JAVA CLASS LOADING TECHNIQUES IN THE HARNESS METACOMPUTING FRAMEWORK

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 an Java-centric metacomputing system based on a principle of dynamic reconfigurability not only in terms of participating computing resources, but also the capabilities of the virtual machine itself. The central feature of the system is a "plug-in" mechanism built upon Java dynamic class loading services enabling integration of new functionality in the run time. In this paper we describe new flexible, framework based Java class loading techniques and their application in the Harness system.

1. INTRODUCTION

Harness [4, 8] is a metacomputing framework based upon such experimental concepts as dynamic reconfigurability and extensible distributed virtual machine. 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 though powerful architectural model of a software backplane which is configured dynamically by attaching "plug-in" modules providing various services. Those "plug-in" modules are simply packages of cooperating Java classes and possibly related resources, and the dynamic "plug-in" loading is based upon underlying Java class loader capabilities to locate and load classes and resources from the dynamically changing set of software repositories.

Current implementation of Harness class loader fails to satisfy two desired features: to support loading classes from Java archive (JAR) files [5], as they improve code integrity and reliability with *package sealing* [5] and digital signatures, and to supply uniform and reliable caching mechanism reducing potentially expensive network access. This paper overviews the flexible class loader framework, intended to support a wide range of possible dynamic class loading models, and describes its application in the Harness system.

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 to the Java VM classes coming from various non-standard sources such as downloaded from the network, fetched from a database, or even generated on-the-fly. Beginning from the JDK 1.1, class loaders support finding not only classes but also other resources such as images, icons, or sound files. Also, the class signers has been introduced to improve the security model.

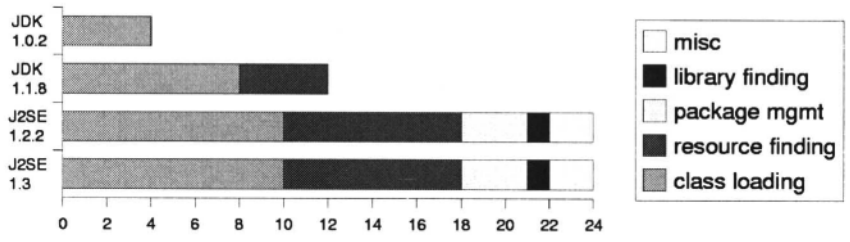

Figure 1. Class Loader API evolution

Java 2 platform brought substantial changes to the class loader capabilities. Its semantics were refined [7] and it has been tightly integrated with the newly established Java Security API [9]. A few new classes were introduced: the `java.security.SecureClassLoader` , intended as a basis for user-defined class loaders incorporating Java security model, and the `java.net.URLClassLoader`, which enables loading data from Java archive (JAR) files [5] and includes support for downloadable extensions [2]. Moreover, in addition to finding and loading classes and resources class loader became responsible to define and keep track of packages and control native library linking. Additionally, the notion of parenthood and request delegation has been introduced improving the run-time model. The evolution of a class loader API (that is, the set of `public` and `protected` methods of class java.lang.Classloader) is illustrated in Figure 1.

3. FLEXIBLE CLASS LOADER FRAMEWORK

Although class loaders are used when there is a need for some application-specific way to find classes, resources, or libraries, they often have much in common. For example, many of them might want to read data from JAR files or download it from the network; similarly, many class loaders treats classes and resources uniformly rather than having separate routines for dealing with them. These similarities, however, do not comprise a common denominator. For that reason, full advantage of a generic class loader can be taken only when its structure is *modular*, so the users can choose and configure the modules they are willing to use in their implementations. Unfortunately, none of the class loaders provided as a part of Java platform is flexible enough to be appropriate for this purpose.

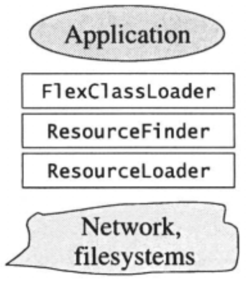

Figure 2. Structure of Flexible Class Loader Framework

The flexible class loader framework presented here is an example of such modular structure, as illustrated in Figure 2. It is based on the observation that the typical class loader, apart from doing its low level duty, must perform independent activities like searching for requested data, being able to download it from the network and manage JAR files. Taking this into account, it becomes reasonable to split the whole 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 full description of the framework is out of scope of this paper. Note, however, that as the most specific part of each class loader is its resource searching algorithm, many sophisticated ones can be derived from this scheme just by customization of the resource finder layer. Also, because many class loaders require the capabilities to cache code downloaded from the network, we introduced the `CachingResourceFinder` providing those as a part of the framework.

