using our protocol over the `Link` implementation did not perform better than ordinary RMI method calls outside the OSGi environment. More than 2/3 of the overhead was exactly in the synchronization of reading and writing on Links. Most likely this is due to the fact that the communication model used by Isolates is very simple and frequent transmission of messages may cause a large overhead [2]. Anyhow, optimizations can still be performed. Although it has outperformed our approach, if we choose to use RMI this implies in more complexity since it needs to add marker interfaces (`javax.rmi.Remote`) to the isolated services needing to be called across domain boundaries, methods need to throw a remote exepction and both isolated platforms (the main one and the sandbox) would need their own RMI object servers to be managed.

Crashes due to misbehaving code from untrusted component bring only the sandbox down, without any harm to the rest of the application. In such cases the platforms coordinator needs to bring back the sandbox and reestablish the communication channel that has been disrupted, by sending new and valid Link objects to the main platform. During this process, before restarting the sandbox and restablishing communication the main platform has to invalidate and to notify the departure of all isolated proxy clients and `IsolatedServiceReferences`.

In our controlled experiment we automatically deployed untrusted bundles in the remote platform after an attempt to install it in the main platform. The following test cases were performed on components providing services using primitive types:

- `StackOverflowErrors` intentionally fired have brought isolated components down along with the sandbox, but the sandbox could be automatically rebooted.
- Stale services (unregistered services that are still and erroneously being used by service consumers) would have their isolated proxies invalidated, similar to [8]. Misprogrammed sandboxed bundles did not prevent the unregistered services from the main platform from being appropriately released and garbage collected.

– Manual runtime promotion of sandboxed components to the main platform.

Although we have not addressed resource accounting, by using the sandboxing approach we can enable that in the component level if we consider isolate one component at a time. A multiple sandbox mechanism would add complexity concerning deployment and communication coordination, but would be a way to achieve fine grained isolation and resource accounting.

5.4 Current Limitations

As it was developed as a proof of concept, there are limitations in this isolation approach that were not yet addressed: policy implementation is just based on a file list, parameters and return types on proxied services are limited to primitives, Strings, and arrays of those types; the current isolation solution enables only one "shared" sandbox, where all untrusted components are executing. A misbehaving untrusted component will affect (e.g. stale services, stale threads, sandbox crash) all components that coexist in the sandbox. Only primitive, String attributes and properties map are available in `BundleEvent` and `ServiceReference` that come from an isolated platform. There is no unified identifier, that is, an untrusted bundle has a given id in the main platform, and most likely a different id in the sandbox platform.

6 Related Work

Other platforms already address the type of isolation we want to provide in OSGi applications. For instance, Microsoft COM components can be either loaded in the client application process or provided in an isolated process [25]. In the latter case, a surrogate process (dllhost.exe) can load the DLL and act as a server. Communication is transparently done via inter-process communication, bringing performance overhead but enabling fault isolation between the client and the component server. The Microsoft .NET platform addresses isolation as well, by means of Application Domains [35], which are like lightweight processes using the same concept that Java uses with Isolates. These domains have fault isolation: one domain can be terminated without interfering in the other domains' execution. Communication across domain boundaries is done in an RPC fashion where objects are sent via marshalling. Application domains can be dynamically loaded but would have unloading limitations if used for implementing a dynamic platform [6] such as OSGi.

Singularity [11] is a Microsoft research micro-kernel OS built with managed code. Instead of having processes isolation ensured by memory isolation, it uses software-isolated processes which have a communication overhead smaller than ordinary OS based process isolation. Secure object access is enforced by using static analysis (code is verified ahead of execution), and by not allowing run-time code generation.

A research [2] performed on alternatives for Java application isolation and resource accounting mentions component isolation as a means of preventing unwanted side effects and full resource reclamation. The paper provides an overview of the issue by presenting non-standard JVM solutions which try to tackle isolation and resource accounting. The custom isolation solutions presented would not be suitable to the dynamicity of OSGi, and even clash with its custom classloader approach. Our technique tries to be compliant with standardized VM mechanisms, such as JSR121 (which is also mentioned in that work) as well as enabling the architecture to work with multiple standard JVMs if no multitask VM is available. An experimental approach [14] uses the Isolate API and the MVM for improving isolation in a J2EE server. They evaluate different grains of isolation, like fine grained individual servlet isolation, and coarse grained isolation where they introduce J2EE application domains. Restructuring the code for isolating servlets individually was difficult, which lead them to discard the implementation of other fine grained isolation cases (e.g. EJBs). Coarse grain isolation of application domains combining the isolation of whole J2EE applications with the isolation of sub-servers (e.g. WebServer, Database, JMS) seemed to be a feasible choice for production servers.

