Flexible Class Loader Framework: Sharing Java Resources in Harness System

Dawid Kurzyniec and Vaidy Sunderam

Emory University, Dept. of Math and Computer Science
1784 N. Decatur Rd, Atlanta, GA, 30322, USA
{dawidk, vss}@mathcs.emory.edu

Abstract. Harness is a Java-centric, experimental metacomputing system based on the principle of dynamic reconfigurability, not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via a "plug-in" mechanism that is the central feature of the system. This "plug-in" mechanism relies on the underlying class loader services which enable loading and accessing Java classes and resources from a set of software repositories. However, the specific class loading requirements imposed by Harness cannot be satisfied by any of the class loaders provided as a part of Java platform. In this paper we describe new, flexible, class loading techniques which are primarily intended to improve plug-in management and security aspects in the Harness system but can also benefit other applications, especially those collaboratively accessing shared Java resources.

1 Introduction

Harness [5,11] is a metacomputing framework based upon such experimental concepts as dynamic reconfigurability and extensible virtual machines. The underlying motivation behind Harness is to develop a metacomputing platform for the next generation, incorporating the inherent capability to integrate new technologies as they evolve. The first motivation is an outcome of the perceived need in metacomputing systems to provide more functionality, flexibility, and performance, while the second is based upon a desire to allow the framework to respond rapidly to advances in hardware, networks, system software, and applications.

Harness attempts to overcome the limited flexibility of traditional software systems by defining a simple but powerful architectural model based on the concept of a software backplane. The Harness model is one that consists primarily of a kernel that is configured by attaching "plug-in" modules that provide various services.

As is natural for a Java-centric system, "plug-in" modules are basically sets of related Java classes and possibly other resources cooperating to provide the desired service. Thus, the ability of Harness to attach "plug-in" modules is based upon the underlying class loader's capabilities to locate and load classes from a dynamically changing set of software repositories. The current implementation of the Harness class loader fails to satisfy two desired features: supporting loading classes from Java archive (JAR) files, and supplying a uniform and reliable caching mechanism to reduce potentially expensive network access.

This paper describes the design and implementation of a flexible class loader framework and its application in Harness system. Although initially designed to support Harness, it is generic and powerful enough to be successfully used in other applications, which include, but are not limited to, the systems collaboratively accessing shared and dynamically changing Java resources. The remainder of this paper is organized as follows. Section 2 provides a general discussion of the role of class loaders in the Java language. Section 3 presents related work. In Section 4 we describe the architecture of the flexible class loader framework; its intended application to the Harness system is presented in Section 5. The paper concludes (Section 6), with an outline of future work plans.

2 Class Loaders in Java

The concept of a class loader evolved together with the Java technology. Primarily it was designed to allow users to load Java VM classes from various non-standard sources such as those downloaded from the network, fetched from a database, or even generated on-the-fly. In JDK 1.0.x, with the API[1] consisting of only four methods loading classes was its only purpose. JDK 1.1 incorporated extensions to the class loader to find and load additional resources such as images, icons, sounds, and property files, which could accompany classes used in an application. Other improvements included an an attempt to extend the Java security model by introducing class signers.

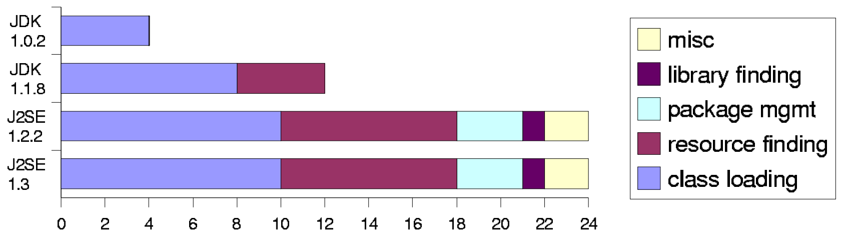


Fig. 1. Class Loader API evolution

The Java 2 platform brought substantial changes to class loader capabilities. The semantics were refined [10] and the design has been tightly integrated with the newly established Java Security API [12]. Two new

[1] Thorough this paper, we use term "class loader API" to denote `public` and `protected` methods of the class `java.lang.ClassLoader`.

classes were added: `java.security.SecureClassLoader`, intended as a base for user-defined class loaders taking advantage of Java security model, and `java.net.URLClassLoader`, which enables loading data from Java archive (JAR) files [7] and includes support for downloadable extensions [2]. Moreover, in addition to finding and loading classes and resources, the class loader became responsible for defining and keeping track of packages and for controlling native library linking. Additionally, the notion of parenthood and delegation was introduced for class loader objects: every class loader is encouraged to delegate class loading requests to its parent before it tries to load classes by itself. The class loader API in Java 2 consists of 24 methods as compared to 4 methods in the original version, highlighting the scale of changes that class loaders have undergone. The evolution of the class loader API is illustrated in Fig.1.

