REFERENCES


[76] Srivastava, S., Nandi, G.C., Enhancing the efficiency of secure network monitoring through mobile agents, International Conference on Computer and Communication Technology (ICCCT), 2010


Ibrahim, M.A.M., “Distributed network management with secured mobile agent support”, International Conference On Electronics and Information Engineering (ICEIE), 2010


N. Santos, P. Marques, L. Silva, "Dynamic Deployment of Services on Mobile Agents Systems", in Proc. of the 2nd Workshop on Reflective and Adaptive Middleware (RM’2003), Rio de Janeiro, Brazil, June 2003


APPENDIX

Appendix A: Java Class Loaders

A.1. INTRODUCTION

Class loaders are the entities responsible for loading and linking classes in the Java Virtual Machine, providing separate name spaces for the classes they load. They are also the unit of code loading and unloading. When the class loader and the classes defined by it are no longer referenced by external classes, the garbage collector is free to unload them. Class loaders are first class entities and can be created by programmers, allowing them some control over the process of loading new classes into the virtual machine. They are used by browsers to load applets from the network, and by application servers and servlet engines to load new components. In this work, class loaders are used to load, unload and reload services in the mobile agent agency.

Java has two kinds of class loaders -- the primordial class loader and class loader objects. The **primordial class loader** is sometimes called the system class loader or the default class loader. **Class loader objects** are sometimes called custom class loaders. The primordial class loader (there is only one of these per JVM implementation) is designed and written by the creators of each JVM. Class loader objects are designed and written by Java programmers. Class loader objects are defined as classes (subclasses of java.lang.ClassLoader) and instantiated into regular Java objects on the heap. Whenever the JVM loads a class or interface, it will use either the primordial class loader or a class loader object. Because it is part of the JVM implementation, one (and only one) primordial class loader is always available to a running application. Class loader objects, by contrast, behave like objects. At any one time during an application's lifetime, the application may have zero to many class loader objects in existence and in use.

A.2. OVERVIEW OF CLASS LOADING

The purpose of class loaders is to support dynamic loading of software components on the Java platform. The unit of software distribution is a class. Classes are distributed
using a machine-independent, standard, binary representation known as the *class file format* []. The representation of an individual class is referred to as a *class file*. Class files are produced by Java compilers, and can be loaded into any Java virtual machine. A class file does not have to be stored in an actual file; it could be stored in a memory buffer, or obtained from a network stream.

The Java virtual machine executes the byte code stored in class files. Byte code sequences, however, are only part of what the virtual machine needs to execute a program. A class file also contains symbolic references to fields, methods, and names of other classes. Consider, for example, a class `C` declared as follows:

```java
Class C {
    Void f() {
        D d = new D();
        ...
    }
}
```

The class file representing `C` contains a symbolic reference to class `D`. Symbolic references are *resolved* at link time to actual class types. Class types are first-class objects in the Java virtual machine. A class type is represented in user code as an object of class `java.lang.Class`. In order to resolve a symbolic reference to a class, the Java virtual machine must load the class file and create the class type.

The Java virtual machine uses class loaders to load class files and create class objects. Class loaders are ordinary objects that can be defined in Java code. They are instances of subclasses of the class `ClassLoader`, shown in Figure A.1.
class ClassLoader {

    public Class loadClass (String name);
    protected final Class defineClass (String name, byte[] buf, int off, int len);
    protected final Class findLoadedClass (String name);
    protected final Class findSystemClass (String name);
}

Figure A.1: The ClassLoader class

The ClassLoader.loadClass() method takes a class name as an argument, and returns a Class object that is the run-time representation of a class type. If a Class C is loaded by the ClassLoader L then L is referred as C’s defining loader. The Java virtual machine will use L to load classes referenced by C. Before the virtual machine allocated an object of Class D, it must resolve the reference to D. If D has not yet been loaded, the virtual machine will invoke the loadClass method of C’s class loader, L to load D:

    L.loadClass("D");

Once D has been loaded, the virtual machine can resolve the reference and create an object of class D.

A.3. MULTIPLE CLASS LOADERS

A Java application may use several different kinds of class loaders to manage various software components. For example, Figure A.2 shows how a web browser written in Java may use class loaders. This example illustrates the use of two types of class loaders referred earlier: user-defined class loaders and the system class loader supplied by the Java virtual machine. User-defined class loaders can be used to create classes that originate from user-defined sources. For example, the browser application creates class loaders for downloaded applets. A separate class loader is used for the web browser application itself. All system classes like java.lang.String are loaded by system class loader. The system class loader is supported directly by the Java virtual machine.
The arrows in the figure indicate the delegation relationship between class loaders. A class loader L1 can ask another class loader L2 to load a class C on its behalf. In such a case L1 delegates C to L2. For example, applet and application class loaders delegate all system classes to the system class loader. As a result, all system classes are shared among the applets and the application. This is desirable because type safety would be violated if, for example, applet and system code had a different notion of what the type java.lang.String was. Delegating class loaders allow us to maintain namespace separation while still sharing a common set of classes. In the Java virtual machine, a class type is uniquely determined by the combination of the class name and class loader. Applet and application class loaders delegate to the system class loader. This guarantees that all system class types, such as java.lang.String are unique. On the other hand, a class named C loaded in applet 1 is considered different type from class loaded in applet 2. Although these two classes have the same name, they are defined by different class loaders. In fact, these two classes can be completely unrelated. For example, they may have different
methods or fields.

Classes from one applet cannot interfere with classes in another, because applets are loaded in separate class loaders. This is crucial in guaranteeing Java platform security. Likewise, because the browser resides in a separate class loader, applets cannot access the classes used to implement the browser. Applets are only allowed to access the standard Java API exposed in the system classes.

The Java virtual machine starts up by creating the application class loader and using it to load the initial browser class. Application execution starts in the public class method “void main (String[])” of the initial class. The invocation of this method drives all further execution. Execution of instructions may cause loading of additional classes. In this application, the browser also creates additional class loaders for downloaded applets.

A.4. AN EXAMPLE OF A CLASS LOADERS

As discussed earlier, all user defined class loader classes are subclasses of ClassLoader. Subclass of ClassLoader can override the definition of loadclass(), thus providing a user defined loading policy. Here, shown in Figure A.3, is a class loader which looks for classes in a given directory:

The public constructor MyClassLoader() records the directory name relative to which the classes needed to be loaded by the ClassLoader are placed. In the definition of loadClass() the findLoadedClass() method checks whether the class has already been loaded. If it returns “null” the class is not loaded yet. We then delegate the loading to the system class loader by calling findSystemClass. If the class we are trying to load in not a system class the helper method getClassData() reads the class file.
After we have read in the class file, we pass it to the `defineClass()` method. The `defineClass()` method constructs the run-time representation of the class from the classes file.