Approaches like FreeBSD Jails [21] provide virtual environments that work as isolated compartments where a user have access only to processes and files from its own "jail" without having access to resources from other jails. Some approaches targeting isolation in the OSGi platform use virtualization [33, 26] as a way for isolating different customer platforms, that is, each provider would host their components and services in its own virtualized platform without accessing other providers' environment. Access to services from the underlying platform can be through a predefined subset [33] or transparently without restrictions [26]. However, the virtualization happens in the same JVM, where multiple OSGi platform instances execute. A malfunctioning component crashing in one platform would bring down all virtualized OSGi instances. Another mechanism [9] combines JSR121 concepts with an extensible VM. They present JVM domains which allow lightweight isolation with the possibility to identify to what domain (i.e. a bundle) an object belongs to. They took the design decision of keeping direct object referencing as a way to keep the fast communication that exists in OSGi, however boundaries for fault containment are not mentioned.

R-OSGi [32] deals with the communication between services located in OSGi platforms in different machines, with the advantage of not being bound to any OSGi implementation. Service consumer proxy bytecode is dynamically generated and loaded as a bundle into the platform, which significantly increases the number of executing bundles, but it is managed by the R-OSGi core. R-OSGi could also be seen as a way of OSGi component isolation, but there is no complete unawareness of distribution in the client code, which in the most transparent case still needs wrapper code to adapt a system to distribution. In our approach we want to leave the isolation decision to the executing platform based on the isolation policies.

Security policies are also a form of isolating a component from having access to certain application APIs or system resources. Although security is an optional layer in OSGi implementations, it adds fine grained access to services and types. It is possible, for example, to prevent one component from having access to another by declaring that one of its provided services needs access permission. A practical implementation of OSGi application isolation enforced by security policies is presented in [12]. However, security policies would not provide fault containment in that case.

7 Conclusions and Perspectives

In this paper we have analyzed component isolation in the Java platform and in the OSGi service platform, a dynamic component platform for Java. We have described our dynamic component isolation approach for OSGi, implemented on top of a mechanism based on an official standard for isolated domains in Java. Our architecture allows its extension for working with multiple JVMs if no multitask VM is available. The isolation approach we propose adds a fault contained sandbox for the execution of untrusted components, enhancing application robustness and dependability. Isolated components that misbehave or become stale can be microrebooted by restarting only the sandbox, without bringing down the whole application. However, choosing to enhance isolation levels between components implies in a trade-off where the cost for components communication increase. The initial mechanism constructed on top of JSR121 Links did not perform as well as standard RMI calls (outside an OSGi environment). Our base tests using a controlled environment have validated the dynamic isolation without losing the collaboration between isolated components. Tests verified isolated faults, automatic sandbox reboot, correct reclamation of unregistered services and dynamic promotion of untrusted code to the trusted environment. This type of isolation could enable resource accounting in the component level.

Next activities consist in working on the current limitations, especially the improvement of the communication between isolates and the support to complex types in interface methods, as well as implementing a two-level isolation which combines the in-VM service isolation via proxies with the present component isolation approach. Automation of the component's promotion from the sandbox to the main platform is also desired and also tests outside the controlled environment in existing OSGi based applications are also necessary in order to validate our approach in a real scenario.

Acknowledgements. Individual thanks to the anonymous reviewers for their valuable feedback and Laurent Daynès from Sun Microsystems for his review, suggestions on the paper, and the discussions about Isolates. Part of this work has been carried out in the scope of the ASPIRE project (http://www.fp7-aspire.eu), co-funded by the European Commission in the scope of the FP7 programme under contract 215417. The authors acknowledge help and contributions from all partners of the project.

References

1. Allamaraju, S. et al. Professional: Java Server Programming J2EE, Wrox Press (2001)
2. Binder, W.: Secure and Reliable Java-Based Middleware – Challenges and Solutions. In: 1st International Conference on Availability, Reliability and Security. ARES, pp. 662--669, IEEE Computer Society, Washington, DC (2006)
3. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. 2004. Microreboot — A technique for cheap recovery. In: 6th Conference on Symposium on Opearting Systems Design & Implementation (2004)