4. CLASS LOADING IN HARNESS

Harness uses specific class loading scheme: classes can be retrieved from various software repositories, possibly maintained by independent organizations.

Each time the class is to be loaded, the system searches for it subsequently on the repository search path. As long as all classes are kept in repositories as separate files and there is a direct mapping between directory paths and package names, the lookup can be performed easily just by specifying proper URLs. Thus, the only activity required to make given class visible to Harness system is to put it on appropriate directory path in the software repository.

4.1. Hierarchical Resource Lookup

The simple class lookup model described above gets complicated if we wish to include 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 Harness system. It is important to note that they must be stored in some designated places known by the class loader, because many Internet protocols (including HTTP as an important special case) do not provide reliable mechanisms to query for directory contents.

As a solution, we provide notion of hierarchical resource lookup based on the package-oriented organization of the Java language. The Java deployment strategies encourage to store in JAR files only single packages, consisting of related classes and resources and providing some concrete functionality. Making such JAR file layout a requirement, it becomes possible to establish a naming contract as follows: the name of the JAR file should be the same as the last part of appropriate package name, and the JAR file itself should be stored where directory of that name would be normally expected. For example, JAR file wrapping package `edu.emory.mathcs.harness` should be named `harness.` jar and be placed on the edu/emory/mathcs path.

```
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
```

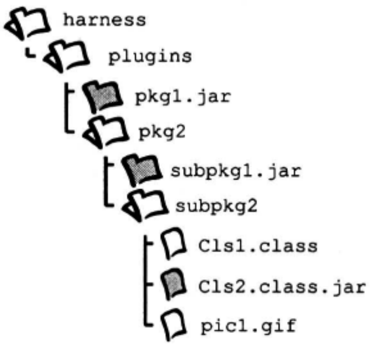

Figure 3. Examples of Hierarchical Resource Lookup

The Figure 3 shows two examples of the hierarchical resource lookup based on such naming contract. When looking for a class or resource, the algorithm goes downward directory structure trying to search JAR files starting from the outermost one possible. If it fails, the traditional lookup method is eventually adopted making the whole approach backward compatible.

The algorithm was implemented in the class derived from the framework's CachingResourceFinder and inheriting its caching capabilities. The new class loader is therefore automatically equipped with uniform and flexible caching mechanism, replicating remote structures of software repositories as the classes and resources are loaded.

4.2. Benefits for Harness

In the Harness system, the class loader plays important role as it is responsible to provide background layer for linking "plug-in" services. The new class loading approach will be beneficial here as it allows to take advantage of JAR file features and as it provides uniform caching mechanism:

- JAR files can be digitally signed and can contain certified code. This will enable security improvement in a Harness system, allowing system administrators to include certificates in security policies.
- JAR files are compressed, so data will be faster downloaded from the network.
- JAR files may contain meta-information related to 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 bunch of classes and resources, will be maintained more easily by storing them in single JAR file. Also, the package consistency and versioning will be improved by taking advantage of JAR package attributing and sealing.

5. CONCLUSIONS AND FUTURE WORK

This paper overviews the flexible class loader framework, which simplifies development of application-specific class loaders and consists of the set of customizable classes providing commonly needed functionality. The ongoing work on the application of the framework in the Harness system is described, including the hierarchical resource lookup algorithm which supports loading resources stored in software repositories.

The future work plans include enabling Harness to take advantage of "plug-in" modules containing native code [1, 6]. To retain high portability, these modules are intended to carry out native source files rather than precompiled libraries; the compilation process would occur "just in time" on the target platform. This technique would benefit from 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 elegant JAR file. Next, the code certification and digital signatures could be used to assure reliability, which is essential when native code is considered. And finally, the general-purpose caching services provided by the new class loader 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. Buba **J. Mościnski**, k, 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/index.html.

[3] 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.

[4] Harness project home page, http://www.mathcs.emory.edu/harness.

[5] Java archive (JAR) features. http://java.sun.com/j2se/1.3/docs/guide/jar/index.html

[6] Java Native Interface. http://java.sun.com/j2se/1.3/docs/guide/jni/index.html.

[7] S. Liang and G. Bracha. Dymanic class loading in the Java Virtual Machine. Available at http://www.java.sun.com/people/gbracha/classloaders.ps.

[8] 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.

[9] Java security home page. http://java.sun.eom/j2se/1.3/docs/guide/security/index.html.