Towards a new Isolation Abstraction for OSGi

Nicolas Geoffray
Université Pierre et Marie Curie - LIP6/CNRS/INRIA - Regal
104 avenue du Président Kennedy
75016 Paris, France
nicolas.geoffray@lip6.fr

Gaël Thomas
Université Pierre et Marie Curie - LIP6/CNRS/INRIA - Regal
104 avenue du Président Kennedy
75016 Paris, France
gael.thomas@lip6.fr

Charles Clément
Université Pierre et Marie Curie - LIP6/CNRS/INRIA - Regal
104 avenue du Président Kennedy
75016 Paris, France
charles.clement@lip6.fr

Bertil Folliot
Université Pierre et Marie Curie - LIP6/CNRS/INRIA - Regal
104 avenue du Président Kennedy
75016 Paris, France
bertil.folliot@lip6.fr

ABSTRACT

The OSGi specification defines a dynamic Java-based service oriented architecture for networked environments such as home service gateways. To provide isolation between different services, it relies on the Java class loading mechanism. While class loaders have many advantages beside isolation, they are poor in protecting the system against malicious or buggy services. In this paper, we propose a new approach for service isolation. It is based on the Java isolate technology, without a task-oriented architecture. Our approach is more tailored to service-oriented architectures and in particular offers a complete isolation abstraction to the OSGi platform.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Security and Protection; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Languages, Design

Keywords

Java, OSGi, Isolation, Class loading

1. INTRODUCTION

Class loaders in the Java programming language offer an easy and convenient approach to isolate multiple applications in the same instance of a Java Virtual Machine. Applications are loaded by different class loaders, hence isolating types in the form of namespaces. They also provide lazy loading, type-safe linkage and user-definable class loading [12]. Class loaders are extensively used by Java Web applications to isolate different software components such as web applets, servlets or JavaBeans.

The Open Service Gateway Initiative (OSGi) [1] platform also extensively uses class loaders. OSGI is a service-oriented architecture where loosely coupled software modules offer and use services. In the OSGi terminology, these services are packaged in deployment units called `bundles`. Initially, OSGi targeted home service gateways, but it expanded to automotive and mobile telecommunications, and even to development applications such as Eclipse. The main feature of OSGi in the domain of embedded systems is the support for dynamic services deployment and life cycle. Furthermore, it provides remote management, which is critical for embedded devices.

Bundles in OSGi are isolated entities, loaded by different class loaders. A bundle can provide and/or request services. When a newly-loaded bundle offers services, they are published in the service registry. Accordingly, when a bundle requests services, it discovers them through the service registry. Bundles communicate through direct method invocation. When a bundle is no longer needed (for example when it is updated, or the physical resource it may manage is disconnected), it can be de-activated and removed from the system.

Even though the architecture is attractive, the implementation has its limits due to the internal mechanisms and the properties of class loading in the JVM [12, 3]. The isolation offered by class loaders is limited: it breaks in the case of static variable usages, and static synchronized calls. For example, if one malicious bundle overwrites the `System.out` global variable, the other bundles are affected.

In order to tackle the class loading limits for isolation, Czajkowski [3, 4] proposed the notion of isolates. Originally, isolates are tasks that execute in the same virtual machine. Each isolate has its own private static variables, thus preventing any interference between tasks.

Isolate, however, is not the right abstraction for OSGi. An isolate is an executing task, whereas a bundle in OSGI can be viewed as a library. The implementations of the OSGi platform may be single-threaded. Furthermore, communications between bundles are direct method calls, whereas isolates only provide IPC mechanisms with expansive parameter copy.

In this paper, we propose a modified Java Virtual Machine with a hybrid approach between class loaders and isolates that we call `domains`. We maintain class loaders because they give the possibility for users to define how classes are loaded (e.g. network, file system, etc.). We also keep the isolate approach, however without the burden of executing tasks. Therefore we have both strengths of isolates and class loaders, without the isolation flaws in the case of class loaders and without imposing executing tasks in the case of isolates. The key features of our solution are:

1. We provide a stronger isolation between bundles: each bundle has its own private environment.
2. We keep the class loading mechanism so that applications can redefine how classes are loaded.
3. We keep the fast communication properties of OSGi with direct method invocations.
4. Applications do not need to be modified.

The remainder of the paper is organized as follows: Section 2 presents OSGi and the isolate specification, their approach for isolating applications, and their limits. Section 3 presents our approach and Section 4 describes its implementation. Section 5 discusses related work and Section 6 concludes the paper.