4. Czajkowski, G., Daynès, L.:. Multitasking without Compromise: a Virtual Machine Evolution. In: the 16th conference on Object-oriented programming, systems, languages, and applications (OOPSLA), pp 125--138, New York, USA (2001)
5. Daynès, L., Czajkowski, G.: Sharing the runtime representation of classes across class loaders. In: the European Conf. on Obj. Oriented Progr. Glasgow, UK (2005)
6. Escoffier, C., Donsez, D., Hall, R. S.: Developing an OSGi-like service platform for .NET. In: Consumer Comm. and Networking Conf., pp. 213--217, USA (2006)
7. Gama, K., Donsez, D.: A Practical Approach for Finding Stale References in a Dynamic Service Platform. In: CBSE 2008. LNCS, vol. 5282, pp.246--261, Springer Berlin/Heidelberg (2008)
8. Gama, K., Rudametkin, W., Donsez, D.: Using Fail-stop Proxies for Enhancing Services Isolation in the OSGi Service Platform. In: MW4SOC'08, pp.7--12, ACM, New York, NY (2008)
9. Geoffray, N., Thomas, G., Folliot, B., Clément, C.: Towards a new Isolation Abstraction for OSGi. In: the 1st Workshop on Isolation and integration in Embedded Systems. M. Engel and O. Spinczyk, Eds. IIES '08. ACM, New York, NY, pp 41--45 (2008)
10. Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., Watson, T.: The Eclipse 3.0 platform: Adopting OSGi technology. IBM Systems Journal 44(2), pp 289--300 (2005)
11. Hunt, G. et al: An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research (2005)
12. Jahn, M., Terzic, B., Gumbel, M.: Do not disturb my circles – Application isolation with OSGi. OSGi Community Event, Berlin (2008)
13. Java Card Technology. http://java.sun.com/javacard/
14. Jordan, M., Daynès, L., Jarzab, M., Bryce, C., and Czajkowski, G. : Scaling J2EE™ application servers with the Multi-tasking Virtual Machine. Softw. Pract. Exper. 36 (6) May. 2006, pp. 557—580 (2006)
15. JSR 121: Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121
16. JSR 195: Information Module Profile. http://jcp.org/en/jsr/detail?id=195
17. JSR 217: Personal Basis Profile 1.1. http://jcp.org/en/jsr/detail?id=217
18. JSR 271: Mobile Information Device Profile 3. http://jcp.org/en/jsr/detail?id=271
19. JSR 284: Resource Consumption Management API. http://jcp.org/en/jsr/detail?id=284
20. Kalaimagal, S.,Srinivasan, R.: A retrospective on software component quality models. SIGSOFT Software Engineering Notes 33, 6 Oct. 2008, pp. 1--10 (2008)
21. Kamp, P. H., Watson, R. N. M.: Jails: Confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference (2000)
22. Kwiatek, M.: Cluster Architecture for Java Web Hosting at CERN. In: the 15th International Conference on Computing In High Energy and Nuclear Physics, Mumbai, India, pp.528--531 (2006)
23. Laprie, J., Randell, B.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Secur. Comput. 1, 1, pp. 11--33 (2004)
24. Liang, S., Bracha, G.: Dynamic Class Loading in the Java Virtual Machine. In: OOPSLA'98, pp. 36--44 (1998)
25. Lowy, J. 2001 COM and .NET Component Services. 1st. O'Reilly & Associates, Inc.
26. Matos, M., Sousa, A.: Dependable Distributed OSGi Environment. In: MW4SOC'08, pp. 1--6, ACM, New York, NY (2008)
27. Nelson, V.P.: Fault-Tolerant Computing: Fundamental Concepts. In: IEEE Computer, 23(7): pp 19--25 (1990)
28. Nierstrasz, O., Dami, L.: Component-Oriented Software Technology, Object-Oriented Software Composition, Prentice Hall (1995)
29. OSGi Alliance. http://www.osgi.org
30. OSGi Alliance. About the OSGi Service Platform, Technical Whitepaper Revision 4.1, 7 June 2007, http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf

31. Parrend, P., Frénot, S.: Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. In: CBSE 2008. LNCS, vol. 5282, pp.80--96, Springer Berlin/Heidelberg (2008)
32. Rellermeyer, J. S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications through Software Modularization. In: the ACM/IFIP/USENIX 8th International Middleware Conference (2007)
33. Royon, Y., Frénot, S., Mouel, F. L.: Virtualization of Service Gateways in Multi-provider Environments. In: CBSE 2006, pp 385--392. Springer Berlin/Heidelberg (2006)
34. Schmidt, H.:Trustworthy components-compositionality and prediction. Journal of Systems Software. 65, 3 (Mar. 2003), pp. 215-225.
35. Stutz, D., Neward, T., and Shilling, G. Shared Source Cli Essentials. O'Reilly (2002)
36. Sun Microsystems. Multitasking Guide-Sun Java Wireless Client Softw., Version 2.1, JME. 04/2008, http://java.sun.com/javame/reference/docs/sjwc-2.1/pdf-html/multitasking.pdf
37. Sun Microsystems. The CDC Application Management System. White Paper, June 2005. http://java.sun.com/j2me/docs/cdc_appmgmt_wp.pdf
38. Squawk Java ME VM. https://squawk.dev.java.net/
39. Szyperski, C, Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition (2002)
40. Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L.: Efficient software-based fault isolation. In: the 14th ACM Symposium on Operating Systems Principles. SOSP '93. pp. 203--216. ACM, New York, NY (1993)