Prior to Java 2, native libraries were almost completely handled in native VM code. This was restrictive to library management: to be loaded by the VM, libraries had to be present in some VM-specific, designated place, typically somewhere in the local file system (e.g. within LD_LIBRARY_PATH on UNIX platforms). Applications which required more complex library management such as dynamic download were forced to adapt sophisticated approaches like preliminary static analysis of the bytecode of a class in order to scan for native library dependencies [4], or use security managers merely for code downloading purposes.[2] In Java 2, native libraries are loaded by class loaders rather than by the Java VM itself. Much of the native library handling code was rewritten in Java giving developers an opportunity to handle library lookup requests in an application-specific way, which provides more flexibility in native library management.

3 Related Work

The `java.lang.URLClassLoader` is a class loader introduced in Java 2 to address loading classes and resources not only from ordinary directory structures but also archived in JAR files. It maintains a search path consisting of a set of base URLs pointing either to directories or JAR files and it searches for classes and resources using this path. It also understands downloadable extensions, so it allows JAR files to add URLs of other JAR files that they depend on to its search path. Such a scheme fits well the industrial software distribution model where code is bound tightly to an organization providing the software, but does not necessarily fit other models, e.g. when the software is stored in large repositories and accessed dynamically as in case of Harness system. The `URLClassLoader`'s class and resource searching algorithms are fixed, making it inappropriate as a base class for implementing such distribution models. The Java platform thus lacks a class loader which would be both flexible and JAR-aware. Also, no class

[2] Harness did this using `SecurityManager.checkLink(String)` method; see `http://java.sun.com/j2se/1.3/docs/api/java/lang/Runtime.html#load(java.lang.String)`.

loader-related proposal can be found in the Java Specification Requests area [9] in Sun's Java website.

Although there are many systems providing customized class loaders to satisfy application-specific requirements [3,6], we do not know about any efforts to develop an architecture that would be generic and customizable enough to be reusable across different systems.

4 Flexible Class Loader Framework

Class loaders are used when there is a need for some non-standard, application-specific way to find classes, resources, or libraries. Nevertheless, such specialized class loaders often have much in common. For example, common tasks might include reading data from JAR files or downloading classes from the network; similarly, many class loaders do not have separate routines for dealing with classes and resources but access them in a uniform way. These similarities, however, do not comprise a common denominator. Therefore, full advantage can be taken of a generic class loader only when its structure is *modular*, so that users can choose and configure the modules they require in their applications.

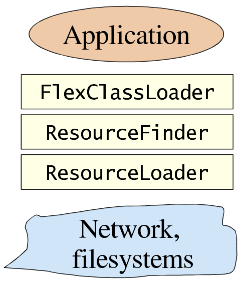


Fig. 2. Structure of Flexible Class Loader Framework

The flexible class loader framework described in this Section presents an example of such a modular structure, as illustrated in Fig.2. It is based on the observation that the typical class loader, apart from carrying out its low level function, must perform independent activities like searching for requested data, downloading data and classes from the network and managing JAR files. Taking this into account, it is reasonable to split the functionality into three layers:

- the class loader itself, serving as an interface to an application,
- the resource finder, encapsulating the searching algorithm but unaware of differences between classes, libraries, and other resources,
- the resource loader, responsible for managing JAR files and downloading data from the network.

The most specific part of each class loader is its resource searching algorithm. For that reason, many sophisticated class loaders can be derived from this scheme just by customization of the resource finder layer. In the following subsections, we describe all three layers in more detail.

4.1 `FlexClassLoader`

The `FlexClassLoader` class is the class inherited from `SecureClassLoader`. Its functionality its intentionally similar to that of `java.net.URLClassLoader` – the code for defining classes and packages is virtually the same – though it is intended to be generic and customizable. It does not perform data searching operations, but relies instead on the capabilities of its underlying resource finder (provided as a constructor argument).