2. ISOLATION MODELS

The Java programming language offers two different approaches for application isolation: classloaders, which are part of the virtual machine specification, and isolates which constitute an external JSR. In this section, we briefly describe how OSGi uses class loaders to isolate bundles, and how the isolates specification provides full isolation between programs.

2.1 Class loaders in OSGi

The Java Virtual Machine loads class through instances of the `ClassLoader` class. Class loaders all have a parent class loader to which it delegates class loading by default. If the parent can not load the class, the loader loads the class itself. Applications can provide subclasses of the `ClassLoader` class to specialize how classes are loaded: for example, they can be looked up on a remote disk, in memory or on a network, and they can instrument the classes to dynamically modify them. The bootstrap class loader acts as a last parent for class loaders and is in charge of loading system classes like the `java.lang.String` class. Applications can not specialize the bootstrap class loader. Class loaders share the same instances of system classes, to prevent hijacking the base classes types.

A class is always associated with its class loader. The same class can be loaded by two different class loaders, and the virtual machine will consider the classes as two different types. Hence, class loaders provide namespaces, in order to prevent name collisions in the same application. Typically, when an application receives mobile codes (i.e. applets), it will load them in different class loaders, so that class names in mobile code A will not interfere with class names in mobile code B.

OSGi uses class loaders for many purposes. First of all, class loaders protect type names between bundles, so that classes in one bundle will not conflict with classes in another one. Second of all, it allows the complete removal of a bundle from memory: when a bundle is unloaded, its class loader and the classes it loaded can all be removed (if other bundles do not have references to services exported from this bundle). Finally, each class loader is associated a class loading policy, which dictates how to fetch classes.

The class loading mechanism in OSGi is different than the standard mechanism in the JVM. The latter uses a hierarchical search order, whereas OSGi uses a graph to load classes [9]. Each bundle has a set of exported and imported classes (which can be empty). Importing a class means that the bundle will use the class definition of the class loader from another bundle. Exporting a class means that a bundle allows other bundles to use the class definition. The platform manages the dependencies between the bundles, hence class loaders. The import and export informations are stored in a metadata file archived with the class files. They are encapsulated in a `.jar` file that represents a bundle.

In terms of isolation between bundles, class loaders only provide weak guarantees. Malicious bundles have many opportunities to prevent correct execution of other bundles: for instance modifying static variables such as `System.out` or synchronizing on shared objects such as static variables, shared interned strings[1] or `java.lang.Class` instances. On a wider perspective, malicious bundles can even perform denial of service attacks against resources like memory or CPU.

2.2 Isolates

The isolate specification directly deals with attacks on shared variables between different applications. The specification does not however deal with memory or CPU attacks.

The isolate specification has its roots in the multitasking virtual machine [4] (MVM). MVM is a modification of the Sun JVM that enables the execution of multiple Java applications in the same JVM, in a "lightweight" isolated fashion. Each application has its own static variables, instances of `java.lang.Class`, string hash table and so on. When an application ends its execution, MVM reclaims all its memory.

[1]While it is not advised to use synchronization of string literals, it is nonetheless correct Java.

OSGi requires fast communication between bundles. Therefore, in this paper, we only consider the implementation of isolates in a single address space to enable direct method calls.

In MVM, the JIT generated code is shared between isolates. This enables better memory usage and faster execution startup of isolates. However, the compilation of some bytecodes must change in order to protect isolates from one another. For instance, to access a static variable (`putstatic` or `getstatic`), the code fetches an isolate identifier, which will be used to look up the variable in a *task class mirror* (TCM). A TCM is an array of instances of the same static variable, each for one isolate.

MVM does not prevent the use of class loaders. Classes loaded by application-defined class loaders are not shared between isolates, therefore there is no need to have a TCM for these classes.

Isolates are tasks, which means each isolate is an executing entity with at least one thread. Each thread is associated with an isolate, and fetches the isolate identifier to access its private environment. This enables lightweight isolation between executing tasks. Compared to class loaders, isolates have the advantages of completely protecting applications against malicious downloaded code. Nonetheless, it imposes that each isolate has an execution environment (e.g. thread, stack).