The flexible class loader framework introduces the notion of *resource handles*.[3] A resource handle is a set of meta-information related to some resource, regardless of whether it is a Java class, native library, image, text file or any other entity. The resource handle provides such information as the resource name, its URL (and origin URL in cases when resource was cached), the URL of its code source (and its origin version as appropriate), and finally the certificates and attributes of the resource as well as the JAR manifest in case the resource is from a JAR file. Furthermore, the resource handle allows loading of the resource, either through the provided input stream or directly into the `byte[]` array. Note, however, that the handle does *not* include the resource; rather, it represents a connection to the resource – and there is no guarantee that resource loading will succeed after the handle is obtained. The `FlexClassLoader` has four designated methods enabling its subclasses to take advantage of resource handles:

```
protected ResourceHandle findClassHandle(final String name);
protected ResourceHandle findResourceHandle(final String name);
protected Enumeration findResourceHandles(final String name);
protected ResourceHandle findLibraryHandle(final String name);
```

The `FlexClassLoader` assumes that classes, resources, and libraries should be searched for in the same way. Its methods related to class and library loading simply transform the class or library names to resource names and perform generic resource lookup using single resource finder, as described in the next Section. If this is not appropriate for some particular implementation that wishes to handle classes, resources, and libraries in a specific way, these methods may be overridden to provide the desired functionality, perhaps using designated resource finders.

4.2 `ResourceFinder`

The resource finder is the entity related to the most specific feature of each class loader – the resource lookup algorithm. In the flexible class loader framework, the entity is represented by the `ResourceFinder` interface, which should be implemented by users to provide a specific searching approach:

[3] The concept is based on the class `sun.misc.Resource`.

```
public interface ResourceFinder {
    /** Returns ResourceHandle of resource with specified name
     *  or null if the resource cannot be found. */
    ResourceHandle findResource(String name);
    /** Returns enumeration of ResourceHandle objects representing
     *  all resources with specified name. */
    Enumeration findResources(String name);
}
```

An important feature of this layer of the framework is that it does not distinguish between types of resources it is asked to search for – resource finders handle only simple, unified queries. Therefore, this two-method interface is sufficient to handle all requests of the `FlexClassLoader`.

A number of customized class loaders are being designed to support loading data from the network. To limit the overhead of accessing data through a potentially slow network connection, it is often desirable to provide a caching mechanism which would allow the storage of downloaded data on the local file system. To address this issue, the flexible class loader framework provides the abstract class `CachingResourceFinder`. To take advantage of this class, users must implement its two abstract methods:

```
public abstract class CachingResourceFinder
    implements ResourceFinder
{
    ...
    protected abstract ResourceLoader getResourceLoader();
    protected abstract Iterator getSearchEntries(String name);
    ...
}
```

The first method should return an instance of the `ResourceLoader` object (as described in next Section) intended to manage resource loading. By implementing the second method, users specify an actual resource lookup algorithm, providing the iteration over consecutive URLs at which the given resource should be subsequently searched for. Both the `findResource()` and `findResources()` methods needed to implement the `ResourceFinder` interface are implemented in `CachingResourceFinder` and they assume that the resource lookup algorithm is simple enough to fit in the scheme of a series of URLs to search for. For more sophisticated lookup algorithms users can override those methods and still take advantage of low-level caching functionality provided by this class.

The caching policy implemented in the `CachingResourceFinder` uses the designated cache directory where it replicates the remote directory structures as resources are loaded from the network. When the specific resource is searched for, the search iterator returned by the `getSearchEntries()` method is iterated twice: in the first pass, the URLs are converted to their cache equivalents so the resource is searched for in the cache. The second pass goes through original URLs; if the resource is found, it is downloaded and stored in the cache for future use.

4.3 `ResourceLoader`

The bottom layer of the flexible class loader framework is represented by the `ResourceLoader` class:

```
public class ResourceLoader {
    ...
    public ResourceHandle findResource(URL base, String name) { ... }
    ...
}
```

This class is a uniform and convenient way to load resources from the file systems and the network, and additionally provides support for JAR files; this support includes attaching appropriate meta-information to resource handles as the meta-information is fetched from the JAR file manifest.

Handling JAR files requires special care. On one hand, the process of decompressing a JAR file, which often involve verification of digital signatures, can be time consuming, so it is reasonable to keep JAR files opened for some period of time in order to speed-up subsequent resource fetching. On other hand, however, opened JAR files can consume a lot of system memory, so they should be closed as soon as possible. The `ResourceLoader` tries to balance these contradictory goals as follows. It keeps the specific JAR file opened as long as there are any live resource handles representing resources loaded from it. When there are no more such handles, instead of being immediately closed the JAR file is moved to a special, limited-size cache pool. The maximal size of the cache pool may be specified by the user and it can be as low as 0 (so that JAR files are closed immediately when they are no longer referenced by resource handles) or may be set to infinity (what means that JAR files, once opened, are never closed).

4.4 Cache Coherency Issues

Perhaps the most important challenge for any caching mechanism is to retain cache coherency. This issue may be addressed in numerous ways, depending on the level of system integration and how strict the coherency requirements are. In the flexible class loader framework we adapted the simple expiration-time oriented algorithm, similar to the one used by HTTP proxies. The `CachingResourceFinder` object has an associated validity-duration value, which denotes how long a given cache entry is considered valid after it has been created. In addition, if the cached entry is a JAR file, it can explicitly override this value providing its own Expiration-Time attribute in a JAR Manifest.

Another issue is related to the JAR files kept open by the `ResourceLoader`. As long as the JAR file remains open, it ignores modifications and even deletion of the actual file from which it was constructed, so the application is not notified that the file changed. To address this issue, the `ResourceLoader` class provides the `invalidate()` method used to notify it that the given JAR file is no longer valid and should be closed. Depending on an additional argument, this either closes the file immediately and invalidates all resource handles derived from it, or it marks the file to be closed as soon as it is no longer referenced by any handles, thus providing an opportunity to be smoothly released. The `CachingResourceFinder` class provides three cache coherency levels based on that feature. At the level HIGH, a change in the cached JAR file causes notification of the loader and immediate invalidation of all derived resource handles. At the level MEDIUM, the JAR file is marked to be closed after the handles have finished. At the level LOW, the loader is never notified about the change so the JAR file can remain open for unspecified periods of time, dependent on loader caching policy.

5 `FlexClassLoader` in Harness Metacomputing Framework

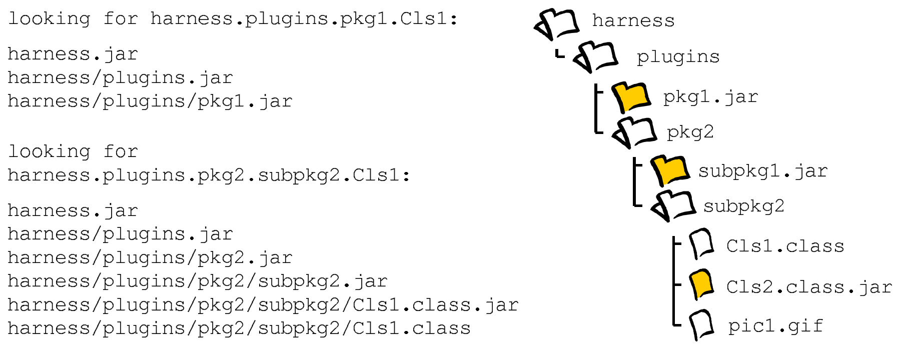
5.1 Hierarchical Resource Lookup

Harness uses a specific class loading scheme: classes can be stored in the set of software repositories, which can be maintained by independent organizations. Each time the class is to be loaded, it is subsequently searched for in the repositories. As long as all classes are stored as an ordinary files and there is a mapping between file paths and package names, they can be easily found just by constructing appropriate lookup URLs (this is the manner in which the current Harness class loader is organized). The only activity required to make a given class visible to Harness is to put it in appropriate place in the repository.

This model gets complicated if we wish to incorporate JAR files as an option to store classes and resources. It is not obvious how JAR files should be stored in repositories to retain simplicity of making the classes and resources available to the Harness system. Most Internet protocols (including HTTP as an important special case) do not provide reliable mechanisms to query for directory contents, so the JAR files must be stored in places where the class loader expects them to be found.

As a solution, we provide the notion of hierarchical resource lookup. It is based on the package-oriented organization of the Java language and the resulting fact that JAR files are encouraged to store single Java packages, which are sets of highly related classes cooperating to provide some service. Making such an assumption a requirement, it becomes possible to establish a contract for the class loader: a JAR file should have the same name as the last part of the package name and it should be stored in the repository in a place where that package is expected. For example, the JAR file containing the package `edu.emory.mathcs.harness` should be named `harness.jar` and be placed on the `edu/emory/mathcs` path in repository.