In the case of OSGi, bundles are not executing tasks. They are software components calling each other. They have their own environment, materialized by class loaders. Furthermore, isolates can communicate via Links [11] or via XIMI [13]. Links only allow sending and receiving primitive types (e.g. integers, byte arrays, strings), and XIMI performs a deep copy of arguments. This is not suitable for OSGi which prefers fast communications. To gain lightweight isolation and fast communications between bundles, a service-oriented platform needs a new abstraction.

3. OUR APPROACH

Our solution is a JVM improved with `domain` possibilities. A domain is an instance of a JVM, but is not related to any executing task. Multiple domains co-exist in the same OS process. As with class loaders, domains share the system classes (and their jitted methods). They do not share the classes they load by default: each domain has its set of classes. Like isolates, a domain has its own set of static variables, `java.lang.Class` instances and interned strings[2]. This enables lightweight isolation between domains.

As in OSGi, domains can export and import classes. Exporting a class means the domain allows other domains to use the runtime representation of this class. Importing a class means the domain asks other domains for its runtime representation.

3.1 Lightweight isolation

[2]Note that this will break the JVM string specification (section 3.10.5 of the Java Language Specification 2.0) of interned strings, *when domains communicate*.

The key idea to implement lightweight isolation and fast communications between bundles, is that each Java method must be able to know which domain its defining class was loaded by. It uses the domain to fetch its private environment. A domain is a virtual machine internal object, and has a unique identifier, just like isolates. The difference with isolates is that a domain is not thread-oriented, but method-oriented. A method *belongs* to a domain when its defining class was loaded by this domain.

We differentiate two kinds of classes:

1. Classes loaded by application domains. In OSGi, these are the classes defined by a bundle.
2. Classes shared between application domains. There are the system classes and classes that provide common facilities Theses classes are loaded in a *shared domain*.

Consider a method M1, that belongs to domain D1, which calls method M2. If M2 belongs to an application domain D2, execution switches to domain D2. If M2 belongs to the shared domain, execution continues with domain D1.

3.2 Resource accounting

It would also be valuable if the JVM could make domains accountable of the resources they use, mainly CPU and memory. Since domains are not task- or thread-oriented, CPU accounting is difficult to achieve, if it makes sense: when a service calls another service, CPU time must be accounted to the caller or the callee? We are still investigating possibilities in this area.

For memory accounting, when a method executes it knows the current domain. Therefore, on allocation opcodes such as `new`, `newarray` or `anewarray`, or on allocations performed in the JVM runtime, the code knows to which domain the allocation can be accounted to.

There are several research projects which perform resource accounting in Java on a per-task basis [8, 5, 3, 4, 10, 16]. They all use an *execution environment* object (for instance an isolate) which indicates to which task the resource has to be accounted to. In our approach, the execution environment object is a domain, and the memory is accounted to the domain. The garbage collection and allocation techniques used in the aforementioned projects can be the same.

4. IMPLEMENTATION

We plan on implementing our approach on our JVM prototype, called JnJVM [15]. No modifications are needed on existing OSGi applications. Our implementation modifies the just in time compiler of the JVM, transparently for applications.

We modify the just in time compiler to add a new parameter to each Java method. This parameter is a domain object, which will be given to each call to a Java method or to a JVM runtime method. With this parameter, the method knows its current domain and its identifier.

Fetching private Java static variables and `java.lang.Class` instances is done in the same way than in the MVM [4]: each

class has a `domain class mirror` array, and the domain's identifier is an offset in this array to access the private instances.

Fetching a string is different. When compiling a `ldc` bytecode that loads a string, the compiler emits code to lookup the string in the domain's table of interned strings. This comes at a higher cost than direct access, which is the default compilation of `ldc` bytecode for strings, but we gain stronger isolation guarantees between domains.

We also modify the compiler to add a new local variable to Java methods whose class has been loaded by domains. This local variable will point to the domain the method belongs to. At compile time, the compiler knows to which domain this method belongs to, because its defining class has already been loaded. The local variable overrides the new parameter, and will be given to each Java or runtime call in this method. This ensures that when executing a method, we use the correct domain object to access the domain private environment. Finally:

1. When executing methods whose class has been loaded by a domain, the domain object used is the local variable.

2. When executing methods whose class has been loaded by the shared domain, the domain object used is the parameter.

An application always starts with the `main` method of an application domain A. A is the first domain to be used. When A will call services from other domains, it will give its domain object as parameter, but the called service method will use its local domain object. When A will call methods from the shared domain, the called methods will use the parameter domain object.