Fig.3 shows two examples of hierarchical resource lookup, based on the naming contract described above. The algorithm goes through the directory structure, trying to find the class or resource in the JAR file related to the outermost package possible. Finally, it looks for the JAR file containing the class or resource only, and if this fails it tries to load the class or resource in the traditional way. This algorithm is implemented in the `RepositoryResourceFinder` class, which



```
looking for harness.plugins.pkg1.Cls1:

harness.jar
harness/plugins.jar
harness/plugins/pkg1.jar

looking for
harness.plugins.pkg2.subpkg2.Cls1:

harness.jar
harness/plugins.jar
harness/plugins/pkg2.jar
harness/plugins/pkg2/subpkg2.jar
harness/plugins/pkg2/subpkg2/Cls1.class.jar
harness/plugins/pkg2/subpkg2/Cls1.class
```

Fig. 3. Examples of Hierarchical Resource Lookup

extends `CachingResourceFinder` and inherits its caching capabilities. As a result, the exact structure of remote repositories is replicated in the local cache as data is loaded.

5.2 Benefits for Harness

In the Harness system, the class loader plays an important role as it is responsible for providing the background layer for linking "plug-in" services. The new class loading approach will be beneficial for Harness as it takes advantage of JAR files and provides a generic caching mechanisms:

- JAR files can be digitally signed and can contain certified code. This will allow security to be improved in Harness, as administrators will be able to specify security policies based on code certificates.
- JAR files are compressed, so data will be downloaded faster from the network.
- JAR files may contain meta-information related to the JAR file as a whole or to its separate entries. Using this meta-information, Harness will be able to avoid performing complex byte-code analysis of the classes being downloaded (the system performs such analysis in order to decide if it can run the "plug-in" in its own Java VM or if it should spawn another one).
- Complex plug-ins, which contain a number of classes and resources, will be maintained more easily by storing them in a single JAR file. Also, package consistency and versioning will be improved by taking advantage of JAR package attributing and sealing features.

6 Conclusion and Future Work

This paper describes the flexible class loader framework, which simplifies development of application-specific class loaders. The framework consists of a set of customizable classes providing commonly needed functionality. Its intended application in the Harness system is also presented, including the hierarchical resource lookup algorithm supporting loading resources stored in software repositories.

Future work plans include enabling Harness to take advantage of "plug-in" modules containing native code [1,8]. To retain high portability, these modules are intended to contain native source files rather than precompiled libraries; the compilation process would occur "just in time" on the target platform. This technique would benefit from the described class loader framework in several ways: first, the Java classes and native source files comprising the "plug-in" could be bundled together in the JAR file. Next, the code certification and digital signatures could be used to introduce trusted code, which is essential when native code is considered. And finally, the general-purpose caching services would support management and caching of native libraries once they were compiled on a target platform.

References

1. M. Bubak, D. Kurzyniec, and P. Łuszczek. Creating Java to native code interfaces with Janet extension. In M. Bubak, J. Mośсinski, and M. Noga, editors, *Proceedings of the First Worldwide SGI Users' Conference*, pages 283–294, Cracow, Poland, October 11-14 2000. ACC-CYFRONET.
2. The Java extension mechanism. `http://java.sun.com/j2se/1.3/docs/guide/extensions/`.
3. P. Gray and V. Sunderam. IceT: Distributed computing and Java. *Concurrency, Practice and Experience*, 9(11), Nov. 1997. Available at `http://www.mathcs.emory.edu/icet/IceT5.ps`.
4. P. Gray, V. Sunderam, and V. Getov. Aspects of portability and distributed execution of JNI-wrapped code. Available at `http://www.mathcs.emory.edu/icet/sp.ps`.
5. Harness project home page. `http://www.mathcs.emory.edu/harness`.
6. M. Izatt and P. Chan. Ajents: Towards an environment for parallel, distributed and mobile Java applications. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 12-14 1999. Available at `http://www.cs.ucsb.edu/conferences/java99/papers/13-izatt.ps`.
7. Java archive (JAR) features. `http://java.sun.com/j2se/1.3/docs/guide/jar/`.
8. Java Native Interface. `http://java.sun.com/j2se/1.3/docs/guide/jni/`.
9. Java Community Process Java Specification Requests. `http://www.java.sun.com/aboutJava/communityprocess/`.
10. S. Liang and G. Bracha. Dymanic class loading in the Java Virtual Machine. Available at `http://www.java.sun.com/people/gbracha/classloaders.ps`.
11. M. Migliardi and V. Sunderam. The Harness metacomputing framework. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, USA, March 22-24 1999. Available at `http://www.mathcs.emory.edu/harness/PAPERS/pp99.ps.gz`.
12. Java security home page. `http://java.sun.com/j2se/1.3/docs/guide/security/`.