5. RELATED WORK

Isolating different software components in the same address space has its roots into single-address space operating systems and type-safe languages [2, 14, 6]. In the case of the Java programming language, several research projects built systems that provide stronger isolation guarantees than class loaders [8, 5, 3, 4, 10, 16]. All these systems concentrate on task isolation, termination and communication, as well as resource accounting. Isolation between different non-executing software components is only provided by class loaders, which is the basic isolation mechanism in OSGi. Class loaders have the advantages of fast communications, simplified task termination at the cost of less isolation guarantees.

Java uses a similar technique than ours for security inside the virtual machine: when a method wants to know the protection level of its caller, it inspects the stack trace to find which class loader loaded the class of the caller. A class loader can have a security policy, which indicates what kind of privileged operations classes loaded by this class loader can perform. Inspection of a stack is an expensive operation, which will drastically reduce performance of our approach if performed on each static variables, strings and `java.lang.Class` loads. Therefore, we explicitly give the domain as an argument or as a local variable.

In order to avoid the Java class loading limitations, Escoffier *et al* made several attempts to provide an OSGi-like framework in .Net [7]. .Net offers the notion of application domains, which is very similar to isolates. An application domain is an executing task, which executes an assembly (a set of classes) with a `Main` method. Hence .Net has the same weaknesses than isolates: application domains are task-oriented and have poor communication performance compared to a simple method call. Furthermore, unloading an assembly in .Net is more complex than unloading classes of a class loader [7].

6. CONCLUSION

The OSGi platform is gaining widespread acceptance as a dynamic service platform. Its application goes from home service gateways and automotive environments to plugin manager in the Eclipse Java development tool. There are however deficiencies in its isolation model based on class loading. None of the current technologies, either in Java or in .Net, provides the adequate abstraction: they all are task-oriented instead of method-oriented.

In this paper, we propose a new approach to isolation in the OSGi context. We introduce the notion of `domain`, which is equivalent to a Java isolate, but is not associated to an executing task. A domain is associated to a set of classes (a service, or a bundle in OSGi). Domains execute in the same process and in a single Java virtual machine. This keeps the requirement of fast communication between bundles with direct method calls. We are currently implementing our concepts in an extended JVM to execute a modified OSGi runtime. We think we will achieve stronger bundle isolation guarantees with small and located performance impact.

7. REFERENCES

[1] OSGi Alliance. www.osgi.org.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Symposium on Operating Systems Principles*, pages 267–283, Copper Mountain, Colorado, United States, 1995. ACM Press.

[3] G. Czajkowski. Application Isolation in the Java Virtual Machine. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications Conference*, pages 354–366, Minneapolis, USA, 2000. ACM Press.

[4] G. Czajkowski and L. Daynés. Multitasking without Compromise: a Virtual Machine Evolution. *SIGPLAN Not.*, 36(11):125–138, 2001.

[5] G. Czajkowski and T. Eicken. JRes: a Resource Accounting Interface for Java. *SIGPLAN Notice*, 33(10):21–35, 1998.

[6] S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. The Inferno Operating System. *Bell Labs Technical Journal, Volume 2, Number 1*, pages 5–18, 1997.

[7] C. Escoffier, D. Donsez, and R. S. Hall. Developing an

OSGi-like Service Platform for .NET. In *Proceedings of the Consumer Communications and Networking Conference*, Las Vegas, CA, USA, January 2006. IEEE Computer Society.

[8] G. Back and W. H. Hsieh and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, San Diego, USA, October 2000.

[9] R. Hall. A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In *Proceedings of the International Working Conference on Component Deployment*, pages 81–96. Springer, May 2004.

[10] C. Hawblitzel and T. Eicken. Luna: a Flexible Java Protection System. *SIGOPS Operating Systems Review*, 36(SI):391–403, 2002.

[11] Java Community Process. JSR-121: Application Isolation API Specification. http://jcp.org/jsr/detail/121.jsp. by Sun Microsystem.

[12] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 36–44, 1998.

[13] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynes. Incommunicado: Efficient Communication for Isolates. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 262–274, Seattle, Washington, USA, 2002. ACM Press.

[14] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, Queen's College, University of Cambridge, Cambridge, UK, 1995.

[15] G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a Flexible Java Runtime upon a Flexible Compiler. In J.-J. V. David Simplot-Ryl and G. Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.

[16] P. Tullmann. The Alta Operating System. Master's thesis, University of Utah, Utah, USA, 1999.